

First Steps with **Embedded Systems**

Byte Craft Limited

**BYTE
CRAFT**

02A4 2 40;
02A6 1 s&0x20)
02A9 (able();



CDS

Code Development Systems

The Byte Craft Limited Code Development Systems are high-performance embedded development packages designed for serious developers. They generate small, fast, and efficient code. They enable the professional developer to produce stand-alone single-chip microcontroller applications quickly. Developers can easily port C language applications written for other embedded platforms to the CDS.

www.bytecrafter.com

Features

The Code Development Systems support entire families of microcontrollers. The optimizing C language cross-compilers are ANSI-compatible within hardware limitations. Tight, fast and efficient code optimization generates clean, customized applications. A built-in Macro Assembler allows inline assembly language in C source. CDS generate symbol and source reference information for C source-level debugging with popular emulators. C language support for interrupt service routines and direct access to ports. Device files for individual parts precisely control code generation and resource usage. Complete user documentation comes with every Code Development System. Absolute Code Mode lets you compile directly to final code without a separate linking phase. Alternatively, you can use the BLink Optimizing Linker. Either method performs a final optimization pass on an entire program. Demonstration versions are available from:
<http://www.bytecrafter.com/>

Versatility

Code Development Systems install under Windows 95, 98, ME, NT, 2000, or under MS/PC DOS. CDS provide symbol table information and a listing file: a merged listing of C source and generated assembly language to permit detailed analysis.

info@bytecrafter.com

Do Byte Craft Limited compilers support ANSI C?

All Byte Craft compilers are ANSI compatible within the limitations of the target hardware.

How efficient is the optimizer compared to hand-written assembler code?

The compiler generates object code as tight and efficient as most hand-written assembler code.

Can I combine C code and assembler in my programs?

You can embed assembler code within your C program, using `#asm` and `#endasm` preprocessor directives. The embedded code can call C functions and directly access C variables. To pass arguments conveniently, embed your assembly code in the body of a C function.

What kinds of emulator hardware do the compilers support?

For more information on supported emulator products, contact Byte Craft Limited support staff.

How do the compilers handle local variable declarations?

Our compilers store locally-declared variables in reusable local memory spaces. The scope of local variables is protected.

What are Byte Craft Limited's terms?

For Canada and the U.S.: For company purchases (on approved credit), NET 30 days after shipping. Byte Craft ships next day FedEx free of charge. All other orders must be prepaid, with American Express, VISA, check with order, or direct wire transfer.

For overseas: All orders, prepaid with American Express, VISA, check with order, or direct wire transfer. Shipping is extra. Please call for more information. Please obtain appropriate import documentation.

If for any reason you are unsatisfied with your purchase, you can return it within 30 days for a full refund.



Byte Craft Limited

A2-490 Dutton Drive
Waterloo, Ontario
Canada
N2L 6H7
Tel: 519-888-6911
Fax: 519-746-6751

First Steps with Embedded Systems

by

Byte Craft Limited

BYTE CRAFT LIMITED

A2-490 Dutton Drive

Waterloo, Ontario

Canada N2L 6H7

Telephone: (519) 888-6911

FAX: (519) 746-6751

Email: info@bytecraft.com

<http://www.bytecraft.com>

Copyright © 1997, 2002 Byte Craft Limited. Licensed Material. All rights reserved.

First Steps with Embedded Systems is protected by copyrights. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of Byte Craft Limited.

All example and program code is protected by copyright.

Table of Contents

1. Introduction	1
1.1 Typographical Conventions.....	1
1.2 Explaining the Microcontroller	2
1.3 Book Contents	3
2. Microcontroller Overview	5
2.1 What is a Microcontroller?.....	5
2.2 The Microcontroller in a System.....	7
2.3 Architecture	7
2.3.1 Von Neumann	8
2.3.2 Von Neumann Memory Map.....	8
2.3.3 Harvard	9
2.3.4 Harvard Memory Map.....	10
2.3.5 The Central Processing Unit	11
2.3.6 Central Processing Unit.....	13
2.3.7 ROM.....	14
2.3.8 RAM	15
2.3.9 I/O Ports.....	16
2.3.10 Timer.....	17
2.3.11 Interrupt Circuitry	18
2.3.12 Buses	19
2.4 Sample Microcontroller Configurations.....	19
2.4.1 Motorola MC68HC705C8	19
2.4.2 National Semiconductor COP8SAA7	20
2.4.3 Microchip PIC16C54.....	20
2.4.4 Microchip PIC16C74.....	21

3. The Embedded Environment	23
3.1 The Embedded Difference	23
3.2 Fabrication Techniques	24
3.3 Memory Addressing and Types	24
3.3.1 RAM.....	24
3.3.2 ROM.....	25
3.3.3 PROM	25
3.3.4 EPROM	25
3.3.5 EEPROM	26
3.3.6 Flash Memory.....	27
3.3.7 Registers	27
3.3.8 Scratch Pad	28
3.4 Interrupts	29
3.4.1 Interrupt Handling	30
3.4.2 Synchronous and Asynchronous Interrupt Acknowledgement.....	30
3.4.3 Servicing Interrupts	31
3.4.4 Interrupt Detection	32
3.4.5 Executing Interrupt Handlers.....	33
3.4.6 Multiple Interrupts	34
3.5 Specific Interrupts	34
3.5.1 RESET	35
3.5.2 Software Interrupt/Trap	35
3.5.3 IRQ	36
3.5.4 TIMER.....	36
3.6 Power	37
3.6.1 Brownout	37
3.6.2 Halt/Idle	37
3.7 Input and Output	37
3.7.1 Ports.....	37
3.7.2 Serial Input and Output.....	38
3.8 Analog to Digital Conversion	40
3.9 Miscellaneous	41
3.9.1 Digital Signal Processor	41
3.9.2 Clock Monitor	41

3.10 Devices	41
3.10.1 Mask ROM.....	41
3.10.2 Windowed Parts	41
3.10.3 OTP.....	41
 4. Programming Fundamentals	 43
4.1 What is a Program?	43
4.2 Number Systems	43
4.3 Binary Information	44
4.4 Memory Addressing	46
4.5 Machine Language	46
4.6 Assembly Language	46
4.6.1 Assembler.....	47
4.7 Instruction Sets	47
4.8 The Development of Programming Languages	48
4.9 Compilers	50
4.9.1 The Preprocessor	50
4.9.2 The Compiler	50
4.9.3 The Linker	50
4.10 Cross Development	51
4.10.1 Cross compiler.....	51
4.10.2 Cross development tools	51
4.10.3 Embedded Development Cycle.....	52
 5. First Look at a C Program	 55
5.1 Program Comments	56
5.2 Preprocessor directives	56
5.3 C Functions	58
5.3.1 The main() function.....	58
5.3.2 Calling a Function	59

5.4 The Function Body.....	60
5.4.1 The Assignment Statement	60
5.4.2 Control statements	60
5.4.3 Calling Functions	62
5.5 The Embedded Difference.....	62
5.5.1 Device Knowledge	63
5.5.2 Special Data Types and Data Access.....	63
5.5.3 Program Flow.....	63
5.5.4 Combining C and Assembly Language	63
5.5.5 Mechanical Knowledge.....	64
 6. C Program Structure	 65
6.1 C Preprocessor Directives.....	65
6.2 Identifier Declaration	65
6.2.1 Identifiers in Memory	66
6.2.2 Identifier names	66
6.2.3 Variable Data Identifiers.....	67
6.2.4 Constant Data Identifiers	67
6.2.5 Function Identifiers.....	68
6.3 Statements	68
6.3.1 The Semicolon Statement Terminator	69
6.3.2 Combining Statements in a Block.....	69
 7. Basic Data Types	 71
7.1 The ASCII Character Set	71
7.2 Data types	71
7.3 Variable Data Types	72
7.3.1 Variable Data Type Memory Allocation	72
7.3.2 Variable Scope.....	73
7.3.3 Global Scope	74
7.3.4 Local Scope.....	74
7.3.5 Declaring Two Variables with the Same Name	74
7.3.6 Why Scope is Important.....	75
7.4 Function Data Types.....	75
7.4.1 Function Parameter data types	76

7.5 The Character Data Type.....	76
7.5.1 Assigning a character value.....	76
7.5.2 ASCII Character Arrangement	77
7.5.3 Numeric Characters	77
7.5.4 Escape Sequences.....	77
7.6 Integer Data Types.....	78
7.6.1 Integer Sign Bit.....	78
7.6.2 The short Data Type	78
7.6.3 The long Data type	79
7.6.4 Different Notations	79
7.7 Data Type Modifiers	79
7.7.1 Signed and Unsigned	80
7.7.2 Other Data Type Modifiers.....	80
7.8 Real Numbers	80
7.8.1 The float Data Type.....	81
7.8.2 The double and long double Types.....	81
7.8.3 Assigning an Integer to a float	81
 8. Operators and Expressions	 83
8.1 Operators	83
8.2 C Expressions.....	84
8.2.1 Binding.....	85
8.2.2 Unary Operators.....	85
8.2.3 Binary Operators	85
8.2.4 Trinary Operator	86
8.2.5 Operator Precedence	86
8.2.6 The = Operator.....	87
8.3 Arithmetic Operators.....	88
8.3.1 Increment and Decrement Operators	89
8.4 Assignment Operators.....	90
8.5 Comparison Operators	91
8.5.1 Expressing True and False.....	91
8.5.2 The Equality Operators.....	92
8.5.3 Relational Operators.....	92
8.5.4 Logical Operators.....	93

8.6 Bit Level Operators.....	95
8.6.1 Bit Logical Operators.....	95
8.6.2 Bit shift operators	97
 9. Control Structures	 99
9.1 Conditional Expressions.....	99
9.2 Decision Structures.....	100
9.2.1 if and else Statements.....	100
9.2.2 Nested if statements.....	101
9.2.3 Matching else and if.....	102
9.2.4 switch and case.....	103
9.2.5 Execution within a switch	103
9.2.6 Fall-through execution.....	104
9.2.7 The default case	105
9.2.8 The goto Statement	105
9.2.9 Comparing goto and switch..case.....	106
9.3 Looping Structures	106
9.3.1 Control expression	106
9.3.2 The while loop.....	107
9.3.3 The do loop	107
9.3.4 The for loop.....	108
9.3.5 How the for loop works	108
9.4 Exiting a Loop.....	109
9.4.1 The break Statement	109
9.4.2 The continue Statement.....	109
 10. Functions	 111
10.1 main()	111
10.2 Executing a Function	111
10.2.1 Calling a Function.....	112
10.3 Function Prototype Declarations	113
10.3.1 Defining the Function Interface	113
10.3.2 Calling Functions in Other Files	113
10.3.3 Function Type, Name and Parameter List	114
10.3.4 Functions and void.....	115

10.4 Function Definitions.....	116
10.4.1 Statement Block.....	116
10.4.2 Variable Declarations in Function Definitions	116
10.5 Function Parameters.....	117
10.5.1 Passing Data by Value.....	117
10.5.2 Passing Data by Reference	117
10.5.3 Functions Without Parameters.....	118
 11. Complex Data Types.....	 121
11.1 Pointers.....	121
11.1.1 Declaring a Pointer	121
11.1.2 Pointer Operators	122
11.1.3 Pointer Pitfalls	123
11.2 Arrays	124
11.2.1 Accessing Array Elements	124
11.2.2 Multidimensional Arrays	125
11.2.3 Array Operations and Pointer Arithmetic	125
11.2.4 Arrays of Pointers	126
11.3 User Defined Data Types	127
11.3.1 Using typedef to Define New Data Types.....	127
11.3.2 Using types defined with typedef	128
11.4 Enumerated Types.....	128
11.4.1 Enumerated Type Elements.....	129
11.4.2 Enumerated Type Value Checks	129
11.4.3 Specifying Values for Enumerated Elements.....	130
11.5 Structures	131
11.5.1 The structure tag	131
11.5.2 Using typedef to Define a Structure	132
11.5.3 Accessing Structure Members.....	132
11.5.4 Indicating a Field with the Dot Operator	132
11.5.5 Indicating a Field with the Structure Pointer	133
11.5.6 Bit Fields in Structures	133
11.5.7 Storing bit fields in memory.....	134
11.5.8 The behaviour of bit fields	134
11.6 Unions	135
11.6.1 Retrieving a Union Element.....	136

11.6.2 Using Unions with Incompatible Variables	137
12. Storage and Data Type Modifiers	139
12.1 Storage Class Modifiers.....	139
12.1.1 External linkage.....	139
12.1.2 Internal linkage.....	139
12.1.3 No linkage.....	140
12.1.4 The extern Modifier	140
12.1.5 Global Variables and extern.....	141
12.1.6 The static Modifier	142
12.1.7 The visibility of static variables	142
12.1.8 The register Modifier	143
12.1.9 The auto Modifier.....	144
12.2 Data Type Modifiers	145
12.2.1 Value Constancy Modifiers: const and volatile.....	145
12.2.2 Allowable Values Modifiers: signed and unsigned.....	146
12.2.3 Size Modifiers: short and long.....	146
12.2.4 Pointer Size Modifiers: near and far	147
12.2.5 Using near and far pointers.....	148
12.2.6 Default pointer type	148
13. The C Preprocessor	151
13.1 Preprocessor Directive Syntax	151
13.2 White Space in the Preprocessor	152
13.3 File Inclusion	152
13.3.1 File Inclusion Searches	153
13.4 Defining Symbolic Constants	153
13.4.1 The #undef directive	154
13.4.2 Defining “empty” symbols	155
13.5 Defining Macros	155
13.5.1 Macro Expansion.....	156
13.5.2 # and ## Operators.....	157
13.6 Conditional Source Code	157
13.6.1 #if and #endif.....	157
13.6.2 The defined() Function.....	158

13.6.3 The #else and #elif Directives	158
13.6.4 #ifdef and #ifndef	159
13.7 Producing Error messages	159
13.8 Defining Target Hardware	160
13.9 In-line Assembly Language	160
13.9.1 The #asm and #endasm Directives	160
14. Libraries	161
14.1 Portable Device Driver Libraries	161
14.2 An Example Development Scenario	162
14.2.1 How SPI Works	163
14.2.2 SPI_set_master(<i>ARGUMENT</i>);	164
14.2.3 SPI_send_rec(0,4);	166
14.3 Device Driver Library Summary	168
15. Sample Project	169
15.1 Project Specifics	169
15.2 Project Foundations	169
15.2.1 Asynchronous	169
15.2.2 SCI	170
15.2.3 RS-232	170
15.3 Electrical Specifications	171
15.4 PIC Implementation	171
15.4.1 Anatomy of a PC serial port	171
15.4.2 A Note On Chip Sets	172
15.4.3 IRQ	172
15.5 Programming Interrupts	177
15.6 The Sample Project Code	179
15.6.1 PIC16C74 Code	179
15.6.2 PC Code	180

16. C Precedence Rules	185
17. ASCII Chart	187
18. Glossary	189
19. Bibliography	197
20. Index	198

Table of Examples

Example 1: Defining ports with #pragma directives	17
Example 2: Using a union structure to create a scratch pad	28
Example 3: Using globally allocated data space in a function	29
Example 4: A typical assembly language program for the COP8SAA	49
Example 5: Program in Example 4 compiled for the 68HC705C8	49
Example 6: A typical microcontroller program	55
Example 7: Syntax for the main() function	59
Example 8: Using the C assignment statement	60
Example 9: The if statement syntax	61
Example 10: Nesting if and while statements	62
Example 11: Calling one function from another	62
Example 12: C functions containing inline assembly language	64
Example 13: Common C keywords	66
Example 14: Using braces to delineate a block	70
Example 15: The while loop	70
Example 16: Declaring variable types	72
Example 17: Assigning a character value	76
Example 18: Octal, hex and binary notation	79
Example 19: Data type modifiers	80
Example 20: Postfix and prefix unary operators	85
Example 21: Sample binary operators	85
Example 22: Trinary conditional operator	86
Example 23: Combining operators in a statement	86
Example 24: Concatenating expressions with the comma operator	87
Example 25: Combining assignment operators in statements	87
Example 26: Addition, subtraction and multiplication operators	88
Example 27: Division and modulus operators	88
Example 28: Differentiating the division and modulus operators	88
Example 29: Prefix and postfix notation for increment and decrement	89
Example 30: Postfix increment and decrement	89
Example 31: Using prefix increment and decrement	90
Example 32: Variations on the assignment statement	91
Example 33: Defining constant values for true and false	91
Example 34: Defining constant values for true and false in a portable way	92

Table of Examples

Example 35: Using the equality operator in control structures	92
Example 36: The inequality operator.....	92
Example 37: Logical NOT and AND operators	93
Example 38: Using the or operator.....	94
Example 39: Sort circuit expression evaluation	94
Example 40: Using short-circuit evaluation.....	94
Example 41: Bitwise AND operation using &.....	95
Example 42: Using the AND bitwise operator with binary values	96
Example 43: Using the bitwise OR operator 	96
Example 44: The bitwise XOR operator	96
Example 45: The bitwise NOT operator	97
Example 46: Shifting bits left and right.....	97
Example 47: Controlling loops without using logical operators	100
Example 48: if and else structure	100
Example 49: Using the if statement structure	100
Example 50: The else statement.....	101
Example 51: Nesting if statements.....	101
Example 52: Converting nested if statements to logical expressions	102
Example 53: Matching if and else statements	102
Example 54: Using braces to clarify the combination of if and else.....	102
Example 55: An alternate format for showing if else pairing	103
Example 56: The switch..case structure	103
Example 57: Using the fall-through effect with switch statements.....	104
Example 58: Multiple case enhancement.....	105
Example 59: Using the default case value.....	105
Example 60: The goto statement	106
Example 61: The while loop syntax.....	107
Example 62: The do loop syntax.....	107
Example 63: Comparing the while and for loops.....	108
Example 64: Using the for loop	108
Example 65: Comparing function and variable declarations	114
Example 66: The function statement block.....	116
Example 67: Variable declarations inside functions	116
Example 68: Passing data to a function by value.....	117
Example 69: Passing a variable to a function by address (reference)	118
Example 70: Using the address of operator.....	122
Example 71: Using the pointer dereference operator	123

Example 72: Dereferencing a pointer set to NULL.....	123
Example 73: Initializing a pointer	124
Example 74: Array operations and pointer arithmetic.....	125
Example 75: The relationship between arrays and pointers	126
Example 76: Declaring and initializing an array of pointers	127
Example 77: Using typedef to define a new data type.....	127
Example 78: Defining a new enumerated type	128
Example 79: Declaring multiple variables of the same enumerated type	129
Example 80: Enumerated types as integer values.....	129
Example 81: Testing the value of an enumerated type.....	129
Example 82: Specifying integer values for enumerated elements	130
Example 83: Specifying a starting value for enumerated elements	130
Example 84: The assignment of integer values to an enumerated list.....	130
Example 85: Declaring the template of a structure.....	131
Example 86: Declaring a structure without a tag.....	131
Example 87: Using typedef to clarify structure declaration	132
Example 88: Accessing elements in a structure	132
Example 89: A structure accessed with a pointer.....	133
Example 90: Bit fields in structures	134
Example 91: Accessing bit fields.....	134
Example 92: Compiler dependant storage of bit fields	134
Example 93: Declaring a union	135
Example 94: Using typedef to declare a union	135
Example 95: Using a union to create a scratch pad	136
Example 96: Using a union to access data as different types.....	136
Example 97: Accessing a union element with the dot operator.....	136
Example 98: Using the right arrow operator to access a union member.....	136
Example 99: Returning the low Byte of a word.....	137
Example 100: Returning a specific part of a word for little endian	137
Example 101: Incompatible variables with different storage methods in unions.....	138
Example 102: Restricting a function's scope by declaring it as extern.....	141
Example 103: Using preprocessor directives to declare extern global variables.....	142
Example 104: Using the static data modifier to restrict the scope of variables	142
Example 105: Using static variables to track function depth.....	143
Example 106: Using the register data type modifier	143
Example 107: Using the auto data modifier.....	144
Example 108: The far pointer type as default	149

Table of Examples

Example 109: Nesting preprocessor directives	151
Example 110: Redefining a constant using #undef.....	155
Example 111: Defining and calling a macro.....	156
Example 112: Using #if and #endif to conditionally compile code	157
Example 113: Using expressions in #if directives for conditional compilation	158
Example 114: Using the defined() function for conditional compilation	158
Example 115: Using !defined() to test if a symbol has not been defined.....	158
Example 116: Using #else and #elif to choose between compilation blocks.....	159
Example 117: Using #elif, #if and #endif for conditional compilation.....	159
Example 118: Using #ifdef and #ifndef.....	159
Example 119: Using the #error directive.....	160
Example 120: Master function for PIC16C74 SPI communication	163
Example 121: Setting up the SPI on the Microchip PIC16C74	165
Example 122: Setting up SPI on the Motorola 68HC705C8	165
Example 123: Setting up SPI on the National COP8SAA7.....	166
Example 124: Initiating SPI send/receive on the Microchip PIC16C74.....	167
Example 125: Initiating SPI send/receive on the Motorola 68HC705C8.....	167
Example 126: Initiating SPI send/receive on the National COP8SAA7.....	168
Example 127: Serial port connection example for the PIC16C74	180
Example 128: Serial port connection example for the PC	182

Table of Figures

Figure 1: The microcontroller.....	7
Figure 2: Von Neumann memory map for the MC68705C8	9
Figure 3: Harvard memory map PIC16C74	10
Figure 4: Harvard memory map COP8SAA7	11
Figure 5: Instruction clocking on the PIC16C54.....	12
Figure 6: The CPU	13
Figure 7: MC68HC705C8 stack	15
Figure 8: Saving the machine state on the MC68HC705C8	33
Figure 9: Data storage VS. data value.....	45
Figure 10: RS-232 signal	170
Figure 11: Project schematic.....	182

Table of Tables

Table 1: Hardware characteristics of the Motorola MC68HC705C8	20
Table 2: Hardware characteristics of the National Semiconductor COP8SAA7.....	20
Table 3: Hardware characteristics of the Microchip PIC16C54.....	21
Table 4: Hardware characteristics of the Microchip PIC16C74.....	21
Table 5: Sample vectored interrupts	32
Table 6: Binary, decimal and hexadecimal.....	44
Table 7: Interpretation of assembly language.....	47
Table 8: Instruction set comparisons.....	48
Table 9: Pointers and pointers-to-pointers	122
Table 10: PC serial port addresses and interrupts.....	171
Table 11: UART chips	172
Table 12: COM port registers	173
Table 13: Interrupt enable register bits	174
Table 14: Interrupt identification register	175
Table 15: FIFO control register	175
Table 16: Line Control Register	176
Table 17: Modem Control Register	176
Table 18: Line Status Register.....	177
Table 19: Modem Status Register.....	177
Table 20: Pin outs on the RS232 port	183
Table 21: Rules of operator precedence.....	185
Table 22: ASCII characters	187

Acknowledgements

This book represents the hard work of many people at Byte Craft Limited. We want to offer as much of our experience as possible to those entering the Embedded Systems field. We are leveraging our experience in embedded systems, in technical communication, and in publishing to bring about informative publications that do just that.

Kirk Zurell edited this publication and designed the cover art.

1. Introduction

This book is intended to fill the need for an intermediate level overview of programming microcontrollers using the C programming language. It is aimed specifically at two groups of readers who have different, yet overlapping needs.

- ① The first group are familiar with C but require an examination of the general nature of microcontrollers: what they are, how they behave and how best to use the C language to program them.
- ② The second group are familiar with microcontrollers but are new to the C programming language and wish to use C for microcontroller development projects.

First Steps with Embedded Systems will be useful both as an introduction to microcontroller programming for intermediate level post-secondary programs and as a guide for developers coping with the growth and change of the microcontroller industry.

1.1 Typographical Conventions

Bold is used to indicate key terms.

Italic is used for emphasis and to denote references to documents.

`Courier` is used for sample code and code excerpts.

Courier is used to indicate place holders in user input or in output produced by the software. For example, the filename *START.ext* has an italicised extension which indicates that the file can have any valid extension.

-- the double underscore contains a small space to display both characters. Do not type the space in the double underscore character in your code.



is used within one section to refer to another section on a related topic.

NOTE

An important note will appear in this way.

0x is used to denote a hexadecimal number. For example: 0xFFFF

0b is used to denote a binary number. For example: 0b010101

1.2 Explaining the Microcontroller

Instead of presenting a detailed examination of a specific microcontroller or microcontroller family, *First Steps with Embedded Systems* explains concepts which are common to *most* 8 bit microcontrollers. This book will focus on several specific parts for example purposes. These include Motorola's MC68HC705C8, National Semiconductor's COP8SAA7 and Microchip's PIC16C54 and PIC16C74.

The industry provides a large array of speciality microcontroller configurations with optional features and feature combinations. However, many 8 bit microcontrollers have a common underlying architecture. This book examines this common architecture and guides you through the issues you need to understand in order to program a microcontroller. Learning common microcontroller architecture has several important advantages:

★ You will not be overwhelmed by details

Microcontrollers have a set of common, general features. These general features form an essential preliminary foundation for learning specific microcontroller implementations. Variations, options and specific implementations offered by various microcontrollers are also included for example purposes.

★ You will learn the basics of portability

One advantage of using C to program microcontrollers is program portability. Each microcontroller has an individual instruction set and assembly language. Modifying assembly language code so a program written for one microcontroller will run on a different microcontroller is very time consuming and effort intensive.

Writing C code that supports general microcontroller features helps to avoid portability problems. Details relating to specific hardware implementations can be placed in separate library functions and header files. Using C library functions and header files ensures that application source code can be re-compiled for different microcontroller targets.

★ You can spend more time on algorithm design and less on implementation

C is a high level language. You will be able to program your applications quickly and easily using C. C's breadth of expression is concise and powerful; therefore,

each line of code written in C can replace many lines of assembly language. Debugging and maintaining code written in C is much easier than in assembly language code.

1.3 Book Contents

Section 2, Microcontroller Overview, describes the standard microcontroller and covers the basic components of a microcontroller.

Section 3, The Embedded Environment, describes basic microcontroller concepts such as input, output, interrupts, timing and memory.

Section 4, Programming Fundamentals, includes brief explanations of basic topics such as number systems, languages and development tools.

Section 5, First Look at a C Program, provides a sample C program and then examines the basic components represented by the example.

Section 6, C Program Structure, covers the main components of a C program: directives, identifiers and statements.

Section 7, Basic Data Types, covers the different data types and how to use them with variables and functions.

Section 8, Operators and Expressions, covers arithmetic, assignment, comparison and bit level C operators and expressions.

Section 9, Control Structures, covers conditional expressions and decision and looping structures.

Section 10, Functions, covers defining, prototyping, calling and declaring C functions. This section also examines function parameters.

Section 11, Complex Data Types, covers pointers, arrays, user defined types, enumerated types, structures and bitfields.

Section 12, Storage and Data Type Modifiers, covers modifiers which specify location, value, size, and sign of data types.

Section 13, The C Preprocessor, covers C preprocessor directives and related issues such as file inclusion, target hardware definition, conditional compilation, and inline assembly.

Section 14, Libraries, describes the standard embedded systems libraries.

Section 15, Sample Project, follows the development of a small sample microcontroller project

2. Microcontroller Overview

This section provides a brief overview of general microcontroller features and resources. It is designed to familiarise you with microcontroller terminology and basic microcontroller architecture. Many of the concepts introduced in this section will be revisited throughout the book.

2.1 What is a Microcontroller?

A **microcontroller** is a single chip, self-contained computer which incorporates all the basic components of a personal computer on a much smaller scale. Microcontrollers are often referred to as single chip devices or single chip computers. The main consequence of the microcontroller's small size is that its resources are far more limited than those of a desktop personal computer.

In functional terms, a microcontroller is a programmable single chip which controls a process or system. Microcontrollers are typically used as embedded controllers where they control part of a larger system such as an appliance, automobile, scientific instrument or a computer peripheral. Microcontrollers are designed to be low cost solutions; therefore using them can drastically reduce part and design costs for a project.

Physically, a microcontroller is an integrated circuit with pins along each side. The pins presented by a microcontroller are used for power, ground, oscillator, I/O ports, interrupt request signals, reset and control. In contrast, the pins exposed by a microprocessor are most often memory bus signals (rather than I/O ports).

NOTE

A microcontroller is not the same as a microprocessor. A microprocessor is a single chip CPU used within other computer systems. A microcontroller is itself a single chip computer system.

Personal computers are used as **development platforms** for microcontroller projects. Development computers, usually personal or workstation computers, use a **microprocessor** as their principle computing engine. Microprocessors depend upon a variety of subsidiary chips and devices to provide the resources not available on the microprocessor. Additional chips required with a

microprocessor support memory storage, input/output control and specialized processing.

A development platform is required to run embedded system development software such as assemblers, compilers, editors and simulators which require the processing power and memory capabilities of a desktop personal computer or workstation.

The **target platform** is the platform on which the finished program will be run. For example, consider a developer who is creating a program for a Motorola 68HC705C8 microcontroller. The developer writes, edits, and tests the program on a Pentium 133 personal computer: the development platform. The developer will use software which runs on a Pentium 133 but whose target device is the 68HC705C8. When the program is ready it is programmed in the target platform, the 68HC705C8.

A microcontroller has seven main components:

- ① Central processing unit (CPU)
- ② ROM
- ③ RAM
- ④ Input and Output
- ⑤ Timer
- ⑥ Interrupt circuitry
- ⑦ Buses

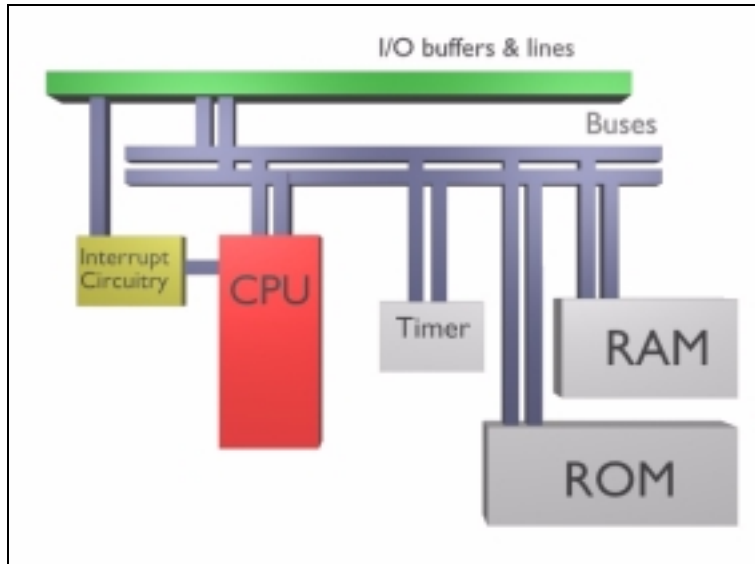


Figure 1: The microcontroller

2.2 The Microcontroller in a System

Microcontrollers do not function in isolation. As their name suggests they are designed to control other devices. The microcontroller can accept inputs from some devices and provide outputs to other devices within any given system. For example, a microcontroller may accept input from a switch and may send output to an LED. If the switch is pressed the microcontroller can be instructed to illuminate the LED.

The microcontroller is often part of a larger system. For example, the switch and LED may be part of a compact disc player in a car stereo system. When a microcontroller is part of a larger system it is often referred to as an embedded controller because it is embedded within the larger system.

2.3 Architecture

There are two basic types of architecture: Harvard and Von Neumann. Microcontrollers most often use a Harvard or a modified Harvard-based architecture.

2.3.1 Von Neumann

Von Neumann architecture has a single, common memory space where both program instructions and data are stored. There is a single data bus which fetches both instructions and data. Each time the CPU fetches a program instruction it may have to perform one or more read/write operations to data memory space. It must wait until these subsequent operations are complete before it can fetch and decode the next program instruction. The advantage to this architecture lies in its simplicity and economy.

NOTE

On some Von Neumann machines the program can read from and write to CPU registers, including the program counter. This can be dangerous as you can point the PC at memory blocks outside program memory space. Careless PC manipulation can cause errors which require a hard reset.

2.3.2 Von Neumann Memory Map

Every microcontroller has a very specific layout for its memory. Usually this is depicted with the help of a memory map. A memory map is a diagram which shows how the microcontroller memory is used. The following example map is from the Motorola MC68HC705C8 microcontroller configured for 176 bytes of RAM and 7744 bytes of PROM:

Contents	Address
I/O 32 bytes	0x0000 0x001F
User Prom 48 bytes	0x0020 0x004F
176 Bytes of RAM	0x0050 0x00BF
STACK	0x00C0 0x00FF
User PROM 96 bytes	0x0100 0x015F
User PROM 7584 bytes	0x0160 0x1EFF
Boot ROM 223 bytes	0x1F00 0x1FDE
Option Register	0x1FDF
Boot ROM vectors 16 bytes	0x1FE0 0x1FEF
Unused 4 bytes	0x1FF3
User PROM vectors 12 bytes	0x1FF4 0x1FFF

Figure 2: Von Neumann memory map for the MC68705C8

2.3.3 Harvard

Harvard architecture computers have separate memory areas for program instructions and data. There are two or more internal data buses which allow simultaneous access to both instructions and data. The CPU fetches instructions on the program memory bus. If the fetched instruction requires an operation on data memory, the CPU can fetch the *next* program instruction while it uses the data bus for its data operation. This speeds up execution time at the cost of more hardware complexity.

Since Harvard machines assume that only instructions are stored in program memory space, how do you write and access data stored in program memory space? For example, a data value declared as a C constant must be stored in ROM as a constant value. Different microcontrollers have different solutions to this problem. A good C compiler automatically generates the code to suit the target hardware's requirements.

Some chips have special instructions allowing the retrieval of information from program memory space. These instructions are always more complex or expensive than the equivalent instructions for fetching data from data memory.

Typically these chips have a register analogous to the program counter (PC) which refers to addresses in program space. Also, some chips support the use of any 16 bit value contained in data space as a pointer into the program address space. These chips have special instructions to use these data pointers.

NOTE

It is important that you understand how your Harvard architecture part deals with data in program space. It is possible to generate more efficient code using symbolic constants declared with `#define` directives instead of declared constants. You may also create global variables for constant values.

2.3.4 Harvard Memory Map

The following memory map is from the Microchip PIC16C74. Notice that program memory is paged and data memory is banked. The stack is implemented in hardware and the developer has no access to it.

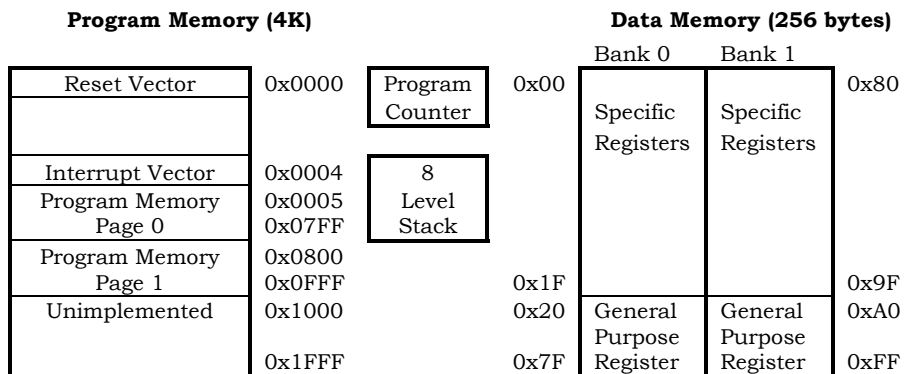


Figure 3: Harvard memory map PIC16C74

The following is the memory map for the COP8SAA7. The stack grows down from the top of general purpose RAM.

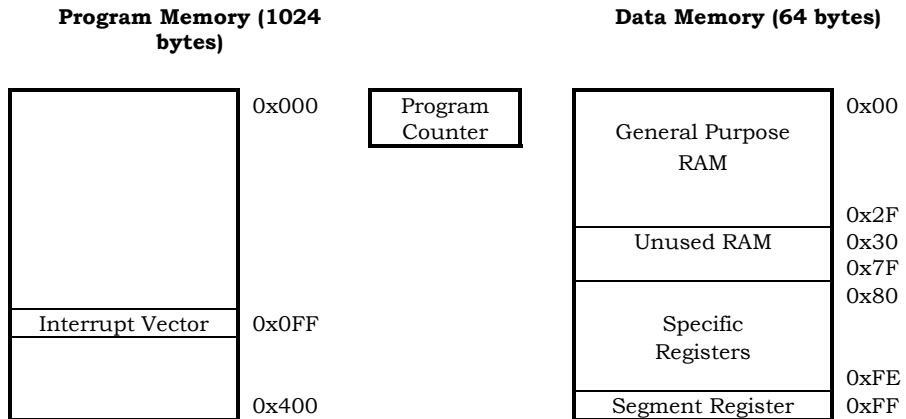


Figure 4: Harvard memory map COP8SAA7

2.3.5 The Central Processing Unit

The **central processing unit** (CPU) does all the computing; it fetches, decodes and executes program instructions and directs the flow of data to and from memory. The CPU performs the calculations required by program instructions and places the results of these calculations, if required, into memory space.

Most CPUs are **synchronous**. This means that they depend on the cycles of a **processor clock**. A clock generates a high-frequency square wave usually driven by a crystal, a RC (resistor capacitor) or an external source. The clock is sometimes referred to as an oscillator. The clock speed, or oscillation rate, is measured in megahertz (MHz) which represents one million cycles/second. For example, if the clock speed is 3 MHz then there are 3,000,000 clock cycles/second.

Clock configurations are microcontroller dependant. The following are some sample clock configurations:

- The National Semiconductor COP8SAA7 has four clock options: crystal with bias resistor, crystal without bias resistor, R/C, and external. The option is selected with bits 3 and 4 of the ECON register. The CK1 and CK0 pins are used for clock related input and output.
- The Motorola MC68HC705C8 has two pins, OSC1 and OSC2, which provide connections for an on-chip oscillator. A crystal, ceramic resonator, or external signal can be attached to the pins. The oscillator frequency is

two times the internal bus rate and the processor clock cycle is two times the oscillator frequency.

- The Microchip PIC16C54 has clock input pin OSC1/CLKIN and clock output pin OSC2/CLKOUT. OSC1/CLKIN is internally divided by four to generate four clocks. There are four possible modes: low power crystal, crystal/resonator, high speed crystal, resistor/capacitor.

The clock controls the sequence of instructions. Most microcontrollers divide their basic clock frequency to arrive at a bus-rate clock. Each instruction takes a specific number of bus-rate clock cycles in order to execute. The following depicts the clocking scheme for the Harvard architecture Microchip PIC16C54:

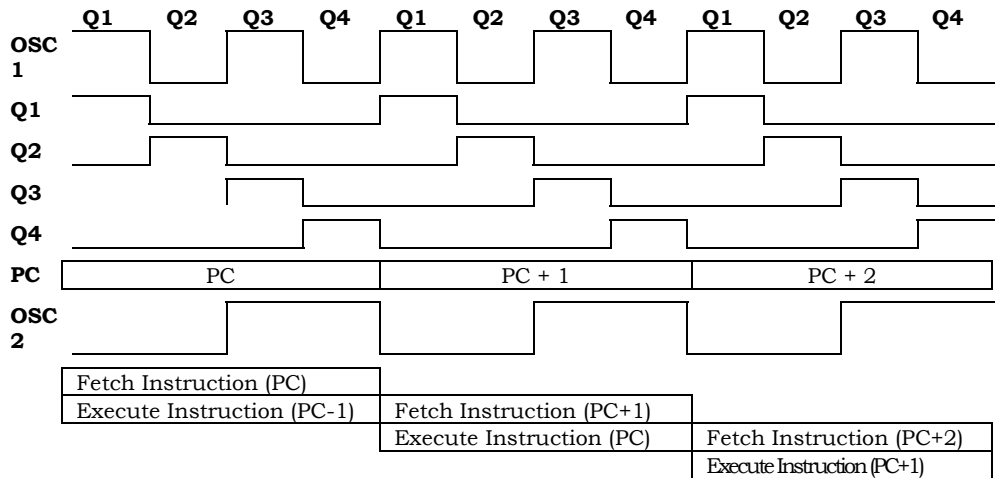


Figure 5: Instruction clocking on the PIC16C54

2.3.6 Central Processing Unit

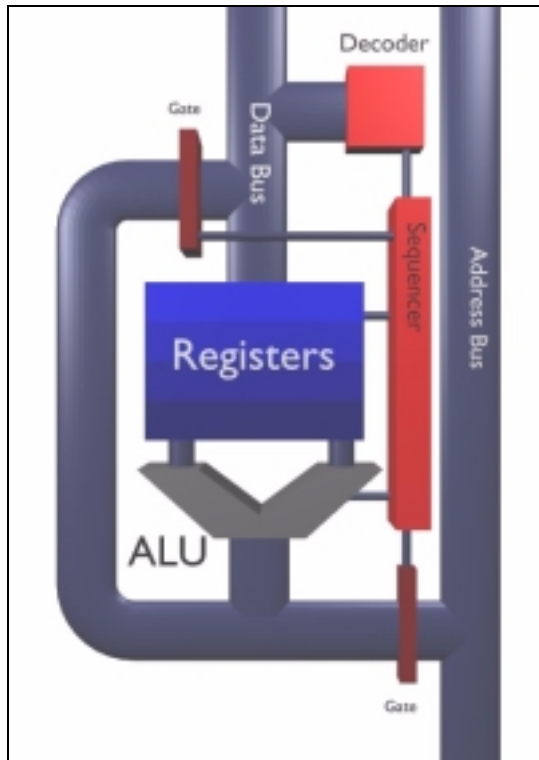


Figure 6: The CPU

One part of the CPU is responsible for performing calculations and executing instructions. This part is called the **arithmetic logic unit**, or ALU. There are a variety of subsidiary components which support the ALU. These components include the decoder, the sequencer and a variety of registers.

The **decoder** converts instructions stored in program memory into codes which the ALU can understand. The **sequencer** manages the flow of data along the microcontroller's data bus. **Registers** are used by the CPU to temporarily store vital data which are volatile: they can change during program execution. Most microcontroller registers are memory-mapped, associated with a memory location, and can be used like any other memory location.

Registers store the state of the CPU. If the contents of microcontroller memory and the contents of these registers are saved it is possible to suspend program operation for an indefinite period of time and restart exactly in the state when the program was suspended.

The number and names of registers varies drastically among microcontrollers. However there are certain registers which are common to most microcontrollers, although the names may vary. These include:

- The **stack pointer**
The stack pointer contains the address of the next location on the stack. The address in the stack pointer is decremented when data is pushed on the stack and incremented when data is popped from the stack.
- The **index register**
The index register is used to specify an address when certain addressing modes are used. It is also known as the pointer register. The Microchip devices use the name FSR (file select register).
- The **program counter**
Perhaps the single most important CPU register is the program counter (PC). The PC holds the address of the next instruction in program memory space. It contains the address of the next instruction the CPU will process. As each instruction is fetched and processed by the ALU, the CPU increments the PC and thereby steps through the program stored in the program memory space.
- The **accumulator**
The accumulator is a register that can hold operands or results of operations as necessary. The Microchip devices use the name W (working) register.

Other registers may reflect results from the instruction just executed, control the options available on the device, and enable access to certain areas of memory.

2.3.7 ROM

ROM, read only memory, is non-volatile memory used for program information and permanent data. The microcontroller uses ROM memory space to store program instructions it will execute when it is started or reset. Program instructions must be saved in non-volatile memory so that they are not

affected by loss of power. The microcontroller usually cannot write data to program memory space.

2.3.8 RAM

RAM, random access memory, is used to write and read data values as a program runs. RAM is volatile: if you remove the power supply its contents are lost. Any variables used in a program are allocated from RAM.

The time to retrieve information from RAM does not depend upon the location of the information because RAM is not sequential, hence the term **random** access.

Most small microcontrollers provide very little RAM which forces you to write applications that use RAM wisely. Manipulating large data structures and using pointers, re-entrant or recursive functions use large amounts of RAM and are techniques which are generally avoided on microcontrollers.

Some C instructions which are rarely used on larger platforms are more commonly used in C programs for microcontrollers. One example is the `goto` instruction reviled by traditional C programmers. While `goto` is rarely used on larger platforms, in embedded system programming it can sometimes be used to save RAM.

If your hardware supports a **stack**, the stack contents and the space required to manage the stack are usually allocated from RAM. A stack is a structure which records the chronological ordering of information. It is used mainly in subroutine calls and interrupt servicing. A stack is a LIFO (last in, first out) structure. The following stack is from the Motorola MC68HC705C8. The stack is 64 bytes from address 00C0 to 00FF:

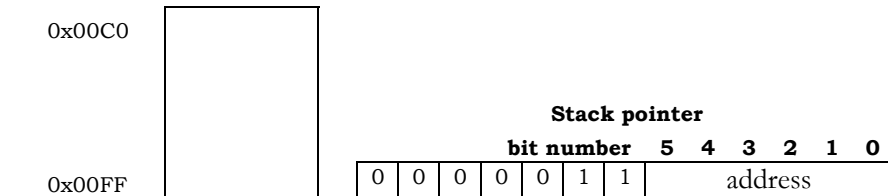


Figure 7: MC68HC705C8 stack

The stack pointer contains the address of the next free location on the stack. On reset the stack pointer for the MC68HC705C8 holds the value 00FF. The stack pointer is decremented when data is pushed on the stack and incremented when data is popped from the stack.

2.3.9 I/O Ports

There are two main port types, parallel and serial, and two port modes, synchronous and asynchronous. Parallel I/O requires a data line for each bit, while serial I/O uses a single line and transfers bits in sequence. Synchronous I/O is synchronised to a clock while asynchronous I/O is not. Microcontrollers most often have parallel I/O capability built in and serial I/O as a peripheral feature.

The following are some sample port configurations:

- The COP8SAA7 has four bidirectional 8 bit I/O ports called C, G, L and F where each bit can be either input, output or tristate. Each port has an associated configuration register and data register. It also has a MICROWIRE/PLUS synchronous serial interface
- The Motorola MC68HC705C8 has 3 8 bit ports called A, B, and C which can be either inputs or outputs depending on the value of the data direction register (DDR). There is also a 7 bit fixed input port called port D which is used for serial port programming. This device also has a SCI (serial communications interface) asynchronous serial interface and a SPI (serial peripheral interface) which both use Port D for their functions.
- The Microchip PIC16C74 has five ports: PORTA through PORTE. Each port has an associated TRIS register which controls the direction. PORTA uses the register ADCON1 to select analog or digital configuration. PORTD and PORTE can be configured as an 8 bit parallel slave port. The PIC16C74 has a SSP (synchronous serial port) module which can operate both in SPI and I²C modes. The device also has a SCI module

Serial ports have a frequency of operation called their **baud rate**. The baud rate is the reciprocal of the transmission time for each bit. For example, if the baud rate is 9600 bits/second then the transmission time for each bit is $\frac{1}{9600}$ of a second.

While microcontrollers do not support the same sophisticated input/output functions as larger platforms, such as those in the C `stdio` library, they still support device I/O. The input/output channels allow the microcontroller to communicate with such peripheral devices as timers, sensors, keypads and LCD screens.

Microcontroller ports are usually memory-mapped and can therefore be used like any other memory location. Ports usually consist of 8 or fewer bits which

often support **tristate** logic with three states: input, output or **high impedance**. High impedance mode is the state of being undefined or floating. Some devices only support binary logic and in those cases the bit can be defined as a combination of only two of the three states. If a port has programmable input and output it will also have an associated register which specifies whether the data is input or output. On many devices this register is called the DDR (data direction register).

To reserve memory-mapped port locations so your compiler does not use them for data memory allocation, you can use a `#pragma` preprocessor directive to specify the location of each mapped I/O register. This also allows you to provide a useful mnemonic name for each I/O port. You can then use the variable name associated with the port to read or write to a particular I/O port. The following defines two ports and their associated direction registers on the Motorola 68HC705C8:

```
#pragma portrw PORTA @ 0x0000;
#pragma portrw PORTB @ 0x0001;
#pragma portrw DDRA @ 0x0004;
#pragma portrw DDRB @ 0x0005;
```

Example 1: Defining ports with `#pragma` directives

It is then possible to write the value AC to the port using the C command:

```
DDRA=0xFF; //set the direction to output
PORTA=0xAC; //set the port to the value AC
```

2.3.10 Timer

A **timer** is a counter that is incremented at a fixed rate when the system clock pulses. There are several different types of timers available. A timer/counter can perform several different tasks. The CPU uses the timer to keep track of time accurately. The timer can generate a stream of pulses or a single pulse at different frequencies. It can be used to start and stop tasks at desired times.

A **COP** (computer operating properly) or **watchdog** timer checks for runaway code execution. The hardware implementation of watchdog timers varies considerably between different processors. In general watchdog timers must be turned on once within the first few cycles after reset and then reset periodically with software. Some watchdog timers can be programmed for different time-out delays. The reset sequence is sometimes as simple as a specialized instruction or as complex as sending a sequence of bytes to a port. Watchdog timers either reset the processor or execute an interrupt when they time out.

Timer configurations vary among microcontrollers. the following are some sample configurations:

- National Semiconductor's COP8SAA7 has a 16 bit timer called T1, a 16 bit idle timer called T0 and a watchdog timer. The idle timer T0 helps to maintain real time and low power during the IDLE mode. The timer T1 is used for real time controls tasks with three user-selectable modes.
- The Motorola MC68HC705C8 has a 16 bit counter and a COP watchdog timer.
- The Microchip PIC17C42a has four timer modules called TMR0, TMR1, TMR2 and TMR3, and a watchdog timer. TMR0 is a 16 bit timer with programmable prescaler, TMR1 and TMR2 are 8 bit timers and TMR3 is a 16 bit timer.

2.3.11 Interrupt Circuitry

An **interrupt** is an event that suspends regular program operation while the event is serviced by another program. Interrupts increase the response speed to external events. Different microcontrollers have different interrupt sources which can include external, timer and serial port interrupts. When an interrupt is received current operation is suspended, the interrupt is identified and the controller jumps (vectors) to an interrupt service routine.

There are two sources of interrupt: hardware and software. Hardware interrupts include a signal to a pin, timer overflow, and serial port interrupts. Software interrupts are commands given by the programmer, such as the SWI instruction for the Motorola MC68HC705C8.

There are two different interrupt types: **maskable** and **non-maskable**. A maskable interrupt can be disabled and enabled while non-maskable interrupts can not be disabled and are therefore always enabled.

Most 8 bit microcontrollers use vectored arbitration interrupts. Vectored arbitration means that when a specific interrupt occurs the interrupt handler automatically branches to an address associated with that interrupt.

The servicing of interrupts in general is dictated by the status of the **GIE** (Global Interrupt Enable). GIE is cleared when an interrupt occurs and all interrupts are delayed until it is set.

2.3.12 Buses

A bus carries information in the form of signals. There are three main buses: address, data, and control.

- 1) The **address bus** is unidirectional and carries the addresses of memory locations indicating where the data is stored. The number of wires in the address bus determines the total number of memory locations. With a 13 bit address bus, for example, there would be 2^{13} or 8192 memory locations.
- 2) The **data bus** is bi-directional and carries information between the CPU and memory or I/O devices. Computers are often classified according to the size of their data bus. The term “8-bit microcontroller” refers to a microcontroller with 8 lines on its data bus. The number of wires in the data bus determines the number of bits that can be stored in each memory location.
- 3) The **control bus** carries data which controls system activity. Often this data includes timing signals which synchronize the movement of other information.

2.4 Sample Microcontroller Configurations

The following are some sample microcontroller configurations.

2.4.1 Motorola MC68HC705C8

The MC68HC705C8 is a member of Motorola’s MC68HC05 family. It based on Von Neumann architecture.

Pins	40 or 44 pins
Clock	4MHz On-chip oscillator with crystal/ceramic resonator
RAM	176 bytes default (options include 208, 272 and 304)
ROM	7744 bytes default (options include 7696, 7648 and 7600)
Voltage	3.0 to 5.5 Volt
Registers	Accumulator, Index, Program Counter, Stack pointer, Condition Code Register
Timer(s)	COP, 16 bit programmable timer
Ports	4: 8 bit I/O ports PORTA, PORTB and PORTC, 7 bit input PORTD
Interrupts	5 interrupts: <u>IRQ</u> pin, SWI, SPI, SCI and timer
Serial	SPI (serial peripheral interface), SCI (serial communications interface)
Options	Clock monitor

Table 1: Hardware characteristics of the Motorola MC68HC705C8

2.4.2 National Semiconductor COP8SAA7

The COP8SAA7 is a member of National Semiconductor's COP8™ feature family. The COP8SAA7 is based on a modified Harvard architecture.

Pins	16, 20, 28, 40, 44 pin
Clock	Four user selectable clock options: 0.455 to 15 MHz
RAM	64 bytes
ROM	1024 bytes + 8 bytes User Storage space
Voltage	2.7 to 5.5 Volts
Registers	Accumulator, Program Counter, PSW, CNTRL, ICNTRL, stack pointer, X, B, S, SIOR, 2 timer registers
Timer(s)	Watchdog, idle timer, 16 bit timer
Ports	5: 8 bit bidirectional I/O Ports C, G, L and F, 8 bit output Port D
Interrupts	8 interrupts: timer1, timer0, portL wakeup, software trap, microwire/plus, external
Serial	MICROWIRE/PLUS (SPI compatible)
Options	Clock monitor

Table 2: Hardware characteristics of the National Semiconductor COP8SAA7

2.4.3 Microchip PIC16C54

The PIC16C54 is a member of the PIC16C5x family. These are 8 bit, EPROM-based CMOS microcontrollers. The PIC16C54 is Harvard architecture.

Pins	18 pins
Clock	20 MHz user selectable from low power crystal, crystal, high speed crystal, R/C
RAM	32 bytes
ROM	512 words (12 bits)
Voltage	2.5 V to 6.25 V
Registers	status, option, INDF, FSR, program counter, Working
Timer(s)	watchdog, 8 bit timer, reset timer
Ports	4 bit I/O Port A, 8 bit I/O Port B
Interrupts	1
Serial	none
Options	none

Table 3: Hardware characteristics of the Microchip PIC16C54

2.4.4 Microchip PIC16C74

The PIC16C74 is a member of the PIC16C7x family. These are 8-bit, EPROM-based CMOS microcontrollers. The PIC16C74 is Harvard architecture.

Pins	40/44
Clock	20 MHz
RAM	192
ROM	4K
Voltage	3 to 6
Registers	48 including Status, Option, Intcon, PIE1, PIR1, PIE2, PIR2, PCON, PCL, PCLATH, INDF, FSR
Timer(s)	2 8 bit, 16 bit, watchdog
Ports	6 bit PORTA, 8 bit PORTB, PORTC, parallel PORTD, 3 bit PORTE also TRISA, TRISB, TRISC, TRISD, and TRISE
Interrupts	12
Serial	SPI, I ² C, SSP, SCI
Options	A/D Converter

Table 4: Hardware characteristics of the Microchip PIC16C74

3. The Embedded Environment

Microcontrollers used in development projects have very limited resources. You are working close to your target machine and you must be familiar with your target hardware construction and operation.

A good quality C development environment incorporates tools which allow you to concentrate primarily on your applications and not on the hardware which runs them. However, you cannot ignore low-level details of your target hardware. The better you understand your run-time environment, the better you can take advantage of its limited capabilities and resources.

3.1 The Embedded Difference

There are many aspects of embedded systems development which must be considered. These are:

Reliability

Embedded systems must be reliable. Personal computer programs such as word processors and games do not need to achieve the same standard of reliability that a microcontroller application must. Errors in programs such as word processors may result in errors in a document or loss of data. An error in a microcontroller application such as a television remote control or compact disc player will result in a product that does not work and consequently does not sell. An error in a microcontroller application such as an antilock braking system or autopilot could be fatal.

Efficiency

Issues of efficiency must be considered in real time applications. A real time application is one in which must be able to act at a speed corresponding with the occurrence of an actual process.

Cost

Many embedded systems must compete in a consumer market and cost is an important issue in project development.

3.2 Fabrication Techniques

CMOS

Complementary Metal Oxide Semiconductor (CMOS) is a technique commonly used to fabricate microcontrollers. CMOS requires less power and CMOS chips can be static which allows the implementation of a sleep mode. CMOS microcontrollers must have all inputs connected to something.

PMP

Post Metal Programming (PMP) allows ROM to be programmed after final metalization. This allows ROM to be programmed very late in the productions cycle.

3.3 Memory Addressing and Types

Each microcontroller has a specific addressing range. An addressing range is the number of addresses a microcontroller can access. The addressing scheme used to access to these spaces varies from processor to processor, but the underlying hardware is similar.

3.3.1 RAM

Random access memory¹ or RAM consists of memory addresses the CPU can both read from and write to. RAM is used for data memory and allows the CPU to create and modify data as it executes the application program.

RAM is volatile, it holds its contents only as long as it has a constant power supply. If power to the chip is turned off, the contents of RAM are lost. This does not mean that RAM contents are lost during a chip reset. Vital state information or other data can be recorded in data memory and recovered after an interrupt or reset.

¹ *random access memory* is used because the CPU can access any block of memory in RAM in the same amount of time. This differs from sequential storage such as tape where access time differs for different parts of the storage space.

Some chips provide an alternate RAM power supply so that memory contents can be maintained even when the rest of the chip is without power. This does not make RAM any less volatile, without a backup power source the contents would still be lost. This type of RAM is called **battery backed-up static RAM**.

3.3.2 ROM

ROM, read only memory, is typically used for program instructions. The ROM in a microcontroller usually holds the final application program.

Maskable ROM is memory space that must be burned in by the manufacturer of the chip as it is constructed. To do this, you must provide the chip builder with the ROM contents you wish the chip to have. The manufacturer will then mask out appropriate ROM blocks and hardwire the information you have provided.

Since recording chip ROM contents is part of the manufacturing process, it is a costly one-time expense. If you intend to use a small number of parts, you may be better off using chips with PROM. If you intend to use a large number of parts for your application, then the one-time expense of placing your program in ROM is more feasible.

3.3.3 PROM

Programmable ROM, or PROM, started as an expensive means to prototype and test application code before burning ROM. In recent years PROM has gained popularity to the point where many developers consider it a superior alternative to burning ROM. As microcontroller applications become more specialised and complex, needs for maintenance and support rise. Many developers use PROM devices to provide software updates to customers without the cost of sending out new hardware.

There are many programmable ROM technologies available which all provide a similar service. A special technique is used to erase the contents of programmable ROM then a special method is used to program new instructions into the ROM. Often, the developer uses separate hardware to perform each of these steps.

3.3.4 EPROM

EPROM (erasable programmable ROM) is not volatile and is read only. Chips with EPROM have a quartz window on the chip. Direct exposure to ultra-violet

radiation will erase the EPROM contents. EPROM devices typically ship with a shutter to cover the quartz window and prevent ambient UV from affecting the memory. Often the shutter is a sticker placed on the window.

Developers use an EPROM eraser to erase memory contents efficiently. The eraser bombards the memory with high-intensity UV light. To reprogram the chip, an EPROM programmer is used, a device which writes instructions into EPROM.

The default, blank state for an EPROM device has each block of memory set. When you erase an EPROM you are really setting all memory blocks to 1. Reprogramming the device resets or clears the appropriate EPROM bits to 0.

Because of the way EPROM storage is erased, you can not selectively delete portions of EPROM – when you erase the memory you must clear the entire storage space.

3.3.5 EEPROM

EEPROM (electrically erasable programmable ROM) devices have a significant advantage over EPROM devices as they allow selective erasing of memory sections. EEPROM devices use high voltage to erase and re-program each memory block. Some devices require an external power source to provide the voltage necessary for erasing and writing and some have an onboard pump which the chip can use to build up a charge of the required voltage.

Developers can reprogram EEPROM devices while the chip is operating. However, EEPROM that can be rewritten is usually restricted to data memory storage. EEPROM storage used as program memory typically requires the use of an external power source and a programmer just like EPROM storage.

The most common use for EEPROM is recording and maintaining configuration data vital to the application. For example, many modems use EEPROM storage to record the current configuration settings. This makes the configuration available to the modem user after cycling the power on the modem. Often the default or factory configuration settings are stored in ROM and the user can issue a command to restore default settings by overwriting the current contents of EEPROM with the default information.

Sometimes chip manufacturers build EEPROM blocks into the chip for last-minute configuration options. This saves manufacturers money as they can design and fabricate a single chip and then set the EEPROM blocks to provide special purpose versions with specific capabilities. This method is often used to

produce microcontroller versions for use on an evaluation board where chip access to its own onboard ROM is turned off and replaced with external EPROM or EEPROM storage. This allows developers to test application code in cycles by downloading it to the board, programming the code into the EPROM or EEPROM, and debugging it as it executes in the target hardware.

3.3.6 Flash Memory

Flash memory is an economical compromise between EEPROM and EPROM technology. As with EEPROM high voltage is applied to erase and rewrite flash memory. However, unlike EEPROM, you can not selectively erase portions of flash memory – you must erase the entire block as with EPROM devices. Many manufacturers are turning to flash memory. It has the advantages of not requiring special hardware and being inexpensive enough to use in quantity.

Manufacturers often provide customers with microcontroller products whose ROM is loaded with a boot or configuration kernel where the application code is written into flash memory. When the manufacturer wants to provide the customer with added functionality or a maintenance update, the hardware can be reprogrammed on site without installing new physical parts. The hardware is placed into configuration mode which hands control to the kernel written in ROM. This kernel then handles the software steps needed to erase and re-write the contents of the flash memory.

Another useful implementation of flash memory includes a device which can connect electronically to a computer owned by the manufacturer. The configuration kernel connects to the manufacturer's computer, downloads the latest version of the control application and writes this application to flash memory. Such elaborate applications are typically beyond the resources of an 8 bit microcontroller; we mention the example to show the advantage of programmable ROM technologies.

3.3.7 Registers

The CPU maintains a set of registers which it uses to store information. Registers are used to control program execution and maintain intermediate values needed to perform required calculations. Some microcontrollers provide access to CPU registers for temporary storage purposes. This can be *extremely* dangerous as the CPU can at any time overwrite a register being used for its designated purpose.

8 bit microcontrollers do not often provide resources for register memory outside the CPU. This means that the `C register` keyword is meaningless because the compiler can not dedicate a CPU register for data storage.

Some C implementations will set aside RAM for special purpose *pseudo-registers* to use when your application attempts certain operations. For example, if you attempt a 16 bit math operation, the compiler can dedicate a portion of base-page RAM for 16 bit pseudo-registers which store values during math operations. You can use these special registers for temporary purposes in places where your code will not require them for their intended purpose. You must be careful, if the compiler uses a pseudo-register it will overwrite current contents.

3.3.8 Scratch Pad

Microcontrollers are typically very short on resources, especially data memory space. Many C compilers use some available RAM for internal purposes such as pseudo-registers. An efficient C compiler will support *scratch pads* in data memory. A scratch pad is a block of memory which can be used for more than one purpose.

The simplest way to conserve data memory is through the judicious use of global variables. For example, in a traditional C environment developers create local counter variables every time they are required because data memory is cheap and plentiful. However, embedded systems developers will often create global counter variables. Any function can then use this allocated block of data memory when a counter or temporary variable is needed. Examine the following union called `ScratchPad` which is declared globally:



See 11.6
Unions

```
union {
    int asInt;
    char asChar;
    short asShort;
    long asLong;
    void near * asNPtr;
    void far * asFPtr;
    struct {
        short loByte;
        short hiByte;
    } asWord;
} ScratchPad;
```

Example 2: Using a union structure to create a scratch pad

To use the global variable as a loop counter within a function, the following code could be used:

```
int somefunc() {
    ScratchPad.asShort=0;
    while (ScratchPad.asShort < 10) {
        // some code
        ScratchPad.asShort += 1;
    } // end while
    return (someIntValue);
}
```

Example 3: Using globally allocated data space in a function

Some C compilers support a C extension which fixes the location of a symbol in memory. In these cases, the compiler typically does not check that the memory specified is not being used by other data. You can use this feature to manage how variables are placed in data memory space. More importantly you can overlay one variable symbol on top of the memory allocated for another. This is a useful technique for reusing allocated variable space.

For example, it is possible to reuse internal the pseudo-register variables created by the compiler in portions of your code that do not use them for their designated purpose. For example, if your compiler creates the 16 bit pseudo index register `__longIX` you can reuse this 16 bit location with the following statement²:

```
long int myTemp @ __longIX;
```

You must ensure that you understand exactly how and when the compiler uses these internal variables before you reuse the variable space.

Fixing a symbol at a specific memory location will likely affect the optimization a compiler will perform with the symbol. It may be more worthwhile to avoid this method of overlaying memory in favour of the savings generated by the compiler's optimizer.

3.4 Interrupts

Interrupts allow the microcontroller to interact with its environment. If your microcontroller does not have interrupts you must poll peripherals to

² The @ symbol uses the address allocated to `__longIX` for the new symbol `myTemp`. This is not standard C so the syntax your compiler provides may be different.

determine if they require servicing. It is much more efficient to have peripheral devices inform, or interrupt, the controller when they require servicing.

An interrupt is a signal sent to the microcontroller which causes it to stop its current execution and perform another action. The chip stops executing your main program and executes some other code. Interrupts can be edge triggered (rising or falling) or level triggered.

3.4.1 Interrupt Handling

Code executed by an interrupt is not generally considered part of the main application. Since this code handles the cases where an interrupt occurs, it is called an interrupt handler or an **interrupt service routine**.

NOTE

It is vital that you understand how *your* target hardware implements interrupts as this affects both the service routines you must write and how you write them.

In general, you must write an interrupt service routine for each interrupt your target hardware can detect even if the handler consists solely of a return from interrupt or a similar instruction.

3.4.2 Synchronous and Asynchronous Interrupt Acknowledgement

Interrupts are asynchronous: they are events that can occur during, after, or before an instruction cycle. Interrupt acknowledgement can be either *synchronous* or *asynchronous*. Most interrupt acknowledgement is synchronous, the instruction currently being executed is completed before the interrupt is acknowledged.

Theoretically, when the processor acknowledges an interrupt asynchronously it halts execution of the *current* instruction and immediately services the interrupt. The only asynchronously acknowledged interrupt is RESET. Since RESET erases the state of the machine, it is a moot point whether the CPU actually halts execution of the current instruction or not.

When the processor acknowledges an interrupt synchronously, it finishes executing the current instruction and, before it performs a fetch for the *next* instruction, it services the interrupt.

3.4.3 Servicing Interrupts

There are two general ways in which microcontrollers service interrupts, each with several variations.

① Vectored Arbitration System

Some machines reserve a portion of program memory for interrupt vectors. The location of each particular vector in program memory may vary from processor to processor but it cannot be changed by the programmer. The programmer can only change the data at each vector location.

Each interrupt vector contains the address of that interrupt's service routine. When the compiler allocates program memory for interrupt handlers, it places the appropriate address for the handler in the appropriate interrupt vector. To help the compiler you must usually tell it *where* the interrupt vector for each interrupt is located in program memory.

When an interrupt occurs, global interrupts are first disabled to prevent an interrupt service from being itself interrupted. On the COP8SAA7 this involves setting the GIE bit to zero. The machine then reads the address contained at the appropriate interrupt vector. It then jumps to the address and begins executing the interrupt service code. Vectored interrupts are much faster than non-vectored.

The following are sample interrupts from the National Semiconductor COP8SAA7:

Rank	Source	Description	Vector Address *
1	Software	INTR Instruction	0bFE - 0bFF
2	Reserved	Future	0bFC - 0bFD
3	External	G0	0bFA - 0bFB
4	Timer T0	Underflow	0bF8 - 0bF9
5	Timer T1	T1A/Underflow	0bF6 - 0bF7
6	Timer T1	T1B	0bF4 - 0bF5
7	MICROWIRE/PLUS	BUSY Low	0bF2 - 0bF3
8	Reserved	Future	0bF0 - 0bF1
9	Reserved	Future	0bEE - 0bEF
10	Reserved	Future	0bEC - 0bED
11	Reserved	Future	0bEA - 0bEB
12	Reserved	Future	0bE8 - 0bE9
13	Reserved	Future	0bE6 - 0bE7
14	Reserved	Future	0bE4 - 0bE5
15	Port L/Wakeup	Port L Edge	0bE2 - 0bE3
16	Default	VIS Instruction Execution without any interrupts	0bE0 - 0bE1

* b represents the Vector to Interrupt Service routine (VIS) block. VIS and the vector table must be within the same 256 byte block. If VIS is the last address of a block the table must be in the next block.

Table 5: Sample vectored interrupts

② Non-Vectored Priority System

When an interrupt occurs, the PC branches to a specific address. At this address the interrupts must be checked sequentially to determine which one has caused the interrupt.

This scheme can be very slow and there can be a large delay between the time the interrupt occurs and the time it is serviced. However, the programmer can set the interrupt priority and non-vectored interrupts are feasible for microcontrollers with less than five interrupts.

3.4.4 Interrupt Detection

On most chips, the interrupt process saves the *state* of the machine including the current program counter, stack pointer, and register contents. This is done to ensure that after an interrupt is serviced execution will resume at the appropriate point in main program with no loss of data.

Some chips save the machine state automatically while others will only save a portion of the machine state. In the second case it is up to the programmer to provide code which saves the current state. Usually each interrupt handler will do this before attempting anything else. The location and accessibility of the saved state information varies from machine to machine. In most cases, it is saved on a *stack* located in data memory.

For example the Motorola MC68HC705C8 saves the machine state in the stack as follows:

1	1	1	Condition Codes
Accumulator			
Index Register			
0	0	0	PC High
Program Counter Low			

Figure 8: Saving the machine state on the MC68HC705C8

Many C compilers for embedded microcontrollers reserve a portion of data memory for internal uses such as for pseudo-registers. You must check your compiler documentation to determine what code you must write to preserve the information located in these memory blocks. Some compilers document their internal data memory overhead so you can determine what you must preserve in your interrupt handlers while others automatically generate code to preserve this data.

One way to conserve memory is to avoid unnecessarily preserving data. If your compiler creates a pseudo register for 16 bit math operations and your interrupt handler does not use this pseudo register, then you need not preserve its state.

3.4.5 Executing Interrupt Handlers

To minimize the possibility of an interrupt handler being itself interrupted, the microcontroller will usually disable interrupts while executing an interrupt handler. The method of doing this varies from chip to chip. Some platforms automatically disable interrupts, while others leave this to the programmer. Masking interrupts is useful during timing critical sections of code. The COP8SAA7, for example, has a GIE (Global Interrupt Enable) which is set to allow interrupts or cleared to prevent interrupts. On the Motorola

MC68HC705C8 the interrupt mask bit of the Condition Code Register is set to prevent interrupts.

Some machines provide a small number of *non-maskable* interrupts (NMI). Interrupts that can be disabled are *maskable*, those which you cannot disable are *non-maskable*. For example, RESET is non-maskable – regardless of the code currently executing the CPU must service a RESET interrupt. Some microcontrollers also designate software interrupts or BREAK instructions that you can use as a non-maskable interrupt.

3.4.6 Multiple Interrupts

What happens after the CPU services an interrupt? This varies depending upon target hardware. In general, the CPU first checks for any outstanding interrupts.

On some machines the CPU first fetches an instruction and then checks for interrupts after executing this instruction. This guarantees that no matter how many interrupts cue up, the machine will always step through program code and no more than one interrupt handler will execute between each main program instruction.

On most machines the CPU will check for interrupts before performing the next instruction fetch. As long as it detects a pending interrupt it will service the interrupt before fetching the next instruction. This means it is possible to halt a program by continuously sending interrupts. On the other hand, it guarantees that an interrupt is serviced before any more main program code is executed.

When an interrupt occurs the signal sets a register bit. When the CPU checks for pending interrupts it reads the register for set bits. Upon completing an interrupt handler, the appropriate bit in the register is cleared.

How does the CPU decide which interrupt to service first? Each interrupt a chip can detect has a precedence, the chip services those interrupts with a higher precedence first.

3.5 Specific Interrupts

Microcontrollers vary widely in the types of interrupts they can detect. Some general types are widely available in one form or another. The only universal interrupt is RESET and some simple chips support no other interrupts.

3.5.1 RESET

The RESET interrupt prompts the chip to behave as if the power has been cycled. It does not *actually* cycle the power to the chip. This means that the contents of volatile memory, typically data memory, can remain intact. The **reset vector** contains the address of the first instruction that will be executed by the CPU.

You can write an initialization routine to be executed before any other program code which first checks specific locations in data memory for particular values and then loads values into those locations. This can be used to check if the RESET was cold, power cycled, or warm, power not cycled. Some compilers support an initialization function which is executed upon RESET before the main program.

On most chips, RESET causes the CPU to halt execution immediately and restart itself. On some chips, RESET may finish the current instruction. Each microcontroller performs a series of actions when it detects a RESET. For example, when a RESET occurs on the Motorola MC65HC705C8 the following actions occur:

- 1) Data direction registers are cleared
- 2) Stack pointer is set to 0x00FF
- 3) CCR I bit is set
- 4) External interrupt latch is cleared
- 5) STOP latch is cleared
- 6) WAIT latch is cleared

A RESET can occur because of a manual reset, a COP time out, low voltage, initial power on, or an attempt to execute an instruction from an illegal address.

3.5.2 Software Interrupt/Trap

Some chips that support interrupts provide an instruction in the instruction set which the programmer can use to halt program execution. This instruction name is different for different devices.

The COP8SAA7 has a Software trap which occurs when the INTR instruction is placed in the instruction register. The software trap is used for unusual and

unknown errors. The Motorola MC68HC705C8 has a software interrupt executable instruction called SWI.

3.5.3 IRQ

IRQ interrupts are physical pins or ports on the chip which generate an interrupt when they are sent a signal. Some chips do not support IRQ type interrupts and those that do implement them in many different ways. The number of pins available for IRQs varies widely from chip to chip.

The developer usually has the ability to configure the IRQ interrupts to detect signals in specific ways. For example, they can be made sensitive to a signal edge, a signal hold, or a signal fall.

For example, the Microchip PIC17C42a has an INT external interrupt pin. The developer can set the interrupt trigger to be either the rising edge or falling edge by setting an appropriate register bit. The INT interrupt can be disabled by clearing the appropriate control bit.

3.5.4 TIMER

A TIMER interrupt occurs when a timer overflow is detected. For example, In the Microchip PIC16C74 there is a TMR0 interrupt which is generated when the TMR0 8 bit timer overflows. An overflow occurs when the timer goes from 1111 1111 to 0000 0000. The timer is usually incremented every instruction cycle.

TIMER interrupts in general provide access to an external clock. This is useful in applications where timing is critical. For control applications, for example, it is important to sample input data at specific time intervals. This is usually accomplished with TIMER based interrupts.

You can also use TIMER interrupts in other ways, depending upon your hardware capabilities. Some chips, such as the Microchip PIC16C74, have readable and writable timers which let you specify a certain duration of time. Each instruction cycle of the CPU counts from this time and when the counter overflows the TIMER interrupt fires. Other chips let you specify the number of cycles as an interval for the TIMER and it will fire every time the specified number of cycles pass.

The TIMER interrupt is most useful in building a *watchdog* or *computer operating properly* timer for devices which do not include one. First you configure the watchdog to tell it how long it can last without attention. Then, you provide

code in your program to touch the watchdog at regular intervals before the time period expires. If your program leaves the watchdog too long without attention, the configured time period passes with no touch instruction, the watchdog activates the RESET interrupt.

This type of timer interrupt provides your program with an independent safety net. Since the watchdog timer depends only upon the clock signals to do its job, if your program ever fails the watchdog will realize that the computer is not operating properly and will activate a RESET.

3.6 Power

Most microcontrollers support 4.5 to 5.5 Volt operation. There are also many low voltage parts which are designed to work at 3 volts or less.

3.6.1 Brownout

Microcontrollers have an on-board circuit which provides brownout protection. A brownout occurs when the operating voltage falls below the defined brownout voltage. When a brownout occurs the device is reset and waits for the operating voltage to rise above the brownout voltage.

3.6.2 Halt/Idle

Individual microcontrollers have specific modes which stop the execution of the program without affecting the power to the microcontroller. In these modes less power is required and power consumption is reduced. Halt mode stops all activities and can be terminated by a reset or an interrupt. Idle mode stops most activities. The clock, watchdog, and idle timer remain active.

3.7 Input and Output

Input and output are lines or devices which carry information between the microcontroller and the outside world.

3.7.1 Ports

A port is a physical input/output connection. Most ports on 8 bit microcontrollers are 8 bits or less. Ports can be either input, output or

input/output. Often the port state is set with a direction register which determines if the port is input, output or input/output. When a port pin is an output it is a latched output. This means that when the pin is in a given state, set or unset, it will remain in that state until reset.

Microcontrollers usually contain several ports.

For example, the Microchip PIC16C74 has five ports called PORTA, PORTB, PORTC, PORTD and PORTE. PORTA is 6 bit latch which is configured as input or output using the register TRISA. PORTA can also be configured as analog or digital using the ADCON1 register. PORTB is an 8 bit bi-directional port with data direction register TRISB.

The National Semiconductor COPSAA7 contains four bi-directional ports: PORTC, PORTG, PORTL and PORTF. Each bit can be configured as input, output or tristate.

3.7.2 Serial Input and Output

CAN

Controlled Area Network was developed by Bosh and Intel. It is a multiplexed wiring scheme.

I²C™ (Inter-Integrated Circuit bus)

A two wire serial interface developed by Phillips. It is a multi-master, multi-slave network interface with collision detection. Up to 128 devices can exist on the network. The two lines consist of the serial data line and the serial clock line which are both bidirectional..

It provides a communication link between integrated circuits. Every component hooked up to the bus has its own unique address.

J1850

J1850 is the SAE (Society of Automotive Engineers) standard.

MICROWIRE PLUS (National Semiconductor)

A serial synchronous bi-directional communications interface used on National Semiconductor devices. It is SPI compatible. It consists of an 8 bit serial shift register with serial data input serial data output and serial shift clock

SCI (Serial Communications Interface)

A Serial Communications Interface is an asynchronous serial interface. It is an enhanced UART. The SCI has a transmitter and a receiver which are functionally independent but use the same data format and baud rate.

SCI features standard nonreturn to zero format, error detection, simultaneous send and receive, 32 different baud rates, selectable word length, and four separate interrupt conditions. There are five registers: SCDAT (serial communication data register), SCCR1 (serial communication control register 1), SCCR2 (serial communication control register 2), SCSR (serial communication status register), and the baud rate register.

SPI (Serial Peripheral Interface)

A Serial Peripheral Interface is a three-wire synchronous serial port which allows several microcontrollers to be interconnected. In the configuration there must be at least one microcontroller master while the remaining microcontrollers can either be masters or slaves.

SPI features four programmable master bit rates, programmable clock polarity and phase and end of transmission interrupt. The clock is not included in the data stream and must be provided as a separate signal. There are three registers, SPSR, SPCR and SPDR that allow for control, status and storage functions.

There are four basic pins which have different names on different devices:

- Data out
- Data in
- SCK (Serial Clock)
- $\overline{SS}$ (Slave Select)

A SPI is a type of SSP.

SSP (Synchronous Serial Port)

The SSP does not require start and stop bits and operates at higher clock rates than asynchronous serial ports.

UART

A Universal Asynchronous Receiver Transmitter is a serial port adapter that receives and transmits serial data with each data character preceded by a start bit

and followed by a stop bit. There is sometimes a parity bit included. A UART is used mainly as a serial to parallel and parallel to serial converter.

USART

A Universal Synchronous/Asynchronous Receiver Transmitter is a serial port adapter used for synchronous or asynchronous serial communication.

3.8 Analog to Digital Conversion

It is often necessary to convert an external analog signal to a digital representation or to convert a digital signal to an analog signal.

Successive Approximation Converter

Most microcontrollers use a successive approximation A/D converter. The converter works with one bit at a time from the MSB (most-significant bit) and determines if the next step is higher or lower. This technique is slow and consumes a great deal of power. It is also cheap and has consistent conversion times.

The Microchip PIC16C74 has an A/D converter module which features 8 analog inputs. These 8 inputs are multiplexed into one sample-and-hold which is the input into the converter.

Single Slope Converter

Appears in National Semiconductor's COP888EK. It includes an analog MUX/comparator/timer with input capture and constant current source. The conversion time varies greatly and is quite slow. It also has 14 to 16 bit accuracy.

Flash converter

Examines each level and decides what level the voltage is at. It is very fast, but draws a great deal of current and is not feasible beyond 10 bits.

3.9 Miscellaneous

3.9.1 Digital Signal Processor

A Digital Signal Processor (DSP) runs repetitive, math intensive algorithms.

3.9.2 Clock Monitor

The clock monitor watches the clock and determines if it is running too slow. It can activate a microcontroller reset.

3.10 Devices

3.10.1 Mask ROM

ROM whose contents are set by masking during the manufacturing process.

3.10.2 Windowed Parts

A microcontroller with a window which allows for ROM contents to be erased.

3.10.3 OTP

OTP (One Time Programmable) devices are microcontrollers where once a program is written into the device it cannot be erased.

4. Programming Fundamentals

It is necessary to understand some basic computer programming concepts before learning C programming.

4.1 What is a Program?

The most important thing to remember about computers is that they can do only what they are instructed to do. To accomplish a meaningful task on a computer, someone must give it exhaustive and very explicit instructions. A collection of such instructions is called a program and the person who writes and revises these instructions is known as a programmer or developer.

4.2 Number Systems

There are several different number systems. We are used to the decimal number system which is of base 10. This means that it has ten digits and coefficients are multiplied by powers of 10. For example 456 is the same as $4(10^2) + 5(10^1) + 6(10^0) = 400 + 50 + 6 = 456$.

Computers use the binary number system with base 2: it has two digits (0 and 1) and the coefficients are multiplied by powers of 2. For example 110 is the same as $1(2^2) + 1(2^1) + 0(2^0) = 6$.

The hexadecimal number system is often used as it is easier to read than binary numbers. It is base 16 and uses 0-9 and A-F to represent values.

Base 10 Decimal	Base 2 Binary	Base 16 Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
100	0110 0100	64
255	1111 1111	FF
1024	0100 0000 0000	400
65,535	1111 1111 1111 1111	FFFF

Table 6: Binary, decimal and hexadecimal

4.3 Binary Information

When people read and write they use extremely powerful and flexible coding systems called alphabets. Computers, however, can only handle information written in the most simple coding system possible — binary notation. The binary alphabet has only two components: 1 and 0.

A computer's memory consists of a long series of switches known as **bits**. These switches can exist in only two states; therefore, they are well suited to the binary alphabet. At any given time a single bit in computer memory can represent either 1 or 0. A bit containing 1 is referred to as being set while a bit containing a 0 is referred to as being unset or clear. Anything that a computer reads, writes, or executes must be encoded as a series of set and unset bits.

The following diagram shows the relationship between data value and data storage:

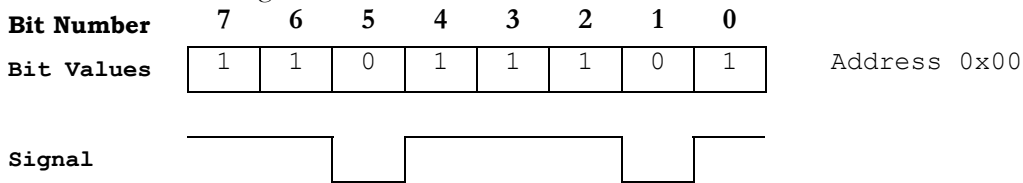


Figure 9: Data storage VS. data value

In the example the data is stored at the address 00 Hex. The data stored at that location has the value 1101 1101 binary or DD hexadecimal.

One bit in computer memory can record either 0 or 1 because it contains a single binary digit which can exist in only two states. Two bits read in sequence, however, can record four possible numbers: 0, 1, 2, and 3 because two bits can exist in the states 00, 01, 10 and 11. As with decimal notation, the first digit records the multiples of one included in the number. The second digit records the multiples of two since the computer only has two digits available. Adding a third digit allows for the encoding of multiples of 4.

Bits are often grouped in sets. 8 bits make 1 byte, while 16 bits make one word. Standard terminology refers to 2^{10} (1024) bytes as a kilobyte.

A programmer can give the computer information and instructions using long strings of 1's and 0's. However, this process would be very time consuming and prone to error. To resolve these problems programmers have developed languages in which to write programs. Languages help the programmer by making the job of programming a computer faster, more efficient, and more reliable.

4.4 Memory Addressing

Computer memory is divided up into addresses. Each address holds an 8 bit (or 1 byte) value. The number of address lines determines the number of locations available. For example, the MC68HC05C8 can address 8192 bytes of memory. Since each bit can hold one of two values (0 or 1) and $2^{13} = 8192$ we know that there are 13 address lines. The first address will be the value 0 0000 0000 0000 (0x0000) and the last address will be the value 1 1111 1111 1111 (0x1FFF).

Microcontrollers have different addressing modes which allow them to access locations in memory as quickly as possible. For example, the first 256 locations on the Motorola MC68HC705C8 can be accessed using direct addressing mode where the CPU assumes that the high byte of the address is 00000000.

4.5 Machine Language

Computers only understand one language: machine language. Each family of computers has its own machine language which can not be understood by any other family of computers. Any particular computer within a family may also use a slightly different dialect of the family language and may incorporate features not available to other members of the family. Any instructions for a specific computer must be given in its individual machine language. Machine language is a collection of binary numbers such as:

```
00000000000001010101011110100110100010101010011010001111101
111000000100000000000111111110001110101011000000000000010
10
```

The hexadecimal equivalent is:

```
000557A68AA68FBC0800FF8EAC000A
```

4.6 Assembly Language

Each microcontroller has its own assembly language or assembly language variation. Assembly language consists of mnemonic instructions and addressing modes where the instruction describes what to do and the addressing mode describes where to do it. The following instructions are from the National Semiconductor COP888 assembly language:

Address	Instruction	Hex	Explanation
0005	LD B,#034	9F 34	Load B register with 34
0007	SBIT 06,[B]	7E	Set bit 6 of B register to 1
000E	IFBIT 06,[B]	76	If bit 6 of B register is 1 then execute next instruction
0011	LD A,[B]	AE	Load Accumulator with contents of location referenced by B
0012	IFNE A,#001	99 01	If the Accumulator is not equal to 1 then execute next instruction
0014	JP 00017	02	Jump to PC + 02 + 1 (0017)
0015	CLRA	64	Clear accumulator
0000	JMPL 00005	AC 00 05	Jump to address 0005

Table 7: Interpretation of assembly language

8 bit microcontrollers usually use byte size instruction codes. Each instruction has two possible components: an opcode and an operand. The opcode is the function that the instruction performs while the operand is data used by the opcode. Neither opcodes nor operands are restricted to 1 byte.

There are several different types of addressing modes. An addressing mode is simply the means by which an address is determined. Some common modes are immediate data, direct address, and indirect or indexed address.

4.6.1 Assembler

Assembly language programs are not directly executable, they must be translated to machine language. This translation is done using a program called an assembler.

4.7 Instruction Sets

Most microcontrollers use the **CISC** (Complex Instruction Set Computer) foundation. CISC is an architecture which handles complex instructions. If one complex instruction encapsulates several simple instructions, the time spent retrieving the instruction from memory was reduced. This is useful with sequential computing designs.

Some microcontrollers are based upon a **RISC** (Reduced Instruction Set Computer) design. RISC is an architecture which handles simple instructions.

The processor can execute these instructions at a very high speed. RISC uses a technique called pipelining the processing of instructions can be overlapped. For example, one instruction can be read from memory while another is decoded and another is executed. Many RISC machines have a single instruction size and a small number of addressing modes.

Some microcontrollers are called **SISC** (Specific Instruction Set Computer) machines. This is based on the fact that the instruction sets are designed specifically for control purposes.

	Instructions	Instruction Size	Cycle	Address Modes
MC68HC705C8	58	8, 16 or 24 bit	2 to 11	10
COP8SAA7	56	8, 16 or 24 bit	1 to 5	10
PIC16C54	33	12 bit	1 or 2	3
PIC16C74	35	14 bit	1 or 2	3

Table 8: Instruction set comparisons

4.8 The Development of Programming Languages

Programming languages were originally developed to reduce program development time. Programming languages also increase the portability, readability and modifiability of programs. For example a program written for the National Semiconductor COP8SAA7 in its assembly language will not run on the Motorola 68HC705C8, actually it may not run on other COP8 parts because of differences in the instruction set. If a program is written in C for the COP8SAA7 it can be ported to the 68HC705C8 with few changes.

The following examples show the same C code compiled for the COP8SAA and the 68HC705C8. When a language such as C is used the program must simply be recompiled while an assembly language program must be completely rewritten.

```

0034 0006                                bit b@0x34.6;
0008                                char j;
0005                                #define bit5 5
0008 0005                                bit bj@&j.bit5;
0008 0005                                bit bj1@j.bit5;
                                void main(void) {
0005 9F 34                                LD      B, #034
0007 7E                                SBIT    06, [B]
0008 BD 08 6D                            RBIT    05, 008
                                b=1;
                                bj=0;

```

```

000B BD 08 7D SBIT 05,008          bjl=1;
000E 76       IFBIT 06,[B]         if (b==1)
000F 6E       RBIT 06,[B]         b=0;
000F 6E       RBIT 06,[B]
0010 57       LD B,#08            b=(j==1)?0:1;
0011 AE       LD A,[B]
0012 99 01    IFNE A,#001
0014 02       JP 00017
0015 64       CLRA
0016 02       JP 00019
0017 98 01    LD A,#001
0019 9F 34    LD B,#034
001B 92 00    IFEQ A,#000
001D 02       JP 00020
001E 7E       SBIT 06,[B]
001F 01       JP 00021
0020 6E       RBIT 06,[B]
0021 8E       RET                  }

```

Example 4: A typical assembly language program for the COP8SAA

```

0034 0006          bit b@0x34.6;
0050              char j;
0005              #define bit5 5
0050 0005          bit bj@&j.bit5;
0050 0005          bit bjl@j.bit5;
                                void main(void){
0200 1C 34          BSET 6,$34          b=1;
0202 1B 50          BCLR 5,$50          bj=0;
0204 1A 50          BSET 5,$50          bjl=1;
0206 0D 34 02      BRCLR 6,$34,$020B    if (b==1)
0209 1D 34          BCLR 6,$34          b=0;
020B B6 50          LDA $50            b=(j==1)?0:1
020D A1 01          CMP #$01
020F 26 03          BNE $0214
0211 4F            CLRA
0212 20 02          BRA $0216
0214 A6 01          LDA #$01
0216 4D            TSTA
0217 26 04          BNE $021D
0219 1D 34          BCLR 6,$34
021B 20 02          BRA $021F
021D 1C 34          BSET 6,$34
021F 81            RTS                  }

```

Example 5: Program in Example 4 compiled for the 68HC705C8

One of the most important tools that programmers developed to deal with new high level languages is the language compiler.

4.9 Compilers

Compilers translate high level programming language instructions into machine language. They perform the same task for high level languages that an assembler performs for assembly language, translating program instructions from a language such as C to an equivalent set of instructions in machine language. This translation does not happen in a single step – three different components are responsible for changing C instructions into their machine language equivalents. These three components are:

- 1) Preprocessor
- 2) Compiler
- 3) Linker

4.9.1 The Preprocessor

A program first passes through the C **preprocessor**. The preprocessor goes through a program and prepares it to be read by the compiler. The preprocessor includes the contents of other programmer specified files, manipulates the program text, and passes on instructions about the particular computer for which the compiler will be translating.

4.9.2 The Compiler

The **compiler** translates a program into an intermediate form containing both machine code and information about the program's contents. The compiler is the second component to handle your program. The compiler has the most important job: digesting and translating the program into a language readable by the destination computer.

Many compilers operate in different passes through the code. There are often passes specifically to handle optimizations of code which will reduce the size of the machine code generated.

4.9.3 The Linker

When programs were written in the past often the development computer was not powerful enough to hold the entire program being developed in memory at one time. Historically, programs had to be divided into separate modules where each module would be compiled into object code and a **linker** would link the

object modules together. Our development machines today are very powerful and the use of a linker is no longer absolutely necessary.

Many implementations of C provide function libraries which have been pre-compiled for a particular computer. These functions serve common program needs such as serial port support, input/output, and description of the destination computer. Functions within libraries are usually either linked with modules which use them or included directly by the compiler if the compiler supports library function inclusion.

When your program has been pre-processed, compiled and linked, the destination computer will be able to read and execute your program.

4.10 Cross Development

A cross compiler runs on one type of computer and produces machine code for a different type of computer. While many 8 bit embedded microcontrollers can support sophisticated and extremely useful programs, they are not powerful enough to support the resource needs of a C development environment. How does a developer create and compile programs for an 8 bit microcontroller? By using a cross compiler.

4.10.1 Cross compiler

An embedded systems developer writes and compiles programs on a larger computer which can support a C development environment. The compiler used does *not* translate to the machine language of the development computer, it produces a version of the program in the machine language of the 8 bit microcontroller. A compiler that runs on one type of computer and provides a translation for a different type of computer is called a **cross-platform compiler** or **cross-compiler**.

The object code formats generated by a cross-compiler are based on the target device. For example, a compiler for the Motorola MC68HC705C8 could generate an S-record file for its object code.

4.10.2 Cross development tools

After a program is compiled it must be tested using a **simulator** or an **emulator**. After testing the developer uses a special machine called a

programmer to imprint the translated program into the memory of the 8 bit microcontroller.

Simulator

A simulator is a software program which allows a developer to run a program designed for one type of machine (the target machine) on another (the development machine). The simulator simulates the running conditions of the target machine on the development machine.

Using a simulator you can step through your code while the program is running. You can change parts of your code in order to experiment with different solutions to a programming problem. Simulators do not support real interrupts or devices.

An **in-circuit simulator** includes a hardware device which can be connected to your development system to behave like the target microcontroller. The in-circuit simulator has all the functionality of the software simulator while also supporting the emulation of the microcontroller's I/O capabilities.

Emulator

An emulator or in-circuit emulator is a hardware device which behaves like a target machine. It is often called a real time tool because it can react to events as the target microcontroller would. Emulators are often packaged with monitor programs which allow developers to examine registers and memory locations and set breakpoints.

4.10.3 Embedded Development Cycle

The development process for embedded software follows a cycle:

1. Problem specification
2. Tool/chip selection
3. Software plan
4. Device plan
5. Code/debug
6. Test
7. Integrate

Problem Specification

The problem specification is a statement of the problem that your program will solve without considering any possible solutions. The main consideration is explaining in detail what the program will do.

Once the specification of the problem is complete you must examine the system as a whole. At this point you will consider specific needs such as those of interrupt driven or timing-critical systems.

For example, if the problem is to design a home thermostat the problem specification should examine the functions needed for the thermostat. These function may include reading the temperature, displaying the temperature, turning on the heater, turning on the air conditioner, reading the time, and displaying the time. Based on these functions it is apparent that the thermostat will require hardware to sense temperature, a keypad, and a display.

Tool/Chip Selection

The type of application will often determine the device chosen. Needs based on memory size, speed and special feature availability will determine which device will be most appropriate. Issues such as cost and availability should also be investigated.

The tools available will also impact a decision to develop with a certain device. It is essential to determine if the development decisions you have made are possible with the device you are considering. For example, if you wish to use C you must select a device for which there is a C language compiler. It is also useful to investigate the availability of emulators, simulators and debuggers.

Software Plan

The first step in the software plan is to select an algorithm which solves the problem specified in your problem specification. Various algorithms should be considered and compared in terms of code size, speed, difficulty, and ease of maintenance.

Once a basic algorithm is chosen the overall problem should be broken down into smaller problems. The home thermostat project quite naturally breaks down into modules for each device and then each function of that device.

For example, the thermostat may have a display to the LCD display module and a read from the keyboard module.

Device Plan

The routines for hardware specific features should also be planned. These routines include:

- 1) Set up the reset vector
- 2) Set up the interrupt vectors
- 3) Watch the stack (hardware or software)
- 4) Interact with peripherals such as timers, serial ports, and A/D converters.
- 5) Work with I/O ports

Code/Debug

The modules from the software plan stage are coded in the project language. The coded modules are compiled or assembled and all syntactic error are repaired.

Debugging should consider issues such as:

- Syntactic correctness of the code
- Timing of the program

Test

Each module should be tested to ensure that it is functioning properly. This testing is done using simulators and/or emulators. It is also important to test the hardware you will be using. This is easily done by writing small programs which test the devices.

Integrate

The modules must be combined to create a functioning program. At this point is important to test routines which are designed to respond to specific conditions. These routines include interrupt service and watchdog support routines. The entire program should now be thoroughly tested.

5. First Look at a C Program

Traditionally, the first program a developer writes in the C language is one which displays the message `Hello World!` on the computer screen. This is a sensible beginning for traditional C platforms where conventional input and output are important and fundamental concepts.

In the world of 8 bit microcontrollers device input and output play radically different roles. Programs rarely have access to keyboard input or screen output devices which are common in traditional C programming ³.

The following introductory program is representative of microcontroller programming. The program tests to see if a button attached to a controller port has been pushed. If the button has been pushed, the program turns on an LED attached to the port, waits, and then turns it back off.

```
#include <hc705c8.h>
// #pragma portrw PortA @ 0x0A; is declared in header
// #pragma portw PortADir @ 0x8A; is declared in header
#define INPUT 1
#define OUTPUT 0
#define ON 1
#define OFF 0
#define PUSHED 1

void wait(register a);           //wait function prototype

void main(void){
    PortADir.0 = OUTPUT; //set pin 0 for output (light)
    PortADir.1 = INPUT;  //set pin 1 for input (button)
    while (1){           // loop forever
        if (PortA.1 == PUSHED){
            wait(1);      // is it a valid push?
            if (PortA.1 == PUSHED){
                PortA.0 = ON;           // turn on light
                wait(10);               // delay (light on)
                PortA.0 = OFF;          // turn off light
            }
        }
    }
} //end main
```

Example 6: A typical microcontroller program

³ Most C compilers for 8 bit microcontrollers do not use `stdio` libraries as these libraries provide functions for input and output rarely used on 8 bit machines.

It is not necessary to understand the specifics of the sample program at this point. It is more important that you become familiar with some of the basic concepts involved in C program development.

The following sections provide a general explanation of the C program in **Example 6**.

5.1 Program Comments

A good programmer includes comments throughout a program. Comments help to explain what the code is doing at a particular point and often state what specific symbols or operations represent.

C compilers use slash and asterisk combinations as comment delimiters. When the compiler encounters a slash immediately followed by an asterisk, `/*`, it treats every character following this pair as a comment until an asterisk immediately followed by a slash, `*/`, is encountered.

Most modern C compilers also accept C++ comment syntax. If the compiler reaches a slash immediately followed by another slash, `//`, in the source code it treats the rest of that line as a comment. The C++ convention is more readable and easier to debug because the effect of the comment syntax does not carry over from one line to the next as in traditional C.

All comments in code examples provided throughout this book use C++ style. If you have a compiler which does not support this comment syntax, you must replace every `//` with `/*` and place `*/` at the end of the comment.

NOTE

Always comment your code. Even if you are sure no other programmer will ever look at your code, a near impossibility, you will still need to understand it. You will often rework code months and even years after it was originally written. Comments drastically improve code readability.

5.2 Preprocessor directives

Example 6 contains three preprocessor directives: `#include`, `#define`, and `#pragma`. Preprocessor directives are specific instructions given to the preprocessor. Preprocessor directives are always preceded by the `#` character which is referred to as a hash mark. These directives are used as follows:

#include <hc705c8.h>

`#include` is one of the most commonly used preprocessor directives. When the preprocessor reaches this directive it looks for the file named in the brackets. In the example above the preprocessor searches for the file `hc705c8.h` which contains device specific specifications for the Motorola 68HC705C8.

If the file is found the preprocessor will replace the `#include` directive with the entire contents of the file. If the file is not found the preprocessor will halt and give an error.

In the example the `#include` directive is used to include the contents of a **header file**. By convention, C language header files have the `.h` extension. Header files contain information which is used by several different sections, or **modules**, of a C program as they contain preprocessor directives and predefined values which help to describe the resources and capabilities of a specific target microcontroller.

#define ON 1
#define OFF 0

`#define` is another commonly used preprocessor directive which is used to define **symbolic constants**. Programs often use a constant number or value many times. Instead of typing in the actual number or value throughout the program, you can define a symbol which represents the value. When the preprocessor reaches a `#define` directive, it will replace all the occurrences of the symbol name in your program with the associated constant. Constants are useful for two specific reasons:

- 1) **Increasing program readability.** A symbolic name is more descriptive than a number. For instance, the name `ON` is easier to understand than the value `1`. Using symbolic constants enhances the readability of your programs and makes them easier to test, debug and modify.
- 2) **Increasing program modifiability.** Since the symbolic constant value is defined in a single place, only one change is necessary if you wish to modify the value: in the `#define` statement. Without the `#define` statement it would be necessary to search through the entire program for every place the value is used and change each one individually.

In the statements `#define ON 1` and `#define OFF 0`, the symbols `ON` and `OFF` are assigned the values `1` and `0` respectively. Everywhere the

preprocessor sees the symbol `ON` it will replace it with the constant `1`; where it sees `OFF` it will replace it with the constant `0`.

```
#pragma portw PortA @ 0x0A;  
#pragma portw PortADir @ 0x8A;
```

The preprocessor handles `#pragma` directives in a slightly different fashion than other preprocessor directives. `#pragma` directives instruct the compiler to behave in a certain way based on the description of the hardware resources of the target computer. `#pragma` statements are most often used in header files which provide the hardware specifications for a particular device.

`#pragma` port directives, for example, describe the ports available on the target computer. The description includes details on port location, whether they are read, write or read/write and the names the program uses to access ports.

In the excerpt from **Example 6** shown above, the compiler is informed that two ports are available. The name `PortA` refers to physical port A's data register, which is available for reading and writing and is located at address `0x0A`. The name `PortADir` refers to physical port A's data direction register, which is available for writing only and is located at address `0x8A`.

5.3 C Functions

C programs are built from functions. Functions typically accept data as input, manipulate data and return the resulting value. For example, you could write a C function that would take two numbers, add them together and return the sum as the result.

5.3.1 The `main()` function

When a computer runs a C program, how does it know where the program starts? All C programs must have one function called `main()` which is always the first function executed.

Notice the notation for `main()`. You specify a function name by following the name with parentheses. This is the notation used by the C compiler to determine when it has encountered a function. As long as the name is not a recognised C command, called a **reserved word**, the compiler will assume it is a function if it is immediately followed by a pair of parentheses. The parentheses may also surround a list of input **arguments** for the function.

```
void main(void){  
    //function statements  
}
```

Example 7: Syntax for the main() function

Example 7 is the definition for the `main()` function in **Example 6**. All the statements that fall between the two braces, `{ }`, have been omitted for example purposes.

The first use of the term `void`, prior to the word `main`, indicates to the compiler that this `main()` function does not return any value at all. The second use of the term `void`, between the parentheses, indicates to the compiler that this `main()` function is not sent any data in the form of arguments.

Braces must surround all statements which form the body of a function. Even functions with no statements in their body require the braces – the braces after a function header indicate to the compiler that you are providing the definition of a function.

5.3.2 Calling a Function

The `main()` function can execute code from other functions. This is referred to as calling another function. The calling function must know about the called function in order to execute its code. A function knows about another function in two ways:

- 1) The entire definition of the called function is positioned earlier in the source file than the calling function.
- 2) A function prototype of the called function is included before the calling function in the same source file.

A **function prototype** describes details of the requirements of a function so that any program code that calls that function will know what information the called function requires. The following is a typical function prototype:

```
void wait(register a);
```

The example above is a function prototype for a function called `wait()`. This function is preceded by the return value `void`; therefore, it does not return a value. Unlike `main()`, the `wait()` function does expect to receive an argument, called a **parameter**. The type of the parameter (`register a`) is

important. It indicates the type of value the parameter will hold – a value of type `register`.

5.4 The Function Body

Every function definition has a **function header**. A function header describes what type of value the function returns, the name of the function, and what input arguments it expects. The body of the function follows the function header. The function body contains a set of statements between braces which are executed when the function is called. There are several different types of C statements.

5.4.1 The Assignment Statement

One of the simplest and most common statements in C is the assignment statement. An assignment statement takes the value of the expression on the right of the equal sign and assigns it to the symbol on the left side of the equal sign. For example:

```
PortADir.0 = OUTPUT;  
PortADir.1 = INPUT;
```

Example 8: Using the C assignment statement

In **Example 8** the symbols `PortADir.0` and `PortADir.1` refer to the first two bits of the port associated with the name `PortADir`.

The first statement assigns the numeric value of the expression on the right of the equal sign to bit 0 of `PortADir`, which represents the port A direction register. From the `#define` directives we know that `OUTPUT` is really a symbolic constant associated with the value 0. Therefore, this assignment statement clears bit 0 of the port A direction register.

By contrast, the second assignment statement sets bit 1 of the port A direction register. How? Recall that `INPUT` is a symbolic constant associated with the value 1 in a `#define` statement.

5.4.2 Control statements

Control statements allow decisions to be made to determine which statements are executed and how often. For example, suppose you need to write a set of instructions for making coffee in a coffee maker. The amount of water you

pour into the coffee maker depends upon the number of cups you want to make. At some point in your instructions, you need to allow the person following them to make a decision about the number of cups needed and, therefore, the amount of water needed. You might say: “if you want to make 4 cups of coffee, then use 5 cups of water”.

In C decisions are made using control statements. Control statements can select between two or more paths of execution and repeat a set of statements a given number of times. Some common control statements are:

while

```
while(1){  
    // statements  
}
```

The `while()` control statement instructs the computer to repeat a set of instructions (**loop**) as long as a condition is valid. The condition is an expression placed in the brackets which follow the `while` statement. C considers any condition which does *not* evaluate to 0 to be true and any condition which *does* evaluate to 0 to be false.

In **Example 6** the condition is the integer 1b (binary), which is interpreted as true. Therefore, once the computer begins to execute statements inside the braces of the `while` loop, it will not terminate until the computer malfunctions or is turned off. This kind of loop is often called an **infinite loop**.

In traditional C programming, an infinite loop is usually avoided. However, it is often used in embedded systems programming. An embedded controller typically executes a single program “infinitely”. Only when the controller is reset or turned off will the loop terminate.

if

```
if (PortA.1 == PUSHED){  
    PortA.0 = ON;  
}
```

Example 9: The if statement syntax

The `if()` statement provides the ability to make decisions. If the `if` statement condition is true then the computer executes the statements in the `if` body. In **Example 9**, the value of `PortA.1` is compared with the value of `PUSHED`, if data bit number 1 is set (has the value 1) then the program will execute any statements in the `if` body. The body statement sets port A data bit number 0 by assigning it the value of `ON`.

```
while (1){
    if (PortA.1 == PUSHED){
        PortA.0 = ON;
    }
}
```

Example 10: Nesting if and while statements

When the `if` decision is placed inside a `while` loop, the program will test bit 1 in `PortA` regularly. Assume a button is attached to pin 1 of port A and an LED to pin 0 of `PortA`. We have written a small control program which will continually test the button attached to pin 1. When the button is pushed, bit 1 of `PortA` will be set. When bit 1 is set and the `if` statement is executed, bit 0 is set. The LED attached to `PortA` pin 0 will be set to 1 and will light up.

5.4.3 Calling Functions

A program can delegate a task by calling another function. Once the program turns on the LED in **Example 10** it never turns it off. Remember, the `while` loop is an infinite loop. How can we solve this problem?

One solution is to write a function called `wait ()` which creates a delay and then turn the LED off. Consider the following example code fragment:

```
while (1)
{
    if (PortA.1 = PUSHED)
    {
        PortA.0 = ON;
        wait(10);    \\ wait ten seconds
        PortA.0 = OFF;
    }
}
```

Example 11: Calling one function from another

When the `wait ()` function is used and the button is pushed, the program turns the LED on by setting bit 0 of `PortA`. The `wait ()` function causes a delay of ten seconds. After the `wait` function has finished and ten seconds have passed, the program turns off the LED by clearing bit 0 of `PortA`.

5.5 The Embedded Difference

Several things make the program in **Example 6** typical of embedded systems programs in C.

5.5.1 Device Knowledge

Most embedded systems programs include a header file which describes the target processor. These header files contain descriptions of reset vectors, ROM and RAM size and location, register names and locations, port names and locations, register bit definitions and macro definitions. Most compiler companies will provide header files for devices supported by their compilers.

Another important aspect of device knowledge is the limits of the device for which the program is written. For example, a certain device may have very limited memory resources and great care must be taken in developing programs which use memory frugally. Along with issues of size comes issues of speed. Different devices run at different speeds and use different techniques to synchronise with peripherals. It is essential that you understand device timing for any embedded systems application.

5.5.2 Special Data Types and Data Access

Embedded systems developers require direct access to registers such as the accumulator. In **Example 6** the `wait()` function is called with an argument of type `register_a`. This is a special type which represents the accumulator.

Embedded developers are much closer to their target hardware than other programmers. They often access and control the basic hardware of the device they are programming.

5.5.3 Program Flow

The previous section mentioned the regular use of the infinite loop `while(1)`. Embedded developers often use program control statements which are avoided by other programmers. For example, the `goto` statement is used regularly by embedded developers and is often condemned by other programmers.

5.5.4 Combining C and Assembly Language

Many developers prefer to write some code segments in assembly language for reasons of code efficiency or while converting a program from assembly language to C. Most compilers for 8 bit microcontrollers allow the inclusion of inline assembly, assembly language in a C program.

The following two definitions of the `wait()` function show the function written in C and the equivalent function in Motorola 68HC705C8 assembly language.

```
//C function
void wait(register delay){
    while (--delay);
}

//function with inline assembly
void wait(register){
    char temp, time;
    // ocap_low and Ocap_hi are the output compare register
    //this register is compared with the counter and the ocf
    //bit is set (bit 6 of tim_stat)
    #asm
        STA time        ;store A to time
        LDA #$A0         ;load A with A0
        ADD ocap_low     ;add ocap_low and A
        STA temp         ;store A to temp
        LDA #$25         ;load A with 25
        ADC ocap_hi      ;carry + ocap_hi + accumulator
        STA ocap_hi      ;store A to ocap_hi
        LDA temp         ;load temp to accumulator
        STA ocap_low     ;store a to ocap_lo
    LOOP    BRCLR 6,tim_stat,LOOP ;branch if OCF is clear
        LDA ocap_low     ;load ocap_lo to A
        DEC time         ;subtact 1 from time
        BNE LOOP        ;branch if Z is clear
    #endasm
}
```

Example 12: C functions containing inline assembly language

5.5.5 Mechanical Knowledge

Techniques used in an embedded system program are often based upon knowledge of specific device or peripheral operation. For example, **Example 6** calls the `wait()` function with a value of 1 after it has detected that the button is pushed and then checks to see if the button is still pushed. The code is written in this manner to deal with the issue of contact bounce.

When a button is pressed it “bounces” which means that it is read as several pushes instead of just one. It is necessary to include debouncer support in order to ensure that a real push has occurred and not a bounce. The `wait()` function creates a delay before the button is checked again. If the button is no longer in a pushed state then the push is interpreted as a bounce and the program waits for a real push.

6. C Program Structure

The previous section described some typical features of a very simple program. In this section we will examine in greater detail the building blocks of the C language.

A C program is built from three components:

- 1) **Directives** are directives handled directly by the preprocessor
- 2) **Declarations** are instructions for the compiler to record the type and associated memory locations of symbols
- 3) **Statements** are the executable instructions in a program

6.1 C Preprocessor Directives

The simple C program shown in **Example 6** in the previous section introduced several preprocessor directives:

- `#include` directives include the contents of another file
- `#define` directives define symbolic constants
- `#pragma` directives describe details of the target hardware

Section 13, The C Preprocessor, provides a detailed explanation of the preprocessor.

6.2 Identifier Declaration

Declarations define the name and type of identifiers used in your program. One benefit of programming in a high level language is the ability to construct generic groups of instructions, called **functions**, to perform tasks whose steps are not dependant upon specific values. For example, you can write instructions to add together two numbers without knowing the values of the numbers. How can this be done? Through the use of **identifiers**.

An identifier can either represent a value, called a **variable**, or a group of instructions, called a **function**. C identifiers represent addresses in computer

memory. At a given memory location the computer can store a value, or a group of program instructions.

6.2.1 Identifiers in Memory

The compiler allocates memory for all identifiers. As the compiler reads a program, it records all identifier names in a symbol table. The compiler uses the symbol table internally as a reference to keep track of the identifiers: their name, type and the location in memory which they represent.

When the compiler finishes translating a program into machine language, it will have replaced all the identifier names used in the program with instructions that refer to the memory addresses associated with these identifiers.

6.2.2 Identifier names

An identifier name can be any word beginning with a letter or underscore character. The rules for naming identifiers are quite straightforward. An identifier can be almost any word that begins with a letter or underscore character, followed by 0 or more letters, numbers or underscore characters.

★ An identifier can not be a C keyword

The C language has keywords which the compiler reserves because they have special meaning in the language. For example, the word `if` is used to signify the beginning of a decision block. A keyword may not be used as an identifier name. Some standard keywords in C are:

<code>auto</code>	<code>default</code>	<code>if</code>	<code>short</code>	<code>union</code>
<code>break</code>	<code>do</code>	<code>int</code>	<code>signed</code>	<code>unsigned</code>
<code>case</code>	<code>else</code>	<code>long</code>	<code>static</code>	<code>void</code>
<code>char</code>	<code>enum</code>	<code>main</code>	<code>struct</code>	<code>volatile</code>
<code>const</code>	<code>extern</code>	<code>pointer</code>	<code>switch</code>	
<code>continue</code>	<code>for</code>	<code>return</code>	<code>typedef</code>	

Example 13: Common C keywords

Compilers that provide special enhancements or extensions to the language will add keywords to this list, so you must check the documentation for your particular compiler to find out what other words not to use for identifier names.

✪ Identifiers only have certain significant characters

Most compilers support identifier names of at least 31 characters in length. This allows you to use precise and meaningful names for variable and function names. However, to conserve memory a compiler will often only consider some characters in an identifier name as *significant*. This means that two identifiers which may seem different are treated as the same symbol by the compiler. For example, a compiler which only considered the first 5 characters of an identifier as significant would treat the following two identifiers as if they were the same symbol:

```
PortADir
PortA
```

Notice that even though the two identifiers are different words, the first five significant characters are identical.

6.2.3 Variable Data Identifiers

Identifiers which represent variable data values, called variables, require portions of memory which can be altered during the execution of the program. The compiler will allocate a block of its data memory space, usually in RAM, for each variable identifier.

For example, the declaration `int currentTemperature;` for the variable `currentTemperature` will cause the compiler to allocate a single byte of RAM.

The keyword `int` in the variable declaration tells the compiler that `currentTemperature` will contain an integer value and will require a single byte of RAM to contain this value.

6.2.4 Constant Data Identifiers

Identifiers which represent constant data values are allocated from computer program memory space. Identifiers which represent constant data values do not require alterable memory: once the value of a constant has been written in

memory it need never change. Therefore, the compiler will allocate a block of its program memory space, usually in ROM, for each of these identifiers⁴.

To declare a constant data value, use a declaration such as:

```
const int maximumTemperature = 30;
```

This declares a variable called `maximumTemperature` and sets its initial value to 30. The keyword `const` tells the compiler that the identifier is a constant and that a single byte in ROM should be reserved to contain the value 30. When the identifier `maximumTemperature` is used in the program it refers to the memory location in ROM which contains the value 30.

6.2.5 Function Identifiers

Function identifiers are not altered during program execution. Once the value of a function has been written in the computer's memory it need never change.

When a function is defined, the compiler places the program instructions associated with the function into ROM. What happens to the local variables used in a function's body of statements? The compiler will write in the data memory addresses where local variable values will be stored in RAM when the program runs.

6.3 Statements

When a program runs it executes program statements. Declarations describe the memory locations which statements can use to store and manipulate values.

The most frequently used statement in any programming language is the assignment statement. C provides many different ways to construct an assignment statement; however, the following example shows the simplest way:

```
currentTemperature = 20;
```

The compiler will generate an instruction to store the value 20 in the RAM memory location set aside for the `currentTemperature` variable.

⁴ This is not always the case. However, you can safely assume that all C constant values are stored in machine program memory space.

6.3.1 The Semicolon Statement Terminator

All statements in C must end with a semicolon. C uses the semicolon as a statement **terminator**⁵. One of the most common errors in C programming is an extra, missing or misplaced semicolon. If you leave out a semicolon the C compiler will not know where a statement should end.

For example, suppose you wrote the following two statements. The compiler would produce an error. Why?

```
currentTemperature = 20
currentTemperature = 25;
```

Forgetting the semicolon at the end of the first line forces the compiler to read both lines as one statement instead of two. According to the compiler you have written the following instruction:

```
currentTemperature = 20 currentTemperature = 25;
```

Notice that the C compiler does not care about white space between tokens as it reads through your program. White space includes space, tab and end-of-line characters. On some computers the end of line will be a single linefeed character, while on others it will be a linefeed and carriage return together. C compilers ignore both carriage returns and linefeeds.

6.3.2 Combining Statements in a Block

When you write a C function you must include function statements as part of the function definition. Statements belonging to a function are indicated by surrounding them with braces which immediately follow the function header. For example, **Example 6** in the previous section has braces surrounding all the statements in the `main()` function.

You may create statement blocks at other times in your program. For example, notice the braces after the `while` and `if` statements:

```
while (1){    // this brace begins the block for while
    if (PortA.1 = 1){    // this brace begins the if block
        PortA.0 = 1;
        wait(10);
        PortA.0 = 0;
    }    // this brace closes the block for if
```

⁵ Unlike languages PASCAL-like where the semicolon is used as a statement *separator*.

```
} // this brace closes the block for while
```

Example 14: Using braces to delineate a block

The general format for the `while` statement looks like:

```
while (condition) statement;
```

However, since you can substitute a statement block anywhere a single statement can occur, the most commonly used form of the `while` statement looks like:

```
while (condition) {  
    statements  
}
```

Example 15: The while loop

It is good programming practice to use braces whenever you use a loop or conditional construct such as `while` and `if`, even with a single statement block. The braces ensure that anyone reading your program code can tell exactly which statements belong to the `while` or `if`.

7. Basic Data Types

It is easy to see how the computer stores binary values in memory as that is the manner in which its memory is structured. We have also seen how the computer stores other types of numbers, such as hexadecimal and decimal, by converting them to binary form. This section examines how other types of data can be used.

7.1 The ASCII Character Set

A computer can store a number in its memory. What about a character? People use alphabets to encode linguistic information while computers must use binary notation. To resolve this problem, computer programmers have settled on encoding schemes for representing characters with numbers such as ASCII (American Standard Code for Information Interchange) encoding. In the ASCII character set each character is associated with an integer value. When the computer needs to store a character it uses the ASCII integer value associated with that character and stores the number in binary notation.

7.2 Data types

Data types act as filters between your program and computer memory. Data types in C provide rules for the storage and retrieval of information from computer memory. C data types also provide a set of rules for acceptable data manipulation.

The primary distinguishing characteristic of a data type is its size. The size of a data type indicates the amount of memory the computer must reserve for a value of that type. For example, on 8 bit microcontrollers the `int` data type (used for storing integer values) is a single byte in size⁶ while a `long` or `long int` data type is two bytes in size. When the compiler translates a program it must write the instructions to account for this size difference. The computer

⁶ The size of the `int` data type in C is the same as the amount of information the computer can process. Since 8 bit microcontrollers work with a byte at a time the size for `int` is 1 byte (or 8 bits). On most modern computers `int` varies from 24 bits to 64 bits.

will know to set aside a single byte for all the `int` values in your program and two bytes for all the `long` values.

7.3 Variable Data Types

When you declare an identifier used in your program, either as a variable or a function, you specify a data type as part of the declaration. The compiler will allocate the appropriate amount of computer memory for use with each identifier.

It is possible to declare a number of variables of the same type in the same declaration by including a list of identifier names separated by commas. Good programmers will most often use this method for declaring a group of variables that serve a similar function within the program. A typical case is a group of counters the programmer will use to regulate the control of program flow through loops:

```
int    currentTemperature;  
char   tempScaleUsed;  
long   TempDifference;  
int    count1, count2, count3;
```

Example 16: Declaring variable types

A declaration can also be used to ensure that a variable will be assigned a certain value when it is allocated. When this is done the compiler allocates the appropriate space for the variable and immediately assigns a value to that memory location, for example: `int currentTemperature = 20;` allocates 1 byte for the variable `currentTemperature` and assigns it the value 20. This is called **initializing** the variable.

Initialization ensures that a variable contains a known value when the computer executes the first statement which uses that variable. If variables are not initialized in their declarations, their values are unknown until they are initialized.

7.3.1 Variable Data Type Memory Allocation

When the compiler comes across a variable declaration it checks that the variable has not previously been declared and then allocates an appropriately

sized block of RAM. For example, an `int` variable will require a single word (8 bits) of RAM or data memory⁷.

When the compiler allocates memory for a variable it decides where to place the variable value based on the existing entries in its symbol table. Since the compiler cannot know what value lies at the address allocated for a particular variable at compile-time, you can not depend upon a specific value for a variable the first time it is used.

Compile-time is the point at which the compiler translates a program into machine code. **Run-time** indicates the point at which the machine code is executed on the host computer. It is useful to remember that compilers have little or no knowledge about a machine's internal state at run-time.

Declarations that initialise variables are very useful – they ensure that you can predict what a variable memory location will contain at run-time. When the compiler reads a declaration which also initializes a variable it first allocates an appropriate block of memory, then immediately loads the appropriate value into that location.

Please note that variable declarations which contain an initialization will automatically generate machine code to place a value at the address allocated for the variable. Normal variable declarations do not generate any code because the machine code contains the address allocated for such a variable. This is not the case for either global variables or *static* local variables – if they are not initialized in their declaration the compiler will initialize them by setting their initial values to 0. The compiler will produce machine instructions to load the 0 value into the appropriate addresses.

7.3.2 Variable Scope

Not all parts of a program recognize declared variables. The visibility of a declared variable is called the variable's scope. If a portion of a program lies outside a variable's scope then the compiler will give an error if you refer to the variable in that portion. The scope of a variable includes the locations in a program where the variable is a recognized and meaningful symbol. Outside that scope the variable is an unknown or undefined symbol.

⁷ The amount of memory required for an `integer` variable varies from computer to computer. 8 bit microcontrollers have a natural integer size of 8 bits.

7.3.3 Global Scope

If you declare a variable outside all statement blocks, the scope of the variable reaches from its declaration point to the end of the source file. Variables declared in this manner are called **global variables** because they can be used by any program code which comes after them in the same source file.

A variable declared outside a statement block *can* be accessed by any statement in your program by declaring the variable in a certain way. In order for a statement block or separate program file to access to such a variable, it must be declared as an **external symbol**. This means using the `extern` storage class modifier. For example, the following declaration tells the compiler to look for the original declaration of `currentTemp` in another file or below in the same file: `extern int currentTemp;`

The use of `extern` in a variable declaration is similar to the use of a function prototype – it informs the compiler of a variable’s name and data type so that it can be used before it is actually defined. As with function prototype declarations, the compiler does not allocate memory when it sees an `extern` variable declaration.

7.3.4 Local Scope

A variable declared inside a statement block has a scope from the declaration to the end of the statement block. Variables declared inside a statement block are called **local variables**, as they are accessible only to statements which follow them within the same statement block. Typically, programmers will declare variables whose scope is local to a specific function. The variable name and value will be defined only within that function and other functions cannot directly refer to the variable.

7.3.5 Declaring Two Variables with the Same Name

What happens if you have two functions which each contain local variables with the same name? Since a variable is local to its respective functions the compiler can distinguish between identically named variables. A variable name *must* be unique within its scope.

What happens when scopes overlap? The most recently declared instance of a variable is used. If you declare a global variable called `temp` outside all statement blocks and a local variable called `temp` inside your `main()`

function, the compiler gives the local variable precedence inside `main()`. While the computer executes statements inside `main()`'s scope (or statement block), `temp` will have the value and scope assigned to it as a local variable. When execution passes outside `main()`'s scope, `temp` will have the value and scope assigned to it as a global variable.

7.3.6 Why Scope is Important

Why is scope an important concept? It can provide tangible benefits to programmers.

Since C is a function-oriented language where programs are built from collections of functions, variable scope promotes **data abstraction**. Variables declared inside a function remain local to that function only. Other functions in the program can use identical local variable names without creating conflicts. This means that you can use functions in your program and only know about the function interface. It is not necessary to see inside a function to use it in a program, it is only necessary to know what to pass in and what will be returned.

Data abstraction allows a programmer to create a function which others can make use of the without seeing the function source code. This may sound dangerous but *all* C compilers take advantage of this principle. The standard library functions available with all C compilers depend upon data abstraction to be useful – programmers include standard library functions in their code all the time without worrying about potential variable name conflicts.

7.4 Function Data Types

A function data type allocates memory for the type of value the function returns. Function identifiers work differently than variables. When a function is defined a data type for the function must be included. Instead of indicating the amount of memory set aside for the function *itself* it indicates the amount of memory the compiler needs to reserve for the value *returned* by the function. For example, a function of type `int` returns a signed integer value and 8 bits are reserved for the return value.

Suppose we have a function defined as follows:

```
void wait(int timeInSeconds);
```

The `void` keyword indicates to the compiler that the function *will not* return a value; therefore, no memory is allocated for a return value.

7.4.1 Function Parameter data types

Parameter data types indicate the size of memory reserved for function parameter values. We define a data type for the parameter the function expects to be passed when it is called. The declaration of `timeInSeconds` as an `int` in the function declaration `void wait(int timeInSeconds);` tells the compiler to allocate a single byte to hold the parameter value when the function is called.

The `void` keyword can also be used in a function parameter list:

```
void main(void);
```

This indicates to the compiler that the function does not expect to receive any parameter values when called. The compiler does not allocate any memory for `void` parameters.

7.5 The Character Data Type

The C language character data type, `char`, stores character values and is allocated 1 byte of memory space. Microcontrollers do not often manipulate alphabetic information, but sometimes it is required. The most common use of alphabetic information is reading input from a keyboard device, where each key typed is indicated by a character value. The `char` type uses a single byte of memory and stores the value of each character by storing its ASCII code.

7.5.1 Assigning a character value

When assigning a character value to an identifier you must place the character in single quotes. The quotes tell the compiler that the value is a character constant and not the name of another identifier.

```
char firstLetter;  
firstLetter = 'a';  
firstLetter = a;
```

Example 17: Assigning a character value

The first assignment in the example above places the ASCII value for the character `a` in the memory location assigned to the `firstLetter` variable. When the compiler reads the second assignment statement, it assumes that `a` is the name of a second variable. If no variable called `a` exists the compiler will generate an error.

7.5.2 ASCII Character Arrangement

The order in which ASCII arranges its characters is called its **collating sequence**. The collating sequence is arranged so that the letters 'A' through 'Z' are in unbroken, ascending order with the decimal values 65-90, as are the letters 'a' through 'z' with the decimal values 97-122. In addition, the digits '0' through '9' are in unbroken, ascending order with the decimal values 48-57. The collating sequence allows for easy sorting of characters and the use of characters in simple arithmetic operations.

7.5.3 Numeric Characters

Numeric characters are *not* the same as integer values. It is important to understand that the character '3' is not the same as the integer 3. In fact, the ASCII decimal integer associated with the character '3' is 51.

It is also important to remember that the upper case alphabetic characters have lower integers associated with them than do the lower case characters; 'A' and 'a' are not the same character. A side effect of this property leads to logical comparisons of character values sorting capital letters before their lower case counterparts. The expression `(A < a)` will therefore evaluate to true.

7.5.4 Escape Sequences

You can specify any character in the ASCII set with a special escape sequence – a backslash immediately followed by the octal or hexadecimal ASCII value for the character⁸. This supports character values that can not be typed using a keyboard.

You can represent common special characters using escape sequences. For example, the escape sequence to produce the carriage return character, the character produced when someone types the **⏏** key, is `\r`.

Escape sequences are useful for typing a literal character that the C compiler might interpret in another way. For example, to type a literal single quote

⁸ For historical reasons you can not specify a decimal ASCII value in an escape sequence. The general format for the escape sequence is as follows:

```
\###    // octal value
\x##    // hexadecimal value
```

character and avoid the compiler interpreting it as a character constant delimiter simply precede it with a backslash, `\'`. To assign the single-quote character to a `char` variable use the statement `char singleQuote = '\'';`

7.6 Integer Data Types

Integer values can be stored as `int`, `short` or `long` data types. The default size for a number on most microcontrollers is 8 bits (a single byte). Therefore, the `int` data type for these computers requires a single byte of storage. Some compilers offer the ability to switch to using 16 bit integers by default. The size of `int` values usually equals the natural data size of the target computer.

7.6.1 Integer Sign Bit

C allows you to manipulate both positive and negative integer values and uses different methods to store each value in memory. Signed integer values have the left-most bit reserved for a **sign bit**. The state of the sign bit indicates whether the stored value should be treated as a positive or negative value.

The existence of a sign bit means that there are only 7 bits left in which to store the actual value of the integer. An 8 bit signed `int` value can therefore range from -2^7 to 2^7-1 . There is one more value available on the negative side of zero because zero itself counts as a positive value.

7.6.2 The short Data Type

On many traditional C platforms, the size of an `int` is more than 2 bytes. The `short` data type helps compensate for varying sizes of `int`. On platforms where an `int` is greater than 2 bytes, a `short` should be 2 bytes in size.

On platforms where an `int` is 1 or 2 bytes in size —most microcontrollers— the `short` data type will typically occupy a single byte. This can be useful for embedded system programmers, especially on systems which provide a switch to “turn on” 16 bit `int` values. In these cases, you can maintain code portability by using `short` for those values that *require* 8 bits and `long` for values which require 16 bits.

Like the `int`, the `short` data type uses a sign bit by default and can therefore contain negative numbers.

7.6.3 The long Data type

Should your program need to manipulate values larger than an `int`, you can use the `long` data type. On most platforms the `long` data type reserves twice as much memory as the `int` data type. On 8 bit microcontrollers the `long` data type typically occupies 16 bits; this allows the representation of signed integers ranging from -2^{15} to $2^{15}-1$.

It is important to note that `long` integer values are almost always stored in a memory block larger than the natural size for the computer. This means that the compiler must typically generate more machine instructions when a program uses `long` values. Programs will usually operate more quickly and efficiently if they only use 8 bit data types.

7.6.4 Different Notations

Integer data types usually hold values expressed in decimal notation. It is also possible to express an integer value in other notations. For example, the following declarations assign the same value to their respective variables expressed in different notations: All C compilers allow the expression of integer values in decimal, octal and hexadecimal notation. The ability to express values in binary notation is an enhancement to the language not available on all compilers.

```
int decimalInt      = 32;
// all octal values begin with 0
int octalInt        = 040;
// all hex values begin with 0x
int hexadecimalInt  = 0x20;
// all binary values begin with 0b
int binaryInt       = 0b00100000;
```

Example 18: Octal, hex and binary notation

Be careful to maintain the distinction between the value assigned to a variable, and the notation used to write that value. Remember that the computer uses a series of binary bits to store all the numbers your program uses regardless of the notation used when you write your program.

7.7 Data Type Modifiers

So far we have seen two general classes of simple data types: the character data type `char` and the integer data types `int`, `short` and `long`. By default, the

char type holds values from 0 to 255 and does not permit any negative values. However, int data types permits a range of both negative and positive values.

The C language allows you to modify the default behaviour of simple data types and thereby produce char variables which *can* hold negative numbers and integer variables which permit only positive values.

7.7.1 Signed and Unsigned

The **data type modifiers** signed and unsigned allow you to specify whether you wish a variable to hold negative numbers or not. They instruct the compiler whether or not to include a sign bit in the allocated memory.

By default, char variables are unsigned and cannot hold negative values. Also by default, integer variables (int, short and long) are signed and can hold negative values. Actually, the short and long types are not data types, they are data type modifiers. As a result, you often see declarations such as:

```
short int myShortInt;  
long int myLongInt;
```

Because int is the default data type in C you can simply declare variables as short and long. Some programmers insist that you should *never* take this short cut. However, some compilers actually implement the short and long as separate data types.

Place modifiers before the data type in a variable declaration. For example:
unsigned int myAge;

7.7.2 Other Data Type Modifiers

There are several data type modifiers available:

auto	const	extern	far
near	signed	static	unsigned
volatile			

Example 19: Data type modifiers

7.8 Real Numbers

While many computers make extensive use of real, or floating point numbers (numbers with digits on both sides of the decimal place) 8 bit microcontrollers

do not. The resources needed to store and manipulate floating point numbers can place overwhelming demands on an 8 bit computer and usually the value gained is not worth the resources expended. Some C compilers for 8 bit microcontrollers offer limited support for floating point data types, but most do not.

7.8.1 The float Data Type

The fundamental data type for representing real numbers in C is the `float` type. Those compilers that do offer this data type store real numbers as **floating point values** – a special way of representing real numbers in computer memory. The maximum value for the target computer is defined in a C header file called `values.h` as a symbolic constant called `MAXFLOAT`.

7.8.2 The double and long double Types

C compilers generally allocate 4 bytes for a float variable – you can see why 8 bit microcontrollers might have difficulty handling such values– which provides approximately 6 digits of precision to the right of the decimal. You can have greater precision with the `double` and `long double` data types. Compilers typically allocate 8 bytes for a `double` variable and more for a `long double`. There are approximately 15 digits of precision with `double` values and perhaps more from `long double` values.

7.8.3 Assigning an Integer to a float

You can assign an integer value to a floating point data type but you must include a decimal and a 0 to the right of the decimal.

```
myFloatVariable = 2.0
```

8. Operators and Expressions

The chief purpose of programming is providing the computer with a set of generalized instructions for solving problems. This concept is so important to programming that programmers use a specific name for a set of generalized instructions – the term **algorithm**. In fact, many programmers insist that programming consists of two simple steps:

- 1) Choosing suitable data structures to contain and organize program data
- 2) Choosing the appropriate algorithm to manipulate that data.

Once you have determined variable and function data types it is time to examine how the functions will manipulate the data.

8.1 Operators

Variables and functions contain and pass values among program modules. Operators allow you to perform calculations with these values. C has more operators than most other programming languages.

When you write a program in any language a significant portion of the program is dedicated to doing simple data manipulation such as incrementing or decrementing counters and multiplying or dividing a variable by a number. In most languages these simple manipulations require a statement of some length or more than one statement. C encapsulates many of the most common simple data manipulations in its operator set.

For example, consider incrementing a counter. In most programming languages the following statement is required to increment a counter.

```
counter = counter + 1;
```

In C incrementing can be done with the following statement:

```
counter++;
```

The original purpose of the increment operator was to create faster and more efficient code. Most computers have a low level hardware instruction which performs a simple increment upon a value. This instruction uses less resources

than the instructions required to add two numbers together and assign the result to a third which is the case in the first example.

Modern compilers are quite sophisticated, especially in the optimization of code during translation to machine language. Most compilers will see the first example written in a program and translate it into the speedier machine-level increment. Programmers who care about readability and clarity will insist upon using the syntax of the first example and allow the compiler to generate the faster and more efficient code.

Recent criticisms of C describe the operator set as overly large. It is true that badly written C code tends to rely on the effects and side effects of operators, making it very difficult to read and debug. Often these problems are a function of bad programming style.

8.2 C Expressions

All calculations and data manipulation in are accomplished using expressions. In C an expression is formed by combining operators, constants and variables. The simplest expression in C is a single constant or identifier with no operators. Imagine constants and identifiers as building blocks and operators as a set of predefined ways to combine these blocks. An expression can be as simple as a single block or it can consist of single blocks and additional operators. It is possible to construct elaborate groups of blocks which are themselves expressions.

An expression is converted to a statement by terminating it in a semicolon. The following example shows a valid C statement:

```
5;
```

All C expressions have values⁹ which your program uses when the expressions are evaluated. In the previous example, the value of the single expression in the statement is 5. Operators work by acting upon the expression values. For example, the + operator takes the value from one expression and adds it to the value of another expression:

```
2 + 3;
```

⁹ This is not always the case. A call to a `void` function has, by definition, no value. However in practical terms an expression always evaluates to some value.

The combination of two expressions (2 and 3) with the addition operator forms a single, larger expression. When the computer evaluates the entire expression, its value consists of the sum of the two smaller expression values joined by the addition operator, the value 5.

8.2.1 Binding

How does the compiler determine which expressions apply to each operator in a program's statements? The rules which govern operator behaviour specify the number of expressions the operator requires – we indicate this relationship by saying that an operator binds to a number of expressions. For example, an operator that manipulates the value of a single expression binds to a single expression.

8.2.2 Unary Operators

Operators that bind to a single expression are called *unary operators*. Some unary operators bind to the expression to their immediate right – these are called prefix unary operators where the operator act as a prefix to the bound expressions. Other unary operators bind to the expression to their immediate left. These are called postfix unary operators. For example:

```
a[6];      //postfix unary operator
a++;       //postfix unary operator
++a;       //prefix unary operator
&a;        //prefix unary operator
```

Example 20: Postfix and prefix unary operators

8.2.3 Binary Operators

Operators that bind to two expressions are called *binary operators*. Binary operators bind the expressions located to their immediate left and right. For example, the addition operator used in our previous example is a binary operator and uses the general form $a+b$.

```
a * b;      //multiply two expressions
a / b;      //divide two expressions
a - b;      //subtract one expression from another
a + b;      //add two expressions
a >> b;     //shift bits right
```

Example 21: Sample binary operators

8.2.4 Trinary Operator

C supports a single *trinary operator* which binds to three expressions. The conditional operator `?:` binds to three expressions, for example:

```
a ? b : c
```

Example 22: Trinary conditional operator

The compiler evaluates expression `a`, if it is *true* (non-zero) then the value of the entire expression is the value of expression `b`. If expression `a` is *false* (zero), then the value of the entire expression is the value of expression `c`.

8.2.5 Operator Precedence

Statements often contain more than one operator. For example, consider the conversion from degrees Celsius to degrees Fahrenheit. The equation for this operation is:

$$F^{\circ} = (C^{\circ} - 32) \times \frac{9}{5} + 32$$
$$C^{\circ} = (F^{\circ} - 32) \times \frac{5}{9}$$

The equivalent expressions in C can be written as:

```
Fahrenheit = Celsius * 9 / 5 + 32;  
Celsius = Fahrenheit * 5 / 9;
```

Example 23: Combining operators in a statement

Note that the statements in **Example 23** are ambiguous. Is `Celsius` reduced by 2, then multiplied by two, then added to 30? This is the desired order of operations; however, without any guidelines as to how to proceed, this statement would not necessarily be executed as expected depending on which operation is performed first.

In order to circumvent the problem of ambiguity, C provides a set of precedence rules. These rules dictate the order in which expressions bind to operators. The complete C precedence rules are available at the end of the book. There are *two* simple rules which allow programmers to avoid creating ambiguous expressions:

- 1) Multiplication and division operators bind before addition and subtraction
- 2) Brackets explicitly declare binding order

Consider **Example 23**: the first rule tells us that multiplication and division are done before addition and subtraction. This means that the expression `2 * 2` will be evaluated first. The result of this expression will then bind with the `-` operator, along with `+ 30`. Putting brackets around the first part of the calculation explicitly demonstrates the desired order of operations:

```
Fahrenheit = (Celsius - 2) * 2 + 30;
```

The revised statement implements the second rule: the brackets leave the compiler with no doubt about which part of the calculation to perform first. More importantly, the brackets explicitly depict the programmer's intentions.

8.2.6 The = Operator

The assignment symbol, `=`, is an operator and has a precedence. C also provides more complex assignment operators which all share the same precedence as `=`. Assignment operators have a lower precedence than most other operators, therefore assignment statements will usually behave as you expect them to. There is one operator with lower precedence than the assignment operators – the comma operator. The comma operator is used to concatenate two or more expressions together into a single expression, for example:

```
for (i=0, j=8; i<8; i++, j--)
```

Example 24: Concatenating expressions with the comma operator

The most common use for the comma operator is inside the initialization or condition expression of `for` or `while` loops.

C allows statements with more than one assignment operation. For example, you can initialise a number of counter variables with a single statement in C. The parentheses in the second line show how each operator in the statement naturally binds.

```
counterOne = counterTwo = counterThree = 1;
(counterOne = (counterTwo = (counterThree = 1)));
counterOne = (counterTwo = counterThree) = 1;
```

Example 25: Combining assignment operators in statements

It is possible to enforce a different binding order with assignment operators by using parentheses. This is shown in the third line where `counterTwo` is assigned the value of `counterThree` because the assignment inside the parentheses occurs first. Then `counterTwo` and `counterOne` are assigned the value of 1. `counterThree` is never assigned the value 1 and the original

assignment of its value to `counterTwo` is overridden. The previous value of `counterThree` is preserved.

Example 24 and **Example 25** show the nature of assignment operators and the importance of placing parentheses around expressions to establish precedence. Statements containing multiple assignment operators should be avoided as they can introduce many debugging difficulties.

8.3 Arithmetic Operators

The arithmetic operators (+, -, *, /, %) perform simple arithmetic on expressions. The first three arithmetic operators add, subtract and multiply values.

```
gamesPlayed = wins + losses;
balance = balance - withdrawal;
area = height * width;
```

Example 26: Addition, subtraction and multiplication operators

The remaining arithmetic operators warrant a few comments.

```
numYears = numMonths / 12;
extraMonths = numMonths % 12;
```

Example 27: Division and modulus operators

The division operator, /, returns the whole quotient. Any fractional portion of the division is truncated and lost. Remember that truncation is not the same as rounding. For example, the expression `5/2` returns the value 2, not 2.5 or 3.

Truncation only occurs during integer division. If floating point numbers are involved in the operation, then the division operator will perform a floating point divide.

The modulus operator, %, returns the remainder of a division operation. For example, the expression `5%2` returns the value 1. Thus, if you have calculated a total number of months, you can easily convert to the number of years and number of extra months using the following expressions:

```
years = totalMonths/12;
extraMonths = totalMonths%12;
```

Example 28: Differentiating the division and modulus operators

8.3.1 Increment and Decrement Operators

The increment and decrement operators are unary operators with higher precedence than the arithmetic operators. The increment operator, `++`, adds one to its binding identifier, while the decrement operator, `--`, subtracts one.

You can use the increment and decrement operators in two ways: prefix and postfix. All of the following expressions are valid.

```
++counter; //prefix increment
counter++; //postfix increment
--counter; //prefix decrement
counter--; //postfix decrement
```

Example 29: Prefix and postfix notation for increment and decrement

Because the increment and decrement operators modify the value of the identifier they bind to, they can *not* be bound to complex expressions. The following statement is not valid: `++(a + b);`.

It is essential to understand how various forms of the increment and decrement operators return values.

★ Postfix returns a result and *then* increments or decrements

When you use the postfix versions of the increment and decrement operators, the computer will return the value of the operator's binding expression *first*. Then it will perform the increment (or decrement). Consider the following example.

```
counter=0;           //counter set to 0
j=counter++;         //j set to 0
i=counter;           //i set to 1

counter=10;          //counter set to 10
j=counter--;         //j set to 10
i=counter;           //i set to 9
```

Example 30: Postfix increment and decrement

The first line of code assigns a value of 0 to the variable `counter`. The second line assigns the value of an increment expression to `j`. Because we used the *postfix* increment operator, the expression returns a value of 0 which is the current value of the `counter` variable. `counter` is then incremented by 1. The postfix decrement operator in the fifth line forces the expression to return the current value of `counter` and *then* decrements `counter` by one. This

sets `j` to 10 as it is assigned before the decrement takes place and sets `i` to 9 as it is assigned after the decrement takes place.

⚙ **Prefix performs an increment or decrement and *then* returns a result**

When you use prefix increment and decrement operators, the increment or decrement operation is performed first and *then* the new value is returned.

```
counter=0;           //counter set to 0
j=++counter;         //j set to 1
i=counter;           //i set to 1

counter=10;          //counter set to 10
j=--counter;         //j set to 9
i=counter;           //i set to 9
```

Example 31: Using prefix increment and decrement

With the prefix notation, the second line of code sets `j` to 1 instead of 0. The increment operation is performed first, then `j` is assigned the new value of `counter`.

NOTE

If you are using increment and/or decrement operators in a complex expression you should carefully document their use. Side effects caused by increment and decrement operators can make reading and debugging code extremely difficult.

8.4 Assignment Operators

The basic assignment operator, `=`, assigns the value of its right hand expression to the identifier on its left hand side. C also provides specialised assignment operators.

Many programs include statements such as `total = total + subTotal;`. Programmers constantly perform operations upon a variable's value and then reassign a new value to that variable.

This simple type of calculation is so prevalent in programming that the authors of C decided to provide a class of operators to act as short cuts. You can combine any arithmetic or bitwise operator with an assignment operator.

```
total += subTotal; //same as total = total + subTotal
cost *= tax; //same as cost = cost * tax
```

Example 32: Variations on the assignment statement

The assignment operator in the first line of **Example 32** takes the value of its right hand expression, the value of `subTotal`, and increments its left hand identifier by that amount. The expression in the second line multiplies the right hand expression by the left hand identifier's value, and then reassigns the new result to the left hand side.

NOTE

Short cut assignments can be obscure and difficult to follow for anyone else reading your program code. Make sure to insert comments to explain the use of the statement.

8.5 Comparison Operators

Most programs depend on the ability to compare values. Are two values equal? Does a variable have a positive value? Are two expressions true? All these questions are typically posed in computer programs. C provides three sets of operators you can use to test and return the truth value of an expression: equality operators, relational operators and logical operators.

8.5.1 Expressing True and False

Any expression which returns a value of 0 is considered false while an expression returning any other value is considered true.

C operators which test whether an expression is true or false will return a 1 to indicate a *true* result and a 0 to indicate a *false* result. It is very useful to define symbolic constants for `TRUE` and `FALSE` in order to improve code readability and portability. These definitions often appear as follows:

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

Example 33: Defining constant values for true and false

The following is a more useful definition of `TRUE` and `FALSE` as it defines them in terms of what they represent instead of associating them with a value. For example, `TRUE` is not defined as 1 but as whatever the compiler uses to represent the truth of `(0==0)`.

```
#define TRUE (0==0)
#define FALSE (0!=0)
```

Example 34: Defining constant values for true and false in a portable way

8.5.2 The Equality Operators

The `==` operator returns 1 if its two binding expressions are identical in value. In the example: `(PortA.1 == 1)` assume that `PortA.1` represents the value of bit one in the port defined as `PortA`. The expression returns 1 if `PortA.1` has the value 1 and 0 if it does not.

NOTE

Do not confuse the `==` equality operator with the `=` assignment operator!

`PortA.1==1` tests bit 1 of Port A to see if it is set while `PortA.1=1` sets bit 1 to 1.

The equality operator is often used as part of a statement which controls the execution of a loop or a conditional action.

```
while (PortA.1 == 1) {
    // statements
}

if (counter == 10){
    // statements
}
```

Example 35: Using the equality operator in control structures

The `!=` operator returns 1 if its binding expressions are *not* identical in value. In the following example, the expression returns 1 if bit 0 of `PortA` is not cleared.

```
while (PortA.0 != 0){
    // statements
}
```

Example 36: The inequality operator

8.5.3 Relational Operators

Relational operators return 1 when they correctly express the relative values of their binding expressions.

The less-than operator, `<`, returns 1 if the left hand side expression's value is less than the right hand side expression's value. The expression `(2<3)` returns TRUE while the expression `(3<2)` returns 0.

The greater-than operator, `>`, returns 1 if the left hand side expression is greater than the right hand side expression. Therefore, the expression `(2>3)` returns a value of 0 while the expression `(3>2)` returns a value of 1.

Both the less-than and greater-than operators have an “or equal to” version. Both the less-than-or-equal, `<=`, and greater-than-or-equal, `>=`, operators return 1 if their left hand sides are equal to their right hand side. For example, `(3<=3)` returns 1 while `(3<3)` returns 0.

8.5.4 Logical Operators

The unary logical NOT operator, `!`, returns 1 if its binding expression's value is 0; otherwise, it returns 0.

The binary logical AND operator, `&&`, returns 1 if both of its binding expressions return non-zero values; otherwise, it returns 0. Consider the following example:

```
#define size 30
int i=0;
char s[size];
void main() {
    for (i=0; (s[i]!=0) && (i<size); i++)
        putchar(s[i]);
}
```

Example 37: Logical NOT and AND operators

In **Example 37** the conditional statement for the for loop is `(s[i]!=0) && (i<size)`. This expression returns 1 if `s[i]` is not equal to 0 *and* `i` is less than `size`. If either `s[i]` is equal to 0 or `i` is greater than or equal to `size` then the expression evaluates to 0.

The binary logical OR operator, `||`, returns 1 if either one of its binding expressions returns 1. The logical OR expression is only false when *both* binding expressions have zero values. Notice that the logical OR is not exclusive – that is, if both binding expressions return 1, a logical OR expression will still return 1. Consider the following example.

```
PortA.0, PortB.0=0;
PortA.1, PortB.1=1;
```

```
(PortA.0 || PortA.1);    // returns 1
(PortB.1 || PortA.1);    // returns 1
(PortB.0 || PortA.0);    // returns 0
```

Example 38: Using the or operator

C permits short circuiting of logical expressions

C is able to perform short-circuit evaluations of expressions which use logical operators. When a program runs the computer will only evaluate as much of a logical expression as is necessary to determine if the expression has the value 1 or 0.¹⁰ For example:

```
PortA.0, PortB.0=0;
PortA.1, PortB.1=1;
if ( PortA.0 && ((PortA.1 || PortB.0) && (PortB.1 || PortA.1))) {
    // statements
}
```

Example 39: Sort circuit expression evaluation

While the logical expression appears complex, it is actually quite simple. In fact, a good compiler will flag the entire `if` structure as unreachable or *dead* code. Why? Because of C's short-circuit evaluation ability. As soon as the computer begins evaluating the logical expression, it determines that `PortA.0` has a 0 value. It knows that this will make the outermost logical AND expression false; therefore, it does not evaluate the any more of the expression.

Careful design can exploit short-circuit evaluation. For example, you might want to avoid calling the function to read a key from the keyboard buffer if no key has been pressed. The following construct shows how you can use short-circuiting of logical expressions to achieve this:

```
if ( (keyPressed() == TRUE) && ((myKey = getch()) == 0) ) {
    // special key has been pressed
    specialKey = TRUE;
    myKey = getch();
}
```

Example 40: Using short-circuit evaluation

If the `keyPressed()` function returns 0, a key has *not* been pressed and the logical expression will short circuit and avoid the call to `getch()` – the compiler knows that if any term in a logical AND expression is *false* the entire expression is *false*.

¹⁰ Some compilers allow you to force *full evaluation* of logical expressions.

Notice that the second term in the logical AND expression serves two purposes. When `keyPressed()` returns 1 a key has been pressed, the second term gets a value for `myKey` and decides whether the pressed key was a special key (the special character NUL) or not. If `getch()` returns 0 then we make another call to `getch()` to retrieve the identity of the special key.

8.6 Bit Level Operators

Bit level or bitwise operators are operators which evaluate and manipulate data at the bit level. These operators are especially useful to embedded system programmers. They fall into two main classes: logical operators and shift operators.

8.6.1 Bit Logical Operators

C supports one unary and three binary bitwise logical operators. Each of these operators act only upon values stored in the `char`, `short int`, `int` and `long int` data types.

NOTE

Binary logical operators perform data promotion on operands to ensure both are of equivalent size. If you specify one `short` operand and one `long` operand, the compiler will *widen* the `short` to occupy the `long` 16 bits. This expression will return its value as a 16 bit integer.

Bitwise AND Operation

The bitwise AND operator, `&`, produces a bit level logical AND for each pair of bits in its operands. For example, if both operands have bit 0 set then the result of the bitwise AND expression has bit 0 set.

```
int x=5, y=7, z; // 5 is binary 101 and 7 is binary 111
z = x & y;        // z gets the value 5 (binary 101)
```

Example 41: Bitwise AND operation using `&`

The AND operation is easier to see if your compiler has an extension which permits data values in binary:

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x & y;           // z gets the value 00000101, or 5
```

Example 42: Using the AND bitwise operator with binary values

The resulting value for *z* has a bit set in every position where both *x* and *y* have a bit set, and bits cleared in every other position.

NOTE

The bitwise AND, `&`, is not the same operation as the logical AND, `&&`.

OR Operations with Bits

The bitwise OR operator, `|`, performs a bit level logical OR for each pair of bits in its operands. If either operand has a bit in a specific position set, then the result of the bitwise OR expression has that bit set. To return to our previous example:

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x | y;           // z gets the value 00000111, or 7
```

Example 43: Using the bitwise OR operator |

The value for *z* has a bit set in every position where either *x* *or* *y* have a bit set, and bits unset in every other position. This produces a result with all the bits that either operand has set.

NOTE

The bitwise OR, `|`, is not the same operation as the logical OR, `||`.

XOR Operations with Bits

The bitwise XOR operator, `^`, produces a bit level logical exclusive OR for each pair of bits in the operand. Slightly different than OR, the XOR sets a bit when one of the operands has a bit set in that position but not if both operands have the bit set. This produces a result with bits set that the operands do not share:

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x ^ y;           // z gets the value 00000010, or 2
```

Example 44: The bitwise XOR operator

NOT Operations with Bits

The bitwise NOT operator, `~`, produces the complement of a binary value. Each bit that was set in the operand is cleared and each cleared bit is set:

```
int x=0b00000101,
    z;
z = ~x;           // z gets the value 00000101, or 5
```

Example 45: The bitwise NOT operator

8.6.2 Bit shift operators

Both operands of a bit shift operator must be integer values.

Shift Right

The right shift operator shifts the data right by the specified number of positions. Bits shifted out the right side disappear. With unsigned integer values 0s are shifted in at the high end as necessary. For signed types the values shifted in is machine dependant. The binary number is shifted right by number bits: `x >> number;`. Right shifting a binary number by n places is the same as an integer division by 2^n .

```
porta = 0b10000000;
while (porta.7 != 1){
    porta >> 1;
}
while (porta.0 != 1){
    porta << 1;
}
```

Example 46: Shifting bits left and right

Shift Left

The left shift operator shifts the data right by the specified number of positions. Bits shifted out the left side disappear and new bits coming in are zeroes. The binary number is shifted left by number bits: `x << number;`. Left shifting a binary number is equivalent to multiplying it by 2^n .

9. Control Structures

One of the most important features of any programming language is the ability to control the way in which program statements are executed. Normally, a computer executes all the statements in your program sequentially. It will start at the first statement in the `main()` function and execute each statement and function call until it finishes executing the last statement in `main()`.

Sometimes you want the computer to deviate from sequential execution. Control structures allow the making of decisions about which instructions to execute. You can also use control structures to repeat a set of instructions.

The C language contains a variety of powerful and flexible¹¹ control structures. In general, control structures fall into two groups – those that branch and those that loop.

9.1 Conditional Expressions

All C expressions return a numerical value. For example, the expression $(2 + 3)$ returns the value 5. A control structure tests the value of a particular expression at run-time and makes a decision about how to proceed.

Consider the expression $(2 > 3)$. This expression is false as it asserts that 2 is greater than 3. Since we know that *all* C expressions return a value we know that $(2 > 3)$ must return a value. In C, any expression which evaluates to be false returns a value of 0 while an expression which evaluates to any other value returns a value of 1. Thus, we can see that the false expression $(2 > 3)$ will return a value of 0.

The representation of true and false is an important concept to keep in mind as you program in C. Many C programmers use control structures to test expressions with absolutely no logical operators in them. This may seem foreign to many programmers familiar with other languages. For example, many C programmers use a loop construct such as:

¹¹ One problem stems from the overburdening of control structures with statements which generate *side effects*.

```
testVariable = 1;
while (testVariable) {
    // some statements
}
```

Example 47: Controlling loops without using logical operators

As long as the `testVariable` retains the value of 1 the loop will continue. At some point a statement inside the loop might set the variable's value to 0, causing the loop to terminate before the next cycle.

9.2 Decision Structures

C provides two structures the programmer can use to support different types of decisions. Decision structures test an expression to determine which statement or statement block to execute.

9.2.1 if and else Statements

if

The `if` structure specifies a specific execution path based on the value of a particular expression. The following example shows the general form for this structure:

```
if (expression) {
    // if expression is true do these statements
}
```

Example 48: if and else structure

Notice that the `if` statement is *not* followed by a semicolon. This is because an `if` statement is not a complete statement by itself. The `if` structure requires a statement or statement block to complement it. The complementing statement or statement block provides the terminating semicolon. All the following examples are syntactically correct:

```
if (a) b=c;
if (a)
    b=c;
if (a) {
    b=c;
    d=e;
}
```

Example 49: Using the if statement structure

When the expression evaluates to 1, the `if` structure's complementing statement or statement block is executed. If the expression evaluates to 0, the complementing statement or statement block is ignored and the statement directly after the `if` structure is executed.

else

There is an additional optional component of the `if` structure which executes a set of statements when the `if` tested expression is false. This additional component is the `else` structure.

The `else` structure must be the first statement following an `if` structure. When the `if` condition evaluates to 0, execution will pass directly to the `else` structure.

```
if (expression) {  
    // if expression is true execute these statements  
}  
else {  
    // if expression is false execute these statements  
}
```

Example 50: The else statement

Like the `if`, `else` needs a complementing statement or statement block which provides its semicolon terminator. Unlike the `if`, `else` does not test the value of an expression.

9.2.2 Nested if statements

It is possible to place one `if` structure inside another `if` structure. Consider the following example.

```
if (PortA_DDR.0 != 1) {  
    if (PortA_DDR.1 != 1) {  
        PortA.0 = 1;  
        PortA.1 = 1;  
    }  
}
```

Example 51: Nesting if statements

If the value of `PortA_DDR` bit 0 is not 1 and the value of `PortA_DDR` bit 1 is not 1 then `PortA` bit 0 and `PortA` bit 1 are set to 1. You could encapsulate the example into a single logical expression such as:

```
if ( (PortA_DDR.0 != 1) && (PortA_DDR.1 != 1) )
```

Example 52: Converting nested if statements to logical expressions

9.2.3 Matching else and if

An `else` always matches with the nearest unmatched `if`. A common problem with `if...else` structures arises from a set of statements such as:

```
if (a)
    b=1;
    if (!a)
        b=2;
else
    b=3;
```

Example 53: Matching if and else statements

If `a` has the value 1, what value will `b` have after these statements? What if `a` has the value of 0? Will `b` ever have the value of 1 after these statements? The answer to all these questions depends entirely upon a syntactic question – which `if` statement does the `else` belong to? In C an `else` structure always belongs to the nearest `if` not already associated with an `else`.

Using this rule, we can see that the `else` associates with the second `if`, not the first. Therefore, if `a` has the value 1, `b` is first given the value 1 by the statement `if (a) b=1;`. When the next `if` statement is evaluated `b` is given the value 3 because the conditional statement is `!a` which evaluates to `!1` or 0. The `else` statement is executed and `b` is assigned the value 3.

Good programmers include braces around the complementing statements of `if` and `else` structures in order to make code easier to read, debug and modify. Applying this principal to the previous example makes the situation much more obvious.

```
if (a) {
    b= 1;
    if (!a) {
        b=2;
    }
    else {
        b=3;
    }
}
```

Example 54: Using braces to clarify the combination of if and else

Some programmers like to make the situation clearer by placing an `else` on the same line as the end bracket of its matching `if`:

```
if (!a) {  
    b=2;  
} else {  
    b=3;  
}
```

Example 55: An alternate format for showing if else pairing

9.2.4 switch and case

`if..else` structures let you make a decision between two paths based on the truth value of a single expression. You can use a series of nested `if` statements to test a variable for a series of possible values, but C includes a statement which tests many possible variable values: the `switch-case` structure. This structure lets you switch between several different possible paths of code to execute.

The `switch-case` structure has a `switch` value or expression upon which the branching of code execution is based. Statement execution depends upon the different cases provided for possible values of the `switch`. The general format for the `switch-case` structure looks like:

```
switch (expression) {  
    case possibleValue :  
        statement;  
        statement;  
        break;  
    case anotherPossibleValue :  
        statement;  
        break;  
}
```

Example 56: The switch..case structure

Like the `if` statement, `switch` is not a statement on its own – it requires a complementing statement block. Unlike the `if` structure, the `switch` statement complementing statement block must have a specific form, consisting of a series of possible cases for the `switch` expression.

9.2.5 Execution within a switch

Each value for the `switch` expression is preceded by the keyword `case`, and followed by a colon. When the `switch` is executed each case is tested in turn. If a case value does not match the evaluated value of the `switch` expression, all code is ignored until the next `case` statement is encountered or the end of the `switch` block is reached. If a case value matches the

switch value, execution begins with the statement following the matching case.

9.2.6 Fall-through execution

Once a case value matches the switch expression every subsequent line is executed, including those after subsequent case statements. Most of the time this is not the desired action; you want the computer to execute the code for only one case. To avoid the “fall through” effect of the C switch-case structure, you must place a break statement at the end of each case.

Sometimes you may wish to take advantage of the “fall through effect”. Consider the following simple example which enables a decimal point if specific digits are being displayed:

```
switch (digit) {  
    case 1:  
        addpt = 0;  
    case 2:  
    case 4:  
        addpt = 0x80;  
        break;  
    case 5:  
        addpt = 0;  
        break;  
}
```

Example 57: Using the fall-through effect with switch statements

Notice that if the second or fourth digit is being displayed addpt is set to 0x80. This variable addpt is a flag which allows the display of a decimal point which delineates between minutes, seconds and fractions of a second. The “fall-through” effect is used with case 2 : where it falls through to case 4 : . If the first or fifth digit is being displayed addpt is set to 0 indicating that no decimal point is displayed.

NOTE

Notice that there is a break statement after the 5 case value, even though this is not compulsory. It is good programming practice which helps in the event that you modify the structure by adding additional cases. The existing break can help prevent debugging problems.

The multiple case enhancement available with some C compilers allows a clearer form of this example:

```
switch (digit) {  
    case 1, 5:  
        addpt=0;  
        break;  
    case 2, 4:  
        addpt = 0x80;  
        break;  
}
```

Example 58: Multiple case enhancement

9.2.7 The default case

Another useful feature of the switch-case structure is the option to provide a default case value. A default case is automatically considered a match with the switch expression value regardless of what that value actually is. This means that the default case should always be placed last in a switch-case structure, otherwise it will match before the switch expression can be tested against any other possible values. The following example shows the use of the default case value.

```
switch (digit) {  
    case 2:  
    case 4:  
        addpt = 0x80;  
        break;  
    default :  
        addpt = 0;  
}
```

Example 59: Using the default case value

9.2.8 The goto Statement

Many C programmers have strong objections to the use of the goto statement. The goto remains a holdover from early programming languages without sophisticated control flow. Because C provides a variety of useful control structures, you should not need to use goto statements.

If you do use a goto statement, be extremely careful and document it well. Consider the following example:

```
void main(void){  
    if (time < limit)  
        time++;  
}
```

```
    else
        goto Done;
Done:
}
```

Example 60: The goto statement

NOTE

Make sure that when you use a `goto` statement you document where the target is. This can help prevent later debugging problems. As a general rule, you should write code which uses some control flow method other than a `goto`.

9.2.9 Comparing goto and switch..case

You may have noticed a similarity between the `goto` statement and the `switch..case` statement. This similarity in form comes with a similarity in function. The `switch-case` behaves like a `goto` or jump table where each case is a label.

NOTE

It is essential to remember that the `switch-case` operates like a jump table. The fall-through effect of `case` statements can be useful, but a source of debugging problems if `break` statements are not used properly.

9.3 Looping Structures

C control structures allow you to make a decision on the path of code execution. C also provides looping structures for control over program flow. Loop control structures allow you to repeat a set of statements.

9.3.1 Control expression

The key component of any loop structure is the control expression. At some point in each iteration, the control expression is tested. If the control expression evaluates to 0 program execution passes to the first statement following the loop structure. If the expression evaluates to 1, execution continues within the loop structure statement block.

NOTE

The only control you have over loop structures is the control expression. The compiler cannot tell you if a loop has a control expression which will never evaluate to 0. In embedded systems programming, infinite loops are often used to keep the program running constantly.

9.3.2 The while loop

The simplest C loop structure is the `while` loop. Here is the general form of the `while` loop:

```
while (controlExpression) {  
    // statement block  
}
```

Example 61: The while loop syntax

In a `while` loop the control expression is at the top of the structure. The `while` loop evaluates the control expression before every loop iteration – including the first loop iteration. Therefore, if a control expression evaluates to 0 the first time the `while` loop is encountered, the statements inside the structure will never execute.

9.3.3 The do loop

The `do` loop tests the control expression value *after* every loop iteration. The general form of a `do` loop is as follows:

```
do {  
    // statement block  
} while (controlExpression); // close do
```

Example 62: The do loop syntax

Because the `do` loop tests the control expression *after* every iteration of the loop the statement block will always execute at least once, even if the control expression evaluates to 0 when the loop is first entered.

NOTE

The `do` loop is one of the few cases where keywords belong at the *end* of a statement block. Because of this, you should place the `while` expression on the same line as the loop's closing brace and put a comment after the `while` explaining that it closes a `do` structure.

9.3.4 The for loop

The most complex and flexible looping structure available in C is the `for` loop. The `for` loop incorporates statements which alter variables used in the control expression. The example on the left shows a `while` loop and that on the right shows the equivalent `for` loop:

while loop	for loop
<pre>counter=0; while (counter<=10){ //statements counter ++; }</pre>	<pre>for (counter=0;counter<=10;counter++){ //statements }</pre>

Example 63: Comparing the while and for loops

A `for` loop executes statements a predetermined number of times. The control expression for the loop is initialized, tested and manipulated entirely within the `for` loop parentheses. It is easy to debug the looping behaviour of the structure as it is independent of the activity inside the loop.

9.3.5 How the for loop works

Each `for` loop has up to three expressions which determine its operation. The following example shows general `for` loop syntax. Notice that the three expressions in the `for` loop argument parentheses are separated with semicolons.

```
for ( initialize; control; increment) {
    // statement block
}
```

Example 64: Using the for loop

The first expression, *initialize*;, provides initial values for variables used in the control expression. When the `for` loop is first encountered this initialization expression is executed.

The second expression, *control*;, is the same as the control expression used in the `while` and `do` loops. Like the `while` loop, the `for` loop control expression is checked *before* each loop iteration. If the control expression evaluates to 1, the loop statement block is executed; otherwise, execution passes to the first statement following the loop. In **Example 63**, the control expression tests to see if `counter` is less than or equal to 10. As long as the expression returns 1, the loop will iterate.

The third increment expression, *increment*, is used to modify value(s) in the control expression. The increment expression is executed after each loop iteration. Execution then jumps to the beginning of the loop and the control expression is tested.

NOTE

You can omit any of the `for` loop expressions, but you must include the semicolon separators so the compiler knows which expressions have been left out. If the *control* expression is omitted the `for` loop will not stop.

9.4 Exiting a Loop

C provides two ways to escape a looping structure: the `break` and `continue` statements. When either of these statements is encountered inside a loop any remaining statements inside the loop are ignored.

9.4.1 The `break` Statement

Use a `break` statement to completely break out of a loop. The most common place for a `break` statement is inside a `switch-case` structure. However, this is not the only place it can be used. You can also use a `break` statement to break out of any looping structure in C. When a `break` is encountered inside a looping structure, the loop terminates immediately and execution passes to the statement following the loop.

9.4.2 The `continue` Statement

You may wish to jump to the next iteration of a loop without breaking out of the loop entirely. A `continue` statement will allow you to do this. When a `continue` statement is encountered inside a looping structure, execution passes immediately to the *end* of the loop statement block. Because execution passes to the end of the loop statement block, the next action is the evaluation of the loop control expression.

If `continue` is used with a `while` or `for` loop, execution jumps from the end of the statement block to the control expression at the top of the loop. If used with a `do` loop, execution passes from the end of the statement block to the control expression at the bottom of the loop. In all cases, the effect is the

same – a `continue` statement does not circumvent the loop control expression, it ignores any statements remaining in the loop iteration.

10. Functions

Functions are the basic building blocks for all C programs.

There are some restrictions for the creation of C functions. Each function in a C program must be self-contained. You may not define a function within another function. Also, you may not extend the definition for a function across more than one file – when you define a function it must be contained within a single file.

10.1 `main()`

Every C program has at least one function called `main()`. When the target computer runs your program, program execution generally begins with the first statement of the `main()` function.

In reality, program execution usually begins with initialization code quietly linked into the program. The C compiler may generate this automatically, based on information contained within the C program, or it may link in a standard library. But the compiler cannot know the entire state of a target embedded system before invoking `main()`. In a desktop system, the OS itself covers most of the hardware details. In an embedded system without an OS, you may be obliged to write initialization code to establish the running state of the MCU before transferring control to `main()`.

There is little to stop you from performing such initialization within `main()` itself.

10.2 Executing a Function

Any function in a C program can execute, or *call*, any other function. Typically, the `main()` function calls one or more other functions which may in turn call other functions. There is a restriction on the calling of functions: a function cannot call a function which it does not recognize. There are two different techniques for allowing a function to be recognised.

- ① Provide the full definition for a function *before* the part of the program which calls it. This method has the following complications:
 - 1) C lets you combine functions from several files into a single program – how would you alert your program to functions found in another file?
 - 2) It is possible to have two functions call each other – which of these functions would you define first?
- ② Use a *function prototype* to alert the compiler about a function *before* you actually provide its definition. This method has several advantages which are explained in the following section.

10.2.1 Calling a Function

The syntax for a function call in C is the function name and a list of parameters surrounded by parentheses. When the C compiler encounters an identifier followed by a left parenthesis it knows that the identifier represents a function. For this reason, function names always include a pair of parentheses, for example `main()`.

If you have defined a function called `sum()` to add two integers and return the result you can assign the return value of `sum()` to a variable with the following line:

```
sumResult = sum(firstNum, secondNum);
```

Notice that the function call to `sum()` fits into an assignment expression in the same way as a variable or variable expression. You can place a call to a function any place an expression can occur. For example:

```
areaRectangle = height * sum(length, width);
```

You can include expressions in the parameter list in place of variable names as long as the expressions evaluate to an appropriate data type. For example, a formula to calculate the hypotenuse of a right-angled triangle could look like this:

```
hypotenuse = sum((sideOne*sideOne), (sideTwo*sideTwo));
```

Notice that there are parentheses around each expression in the parameter list. This is for the sake of clarity but it is not required because of the extremely low precedence of the comma operator which separates elements in the parameter list.

10.3 Function Prototype Declarations

Function prototype declarations ensure that your program knows about a function. Like variable declarations, function prototype declarations let the compiler know about the function names before the function is used. You may not include a function prototype inside the definition of a function.

10.3.1 Defining the Function Interface

Function prototypes allow you to fully define the interface to a function without worrying about its contents. This concept is referred to as data abstraction. A function interface contains:

- 1) Data type returned by the function
- 2) Function name
- 3) Data types of function parameters

Once a function prototype defines an interface, the compiler can check calls to the function. Do the calls use the right number and type of parameters? Does your program treat the functions return value appropriately for its type?

10.3.2 Calling Functions in Other Files

Function prototypes let you take advantage of functions in other files, even if the files have already been compiled. Pre-compiled files of functions are called **object libraries** and most C development environments make extensive use of them. The prototypes for functions in a pre-compiled library are often contained in a **header file**. This allows you to take advantage of pre-compiled functions without having to worry about compiling or maintaining them.

For example, traditional C development environments provide standard library functions to handle user input and output. A header file called `stdio.h` contains the prototypes for these functions. Since very few 8 bit platforms provide resources for user input and output, these library functions are not typically needed. 8 bit microcontroller libraries are often for such things as A/D, serial and peripheral support. For example, the library `lcd8.h` contains functions which write data to the LCD, control the LCD and initialize the LCD. If you use any functions from the LCD library, you must include the appropriate header file in your program:

```
#include <lcd8.h>
```

The angle brackets in the `#include` directive instruct the preprocessor to search for the header file in the location reserved for library header files. The directory or path searched is dependent on the compiler and operating system.

10.3.3 Function Type, Name and Parameter List

There are similarities between a function prototype and a variable declaration. Consider the following example.

```
int counter;      //variable declaration
int sum(int numOne, int numTwo); // function prototype
```

Example 65: Comparing function and variable declarations

The first element in the function prototype is a data type. This tells the compiler the data type of the function's return value. The type of the return value informs the compiler how much memory to allocate in RAM to hold that value. It also ensures that you use the function properly in expressions elsewhere in the program. In this case, you can put a call to `sum()` in any expression where an `int` value could occur.

After the function data type comes the name of the function. This identifier is entered in the symbol table and associated with an address which contains the beginning of the function's executable code. When your program calls a function, execution jumps to the address associated with the function name.

Parentheses following an identifier inform the compiler that you are declaring a function, not a variable. You must include the parentheses, even if a function accepts no parameters. If there are function parameters each one should include a data type and a meaningful name. It is only necessary to include the data type of each parameter in a function prototype declaration. For example:

```
int sum(int, int);
```

However, this form is unclear. Including meaningful names for each parameter increases program readability. It also helps to understand the order in which a function reads parameters. For example, suppose you encounter the function prototype:

```
int portControl(int, int);
```

This prototype is ambiguous – presumably each parameter holds an integer value, but these could be used in very different ways. For example, suppose this function controls a port – the first parameter could specify the port address and

the second parameter the data direction values. A prototype like the following is much clearer:

```
int portControl(int portLoc, int DDR);
```

Function prototype names need not be those used in the function definition, but using the same names helps to avoid confusion.

10.3.4 Functions and void

Some functions accept no parameters or return no value¹². For example, you could create a function prototype such as:

```
wait();
```

When most compilers encounter such a declaration, they assume that the function will return an `int` value. In embedded systems this practice wastes memory resources because the space for the `int` is reserved. To avoid this problem use the `void` keyword:

```
void wait();
```

The `void` keyword tells the compiler explicitly that the function will not return a value so no memory is allocated for a return value. You can also use `void` inside the parentheses of a function prototype to explicitly declare that the function accepts no parameters:

```
void wait(void);
```

Notice that `void` applies to function definitions as well as the declaration of function prototypes. You will see programs that begin like this:

```
int main(void)
```

It is best to include the `void` keyword whenever you have a function without a return value or parameters. This clarifies the purpose of your functions.

¹² Some C programmers insist that functions which just produce side effects should return a value to indicate success, failure or error. Since memory is at a premium in 8 bit development, embedded developers see this practice as a luxury.

10.4 Function Definitions

A function prototype describes the interface to a function while a function definition describes the function interface *and* contents. The definition includes the statements that will execute the function is called. When the compiler reaches the function definition, it reserves enough program memory to hold the statements in the function and stores the address of the first statement with the function name.

10.4.1 Statement Block

A function definition includes a statement block which contains all function statements. A statement block is a group of one or more statements enclosed in braces { }. Even if a function has only a single executable statement it must be enclosed in a statement block. For example, consider the following simple function which returns the sum of two integers passed as parameters:

```
int sumInt(int firstNumber, secondNumber) {  
    return(firstNumber + secondNumber);  
}
```

Example 66: The function statement block

10.4.2 Variable Declarations in Function Definitions

A function statement block can contain any number of variable declarations¹³. You may declare a variable anywhere in your function, as long as it is declared *before* it is used. Code is generally more readable if you declare variables at the top of the function block:

```
int sumInt(int firstNumber, secondNumber) {  
    int sumTotal;    // local variable holds sum  
    sumTotal = firstNumber + secondNumber;  
    return(sumTotal);  
}
```

Example 67: Variable declarations inside functions

¹³ Any statement block can include new variable declarations.

10.5 Function Parameters

Most functions required information from the code that calls them. The most common way to pass information to a function is through its list of parameters. You can also pass information to a function using of global variables – any variable in global program space can be used by any program function. It is good programming practice to avoid the use of global variables if possible.

10.5.1 Passing Data by Value

When you call a function, parameter values are passed to the function. The compiler will set aside the appropriate amount of memory to hold these values. This is why it is important to specify function parameter data types in the function prototype. The following code clarifies this.

```
void change(int num) {
    num = 4;
}
void main(void) {
    int val = 2;
    change(val); //send value of val to change()
    val += val;  // val = 2 + 2 = 4
}
```

Example 68: Passing data to a function by value

What value will `val` have after last line in `main()`? The answer is 4, not 8. When `main()` calls `change()`, the value of `val` is passed to the function, not its address. The function stores the value in the memory location reserved for its parameter, `num`. The value at this memory location is changed by the function. `change()` but the change has no effect on the value stored in `val` because `val`'s address is not known. This method of parameter handling is called passing parameters by value.

10.5.2 Passing Data by Reference

How can you write a function which can change variables belonging to its calling function? A variable value can be changed by accessing the variable's address to change its value. Variables are accessed by their addresses using pointers. A pointer is a data type which stores an address. A pointer can be used like any other data type, therefore you can write a function which accepts a pointer as a parameter. The following is another version of the example from the previous section.

```
void change(int *num) { //pointer to an int value
    *num = 4;           // place 4 at address in *num
}
void main(void) {
    int val = 2;

    change(&val); //pass the address of val
    val += val;
}
```

Example 69: Passing a variable to a function by address (reference)

In this example, `val` will have a value of 8 after the last line in `main()`.

The definition of `change()` includes a pointer to an integer parameter, instead of the integer parameter itself. When `main()` calls `change` the function creates a copy of `val`'s *address* in memory, not its value. The assignment performed by the function uses the dereference operator, `*`. Instead of assigning the value 4 to `num`, the dereference operator assigns 4 to the memory location corresponding to `val`'s address which is stored in the pointer `num`. The dereference operator reads the value of its binding identifier as an address and then represents the value stored at that address.

NOTE

Notice that in the call to `change()` you specify the address of `val` with the unary *address operator* `&`. The address operator returns the address in memory which stores the value of its binding identifier.

10.5.3 Functions Without Parameters

Most programs have at least one function which accepts no parameters – typically `main()`. A function with no parameters can be declared it with an empty parameter list.

```
int myFunc()
```

However, it is good practice to specify that the function has no parameters with the `void` parameter type:

```
int myFunc(void)
```

Functions with no parameters create side effects. A program expects calls to functions to return values. Anything a function might do besides return a value is considered a side effect. Side effects can be important and quite useful;

however, you should be careful when including many functions which produce side effects.

Functions which produce extensive side effects are harder to maintain and debug, especially for members of a development team. To safely use abstract functions, you only need to know the data which goes in and comes out – the function interface. When a function produces side effects, you need to know about the interface *and* behaviour to use it safely.

11. Complex Data Types

This section introduces several complex C data types. Complex data types include pointer, arrays, enumerated types, unions, and structures. A solid understanding of pointers and arrays in particular is absolutely vital to an effective use of the C language.

11.1 Pointers

The elementary C data types, `char`, `int` and `float`, store values which are used directly. Unlike these basic types, the pointer data type represents values used indirectly.

All data stored in computer memory is stored as a series of ones and zeroes. C data types act as filters which interpret these ones and zeroes. When the computer evaluates a pointer value, it reads the ones and zeroes as a memory address. Consider computer memory a single long street and each block of memory as a building, then a pointer contains a number which identifies a specific building on the street.

A pointer value can be interpreted as a number just as a real address could. Because of the pointer's special nature, the computer knows to interpret that number as an address in memory.

NOTE

Pointers can be difficult to understand. A pointer contains a numeric value, the difference is in the way the value is *interpreted*: as an address in memory.

11.1.1 Declaring a Pointer

The declaration of a pointer data type must specify the type of data it can point to. Consider the following statement which declares a pointer able to point at any data of type `int`: `int * myIntPtr;`

When you declare a pointer, the compiler assigns it the value `NULL` – this signifies that it points to no valid address.

A pointer's data type is important. The computer uses the data type to determine the size of the memory block the pointer points to. For example, on

8 bit microcontrollers an `int` takes up memory in 8 bit blocks. Therefore, `myIntPtr` from the previous example points to a block of memory 8 bits in size.

11.1.2 Pointer Operators

To help manipulate pointers, C provides two, unary operators: the **address-of**, `&`, operator and the **dereference**, `*`, operator. The dereference operator is used with any pointer and the address operator with any type of data.

The Address Operator

The address operator, `&`, is a prefix unary operator. It binds with the identifier to its immediate right and returns the memory address of its bound identifier. Typically, the address operator is used to assign the address of a variable to a pointer or to pass the address of a variable to a function. Consider the following code example.

```
int * myIntPtr;    // myIntPtr is a pointer to an integer
int ** myPtrPtr;   // myPtrPtr is a pointer to a pointer
int myInt = 2;
// assign address of myInt to myIntPtr
myIntPtr = &myInt;
// assigns address of myIntPtr to myPtrPtr
myPtrPtr = &myIntPtr;
```

Example 70: Using the address of operator

Memory Space	Address	Type	Name	Type Value
00000010	0x00	int	myInt	The integer 2
00000000	0x01	pointer to int	myIntPtr	The address 0x00
00000001	0x02	pointer to pointer	myPtrPtr	The address 0x01

Table 9: Pointers and pointers-to-pointers

It is possible to use the address operator with a pointer. In these cases, the address operator returns the memory address where the pointer's value is stored. This double indirection is described as a pointer to a pointer which is also called a **handle**.

The Dereference Operator

The dereference operator, `*`, is a prefix unary operator. It interprets the value of its bound identifier as a memory address and returns the value stored at that location. For example, all of the following equality expressions evaluate to 1:

```
myIntPtr = &myInt; //point to myInt

*myIntPtr == myInt; //dereferenced pointer = integer value
(*myIntPtr += 1) == (myInt += 1);
(*myIntPtr)++ == myInt++;
```

Example 71: Using the pointer dereference operator

Why do these last two logical expressions use parentheses? Because of C's precedence rules.

In the first case the `==` equality operator has a higher precedence than the `+=` assignment operator, so the parentheses ensure that both assignments are performed before the equality evaluation.

In the second case the postfix `++` increment operator has a higher precedence than the `*` dereference operator. To perform the dereference *first*, we need to place parentheses around its sub-expression. Without these parentheses, the increment operator would increment the *pointer* instead of the what the pointer points at! The results of this side effect are not obvious until the *next* time you use `myIntPtr`.

NOTE

It is essential to remember that `*` and `&` are *operators* and that careless use of them can create bugs which are difficult to locate. Always include parentheses and comments to facilitate debugging pointer problems.

11.1.3 Pointer Pitfalls

Dereferencing a pointer set to `NULL` will cause problems. Pointers pointing to `NULL` do not point at a valid memory address and cannot be dereferenced. The following code fragment shows a common mistake made with pointers:

```
int * myIntPtr;
int myInt;

*myIntPtr = myInt // dereference a NULL pointer!
```

Example 72: Dereferencing a pointer set to `NULL`

Remember that assignment operators have lower precedence than the dereference operator. The assignment will not happen until you attempt to dereference the NULL pointer! Before you dereference a pointer, it must have a valid address value. The following fragment shows the proper way to initialize a pointer.

```
int * myIntPtr;  
int myInt;
```

```
myIntPtr = &myInt; // correct way to initialise a pointer
```

Example 73: Initializing a pointer

The address operator has a higher precedence than the assignment operator. The address of `myInt` is returned before the assignment to `myIntPtr`. Notice that we do not need to initialize `myInt` in order to point `myIntPtr` at it. The declaration of `myInt` sets aside a specific memory block for `myInt`.

11.2 Arrays

It is useful to arrange related elements of data in blocks or structures. The fundamental block arrangement is the array¹⁴. When you declare an array, you must declare both an array type and the number of elements it contains. For example, the following declares an array containing 8 `int` elements:

```
int myIntArray[8];
```

When you declare an array a single, contiguous block of memory is reserved to hold it. This is why you must specify the array size. As soon as an array is declared a block of memory large enough to hold all the array elements is allocated.

11.2.1 Accessing Array Elements

The postfix subscript operator, `[ ]`, is used to refer to an array element. The operator binds to an identifier which returns an address in memory. The integer expression inside the square brackets is evaluated and this number determines how many units of memory should be moved past the bound identifier address value.

¹⁴ Another way to arrange related elements of data is with the `struct` data type.

How big is a unit? The size is determined from the data type of the bound identifier expression. For example, if `myIntArray` is an array of `int` then the expression `myIntArray` returns the starting address of the block of memory occupied by the array. The expression `myIntArray[2]` jumps two `int` sized blocks from the address returned by `myIntArray`.

NOTE

When you *declare* an array in C you must specify the number of elements it contains. However, when you *subscript* an array the number in the brackets indicates the number of elements past the first element in the array. The *first* element in a C array is number 0. This is because the notation `myArray[0]` is interpreted as a jump 0 elements past the first element in the `myArray` memory block.

11.2.2 Multidimensional Arrays

A multidimensional array is declared with an array subscript for each dimension. For example a two dimensional array is declared as: `dataType arrayName[dim1][dim2];`

11.2.3 Array Operations and Pointer Arithmetic

Two operations specifically take an array as their argument:

- 1) Deriving the total array size with the `sizeof()` operator
- 2) Deriving the address of the first array element

All access to array data is handled using pointer arithmetic. Consider the following code:

```
int myIntArray[8];
int * myIntPtr;
myIntArray[0] = 5; //first element of array set to 5
myIntArray[1] = 10; //second element of array set to 10

// set myIntPtr to point to first element in myIntArray
myIntPtr = &myIntArray[0];
myIntPtr = myIntArray; //same effect as preceding line
// this equivalency expression is true
*(myIntPtr + 1) == myIntArray[1];
```

Example 74: Array operations and pointer arithmetic

First we declare an array of `int` values and a pointer to an `int`. We then set the pointer, `myIntPtr`, to point to the first element of the array. Notice how to set a pointer to point at an array. You can use the expression `myIntPtr = myIntArray;` because `myIntArray` returns the address of the first array element.

The tricky part of **Example 74** is the last statement. Pointer arithmetic allows us to specify the `int` sized block of memory next to `myIntPtr` with the expression `myIntPtr + 1`. Since we know that arrays are always stored in contiguous blocks of memory, it follows that the `int` sized block of memory next to `myIntArray[0]` must be `myIntArray[1]`.

In general, the expressions `*(myIntPtr + x)` and `myIntArray[x]` are equivalent when `myIntPtr` points to the first member of `myIntArray[]`. Because the subscript square brackets are an operator, the expressions `myIntPtr[x]` and `myIntArray[x]` are also equivalent. The subscript operator checks the underlying type of `myIntPtr` and, finding that it points to an `int`, jumps over `x` `int` sized blocks.

Be careful! The apparent symmetry between pointers and arrays emerges from the way their related operators work. Arrays and pointers are not fundamentally the same. The first two equivalency expressions return 1, but the third may return either 1 or 0:

```
*(myIntPtr + x) == myIntArray[x];
myIntPtr[x] == myIntArray[x];
// this may not be true
*(myIntPtr + x) == (myIntArray + x);
```

Example 75: The relationship between arrays and pointers

Even though the expression `myIntArray` returns an address value it is *not* a pointer. Since `myIntArray` is not a pointer, pointer arithmetic will not always work as expected¹⁵.

11.2.4 Arrays of Pointers

An array can contain pointers to other data types. The most common use for an array of pointers is to use an array of pointers to type `char` which are pointed to strings. This technique can be used to send messages to a screen. In the

¹⁵ For a useful treatment of array-pointer distinctions see Koenig's *C Traps and Pitfalls*.

following example the array is declared in main but the array is passed to a function where the values of the pointers are assigned.

```
void func1(char *p){
    p[0]="Press 1 to start";
    p[1]="Press 2 to continue";
    p[2]="Press 3 to RESET";
    p[3]="Press 4 to quit";
}
void main(void){
    int val;
    char *message[10];
    if (val==TRUE){
        func1(message);
    }
    else
        message[0]="Status is OK";
}
```

Example 76: Declaring and initializing an array of pointers

11.3 User Defined Data Types

The most flexible complex data types are those you define yourself. C allows you to construct new data types in terms of those already defined.

11.3.1 Using typedef to Define New Data Types

The `typedef` keyword is used to define new data types. You must include an underlying type for your new type and the name of your new type.

For example, you can create a new type called `BYTE` using unsigned short int as the underlying type:

```
typedef unsigned int UBYTE;
typedef unsigned long UWORD;
UBYTE Var1;           // new variable of type UBYTE
UWORD Var2;           // new variable of type UWORD
```

Example 77: Using typedef to define a new data type

With `typedef`, the name of the new type is in the same location as the variable name in a simple variable declaration. For example, what new types are created with the following declaration?

```
struct coord_tag {
    int xVal;
    int yVal;
```

```
};  
typedef struct coord_tag COORD;  
  
typedef struct location_tag {  
    int xLoc;  
    int yLoc;  
} LOC;  
  
COORD first, last;  
LOC pt1, pt2;
```

Example 78: Defining a new enumerated type

COORD and LOC are the new types. In this case, coord_tag and location_tag are the tags for the new structures. Tags are discussed in the next section. **Example 78** shows two different techniques for using typedef with struct.

11.3.2 Using types defined with typedef

Once you have defined a new type using typedef, it can be used like any C data type.

- You can use sizeof () to retrieve memory size requirements:

```
byteSize = sizeof(UBYTE);  
structureSize = sizeof(COORD);
```

- You can cast the results of expressions:

```
// get next char from buffer, store number value in myByte  
myByte = (UBYTE)getNextChar();
```

11.4 Enumerated Types

The most straightforward complex data type is the enumerated data type, declared as type enum. The enum type is used to represent a set of possible values. The traditional example for this type is the days of the week:

```
enum WEEK { Su, Mo, Tu, We, Th, Fr, Sa } dayOfWeek;
```

This declaration creates an enumerated type called WEEK, provides seven possible values, and declares a variable called dayOfWeek of this new enumerated type. You can also separate this process into two declarations:

```
enum WEEK { Mo, Tu, We, Th, Fr, Sa, Su };  
enum WEEK dayOfWeek;
```

The label WEEK is not a new type, it is called a tag. The second line of code in the previous example requires the `enum` keyword for the declaration of `dayOfWeek`. To use WEEK as a user defined data type you require a declaration such as:

```
typedef enum { Mo, Tu, We, Th, Fr, Sa, Su } WEEK;
```

You can declare the enumerated variable `dayOfWeek` on a single line. Since the enumerated list tag WEEK represents the list itself we do not need to include it in the declaration.

```
enum { Su, Mo, Tu, We, Th, Fr, Sa } dayOfWeek;
```

The tag is useful as it can represent a list of enumerated elements to declare more than one variable of that type.

```
enum WEEK { Su, Mo, Tu, We, Th, Fr, Sa } dayOfWeek;  
enum WEEK dayOfWeek;  
enum WEEK payDay = Th;  
enum WEEK groceryDay = Sa;
```

Example 79: Declaring multiple variables of the same enumerated type

11.4.1 Enumerated Type Elements

Enumerated type elements are interpreted as integer constants. By default the first element in an enumerated list is given the integer value 0, the second element is given 1 and so on. This allows for the manipulation of values in an enumerated list as numbers:

```
dayOfWeek = Mo;  
dayOfWeek += 1; // dayOfWeek now has the value Tu
```

Example 80: Enumerated types as integer values

You can also test the elements of an enumerated type:

```
while (dayOfWeek < Sa) {  
    weekDay = TRUE;  
    getNextDay(dayOfWeek);  
}
```

Example 81: Testing the value of an enumerated type

11.4.2 Enumerated Type Value Checks

A drawback of enumerated type variables in C is the lack of boundary checking. You can legally use the expression `dayOfWeek = Fr+3`. Since the `Fr` element has the value 4, `dayOfWeek` is assigned the value 7. However, there is

no element representing member 7 of the enumerated type WEEK. These errors are often not detectable at compile time.

NOTE

Ensure that enumerated variables have the values you expect them to have by performing your own boundary checking.

11.4.3 Specifying Values for Enumerated Elements

By default, the compiler supplies a range of integer values beginning with 0 for any list of enumerated elements. This default behaviour can be modified in two ways:

- 1) Specify values for each enumerated element. The following example is from the COP8SAA7 WATCHDOG service register WDSVR. Bits 6 and 7 of this register select an upper limit to the service window which selects WATCHDOG service time.

```
enum WDWinSel { Bit7 = 7,  
                Bit6 = 6};
```

Example 82: Specifying integer values for enumerated elements

- 2) Specify a starting value for the enumerated elements. By default, the compiler assigns the value 0 to the first element in the list. You can set the list to begin with another value.

```
enum ORDINALS {first = 1, second, third, fourth, fifth};
```

Example 83: Specifying a starting value for enumerated elements

When the compiler encounters an element in an enumerated list without an assigned value it counts from the last value that was specified. For example, the following enumerated list specifies the appropriate values for its elements.

```
enum ORDINALS {first=1, second, fifth=5, sixth, seventh};
```

Example 84: The assignment of integer values to an enumerated list

NOTE

Since character constants are stored as integer values they can be specified as values in an enumerated list. `enum DIGITS {one='1', two='2', three='3'};` will store the appropriate integer values of machine character set (usually ASCII) for each digit specified in the element list.

11.5 Structures

Structures support the meaningful grouping of program data. Building the appropriate data structures is one key to the effectiveness of a new program.

The following declaration creates a structured type for the number shown by an LED display and describes each element within the structure. The display is defined as having the components `DisplaySelected`, `hundreds`, `tens` and `ones`.

```
struct Display_tag {
    int DisplaySelected;
    int hundreds;
    int tens;
    int ones;
    char AorP;
};
```

Example 85: Declaring the template of a structure

11.5.1 The structure tag

The structure tag is used as a shorthand representation for a group of structure elements. In the previous example the tag `Display_tag` represents the structure description. Note that, as with enumerated types, the compiler allocates no memory for the structure declaration itself because it is used solely as a template for variable declarations. When you declare a variable for a structure, the compiler will allocate an appropriate block of memory:

```
struct Display_tag CurrentTime;
```

You must repeat the keyword `struct` because `Display_tag` is not a valid data type, it is a structure tag. Like the enumerated type tag, it is syntactically correct to leave the tag out:

```
struct {
    int DisplaySelected;
    int hundreds;
    int tens;
    int ones;
    char AorP;
};
```

Example 86: Declaring a structure without a tag

11.5.2 Using typedef to Define a Structure

If you create a structure which is used several times in your program or you are using more than one kind of structure, it is good practice to create structure types using `typedef`.

```
typedef struct Display_tag {
    int DisplaySelected;
    int hundreds;
    int tens;
    int ones;
    char AorP;
}DISPLAY;

DISPLAY currentTime;
DISPLAY alarmTime;
```

Example 87: Using typedef to clarify structure declaration

Remember that you can declare a pointer to a `struct` before the `struct` itself is declared. The example declares a new structure type called `DISPLAY`. The use of `typedef` helps at other points in the program when you need a structure instance.

11.5.3 Accessing Structure Members

C includes two binary operators which allow access to structure members: the dot operator, `.`, and the structure pointer operator, `->`. In each case, the binding identifier to the left of the operator indicates the structure and the binding identifier to the right of the operator indicates the element within that structure.

11.5.4 Indicating a Field with the Dot Operator

Once a `struct` variable is declared you can use the dot operator to reference an element of the structure. The following assigns values to the elements of the `currentTime` variable for the structure defined in **Example 87**.

```
currentTime.DisplaySelected = 1;
currentTime.hundreds = 9;
currentTime.AorP = "A";
alarmTime.AorP = currentTime.AorP;
```

Example 88: Accessing elements in a structure

11.5.5 Indicating a Field with the Structure Pointer

Structures are often manipulated using pointers. C has an operator specially for this purpose; the structure pointer operator. In order to use a pointer to access members of a structure the pointer must first be pointed at the structure instance. The following example points `Display_Ptr` to `alarmTime` and then accesses the elements of `alarmTime`.

```
struct Display_tag * Display_Ptr;
struct Display_tag {
    int DisplaySelected;
    int hundreds;
    int tens;
    int ones;
    char AorP;
}alarmTime;

Display_Ptr = &alarmTime; //point Display_Ptr to alarmTime
Display_Ptr->ones = 7;    //set alarmTime.ones to 7
Display_ptr->AorP = 'P';  //set alarmTime.AorP to P
Display_Ptr->tens = 9;     //set alarmTime.tens to 9
(*Display_Ptr).tens = 9;  //set alarmTime.tens to 9
```

Example 89: A structure accessed with a pointer

The last line `(*Display_Ptr).tens = 9;` does the same thing as `Display_Ptr->tens = 9;`, assigns the value 9 to the `tens` element of `alarmTime`.

Notice the parentheses around the dereference sub-expression. The dot operator has a higher precedence than the dereference operator. If the parentheses are omitted the expression `*Display_Ptr.tens` would attempt to return the address of the `tens` element of the `Display_Ptr` structure. Since `Display_Ptr` is not a structure, this would give an error. `(*Display_Ptr).value` dereferences the pointer *first* and returns the structure object pointed to by `Display_Ptr`. The expression then returns the `tens` element from this structure.

11.5.6 Bit Fields in Structures

Using bit fields allows the declaration of a structure which takes up the minimum amount of space. A bit field contains a specified number of bits, it is a member of a structure and is accessed like any other structure member. The following example for the Motorola MC68HC705C8 defines the Timer Control Register (TCR) bits as bit fields in the structure called TCR.

```
struct reg_tag {
    int ICIE : 1; // field ICIE 1 bit long
    int OCIE : 1; // field OCIE 1 bit long
    int notUsed : 3 = 0; //notUsed is 3 bits and set to 0
    int IEDG : 1; // field IEDG 1 bit long
    int OLVL : 1; // field OLVL 1 bit long
} TCR;
```

Example 90: Bit fields in structures

C implements bit fields as variable length integer elements within a structure. A bit field is accessed with the structure operators. You cannot use a pointer to point to the bit field element directly; you must access it through the structure using the `->` operator:

```
struct reg_tag * TCRFieldPtr;
TCRFieldPtr = &TCR;
TCR.ICIE = 1; // access using dot operator
TCRFieldPtr->ICIE = 1; // using right arrow operator
```

Example 91: Accessing bit fields

11.5.7 Storing bit fields in memory

Storage of bit fields in memory varies from one compiler to another. Some compilers cannot store a bit field over a word boundary. In this case the following structure would place the second field entirely in a separate word of memory from the first:

```
struct {
    unsigned int shortElement : 1; // 1 bit in size
    unsigned int longElement : 8; // 8 bits in size
} myBitFields;
```

Example 92: Compiler dependant storage of bit fields

The order in which the compiler stores elements in a structure bit field also varies from compiler to compiler. Some compilers may use the first word of allocated memory to hold `longElement` in the previous structure. Other compilers may use the first word to contain `shortElement` and part or none of `longElement`.

11.5.8 The behaviour of bit fields

Bit field elements behave exactly as an `int` of the same size. Thus an element occupying a single bit could have an integer value of either 0 or 1, while an element occupying two bits could have any integer value ranging from 0 to 3.

You can use each field in calculations and expressions exactly as you would an `int`.

11.6 Unions

C programmers developing for traditional platforms do not often use the union data type, but it is very useful resource for the embedded system developer. The union type filters data stored in a single block of memory based on associated data types.

For example, when you declare two individual `int` and `char` variables the compiler will allocate two 8 bit blocks of memory:

```
int anInt;
char aChar;
```

However, if you place both these variables in a union the compiler only allocates a single 8 bit block of memory for both variables:

```
union share_tag {
    int as_Int;
    char as_Char;
} share; //share is the variable name
```

Example 93: Declaring a union

The format of union resembles that of the structure. You can identify a union with a tag name. To make your union a data type you must use the `typedef` keyword. In the following example, a new type called `share` is created.

```
typedef union share_tage {
    int asInt;
    char asChar;
} share_type; //share_type is the data type
share_type share; //share is the variable name
```

Example 94: Using typedef to declare a union

One common use of the union type in embedded systems is to create a scratch pad variable that can hold different types of data. This saves memory by reusing one 16 bit block in every function that requires a temporary variable.

The following example shows a declaration to create such a variable:

```
struct lohi_tag{
    short lowByte;
    short hiByte;
};
```

```
union tagName {
    int asInt;
    char asChar;
    short asShort;
    long asLong;
    int near * asNPtr;
    int far * asFPtr;
    struct hilo_tag asWord;
} scratchPad;           //scratchPad is the variable name
```

Example 95: Using a union to create a scratch pad

Another common use for union is to facilitate access to data as different types. For example, the Microchip PIC16C74 has a 16 bit timer/counter register called TMR1 made up of two 8 bit registers called TMR1H (high byte) and TMR1L (low byte). It is possible that sometimes you would like to access the register as two 8 bit values or as one 16 bit value. A union will facilitate this type of data access:

```
struct asByte {
    int TMR1H;    //high byte
    int TMR1L;    //low byte
}
union TIMER1_tag {
    long TMR1_word;           //access as 16 bit register
    struct asByte TMR1_byte;
} TMR1;
```

Example 96: Using a union to access data as different types

11.6.1 Retrieving a Union Element

As with structures, union elements are accessed with the dot and right arrow operators. Use the dot operator to specify an element by placing it after the name given to the union. In the following example, the data in the scratchPad memory block is interpreted as a char.

```
scratchPad.asChar = 'b';           //assign b to scratchPad
tempChar = scratchPad.asChar;      //retrieve as character
```

Example 97: Accessing a union element with the dot operator

If you indicate the union with a pointer, use the right arrow operator to specify an element. In the following example, scratchPad is interpreted as an int.

```
union tagName * scratchPad_ptr;    //declare pointer type
scratchPadPtr = &scratchPad;       //point to scratchPad
someInt = scratchPad_ptr->asInt;    //retrieve as integer
```

Example 98: Using the right arrow operator to access a union member

11.6.2 Using Unions with Incompatible Variables

Since the compiler uses a single block of memory for the entire union, it allocates a block large enough for the largest element in the union. For example, the compiler will allocate a 16 bit block for the union `scratchPad` in

Example 98 because the elements `asLong` and `asFPtr` require 16 bits¹⁶.

The compiler will align the first bit of each element in the memory block. If you assign a 16 bit value to `scratchPad` and then read it as an 8 bit value, the compiler will return the first 8 bits of the data stored.

NOTE

Verify your target hardware's method for storing 16 bit integer values. Some hardware stores long data with a higher address for the low byte. This is called *big endian* because the "big end" comes at the end. Other hardware stores the high byte at the higher address. This is called *little endian* because the "little end" comes last. The results returned from extracting 8 bits from a 16 bit value will differ depending on the hardware storage method.

The `scratchPad` variable can handle the 16 bit value as a word and can provide access through a structure to either byte in the word. This is useful so you can use the `asWord` element to return a specific part of the word.

```
scratchPad.asLong = someLong;
someInt = scratchPad.asWord.lowByte;
```

Example 99: Returning the low Byte of a word

Notice that the `scratchPad` example assumes the target hardware is big endian (high byte last). For a little endian target (low byte last), the `asWord` element needs to be defined as follows¹⁷. Notice that redefinition does not affect the statements in the previous example.

```
struct hilo_tag {
    short highByte;
    short lowByte;
} asWord;
```

Example 100: Returning a specific part of a word for little endian

¹⁶ `asInt` may require 16 bits, depending on the compiler.

¹⁷ To promote even greater portability and clarity define a new data type called `BYTE` based on the underlying 8 bit data type on the target hardware.

The problem of incompatible variables is exacerbated when the variables have different underlying storage methods. For example, the following `union` gives surprising results if you do not keep track of the last data assigned to it. Since floating point numbers typically use mantissa/exponent representation the result may not be as expected:

```
union {
    int asInt;
    float asFloat;
} someUnion;

someUnion.asFloat = someFloat;
someInt = someUnion.asInt;
```

Example 101: Incompatible variables with different storage methods in unions

12. Storage and Data Type Modifiers

C provides the capability to further specify how stored values should be interpreted with the use of **storage class** and **data type modifiers**. Many of these modifiers have been introduced briefly in other sections of this book. Both storage class and data type modifiers are keywords which are included in a variable or function data type declaration.

12.1 Storage Class Modifiers

Storage class modifiers control memory allocation for declared identifiers. C supports four storage class modifiers¹⁸ which can be used in variable declarations: `extern`, `static`, `register` and `auto`. Only `extern` is used in function declarations.

When the compiler reads a program it must decide how to allocate storage for each identifier. The process used to accomplish this task is called **linkage**. C supports three classes of linkage – external, internal and none. C uses identifier linkage to sort out multiple references to the same identifier.

12.1.1 External linkage

References to an identifier with **external linkage** throughout a program all call the same object in memory. There must be a single definition for an identifier with external linkage or the compiler will give an error for duplicate symbol definition. By default, every function in a program has external linkage. Also by default, any variable with global scope has external linkage.

12.1.2 Internal linkage

In each compilation unit¹⁹, all references to an identifier with **internal linkage** refer to the same object in memory. This means that you can only provide a single definition for each identifier with internal linkage in each compilation unit of your program.

¹⁸ The ANSI standard specifies a fifth modifier: `typedef`

¹⁹ A compilation unit is not always a single file of code because of `#include` files

No objects in C have internal linkage by default. Any identifier with global scope (defined outside any statement block), *and* with the `static` storage class modifier, has internal linkage. Also, any variable identifier with block scope (defined within a statement block), *and* with the `static` storage class modifier, has internal linkage.

Although you can create local variables with internal linkage scoping rules restrict local variable visibility to their enclosing statement block. This means that you can create local variables whose values persist beyond the immediate life of the statement blocks in which they appear. Normally the computer re-allocates local variable space every time a statement block is entered. If a local variable is declared as `static`, space is allocated for the variable once only – the first time the variable is encountered.

NOTE

Unlike other internal linkage objects, static local variables need not be unique within the compilation unit. They must be unique within the statement block which contains their scope.

Objects with internal linkage typically occur less frequently than objects with external or no linkage.

12.1.3 No linkage

References to an identifier with no linkage in a statement block refer to the same object in memory. If you define a variable within a statement block, you must provide only one such definition. Storage for objects with no linkage is traditionally allocated from stack space.

Any variable declared within a statement block has no linkage by default, unless the `static` or `extern` keywords are included in the declaration. Both function return values and function parameters have no linkage, allowing recursive function calls. Each copy of a recursively called function can allocate private copies of parameters and return values.

12.1.4 The `extern` Modifier

An identifier with external linkage can be used at any point within a program as long as it is visible. Suppose the function `int Calculate_Sum()` is declared in a source file. If you want to use this function in any other compilation unit, you must tell the compiler where to look for the function

definition. The concept is identical to prototyping a function so that it can be used before it is defined. To declare an external function use the `extern` keyword:

```
extern int Calculate_Sum();
```

When the compiler encounters an external function declaration it interprets it as a prototype for the function name, type and parameters. The `extern` keyword tells the compiler that the function definition is in another compilation unit. The compiler leaves the connection of such code to the *linker* whose job it is to resolve references to symbols between compilation units.

If you build a library of functions to use in many programs it is good practice to include `extern` function declarations in a header file which is included in the source files for your program.

You can declare an external function within a statement block using the `extern` keyword. This informs the compiler that the function is defined elsewhere in the program and restricts the scope of the function to the statement block.

For example, suppose the `initialize()` function is in a subsidiary source file and you want it visible only to `main()`. The following code lets `main()` know about `initialize()` while hiding it from other functions in the same source file as `main()`.

```
void main(void) {  
    extern int initialize(void);  
    initialize();  
}
```

Example 102: Restricting a function's scope by declaring it as `extern`

12.1.5 Global Variables and `extern`

Like functions, global variables have external linkage. To use a global variable in more than one source file, you must declare it as `extern`:

```
extern int myGlobalInt;
```

The compiler interprets an external declaration not as a variable declaration but as a notice that the variable definition occurs in another file. You must link files with external declarations with a main module whose source contains the declaration for the variable:

```
int myGlobalInt;
```

Developers often collect global variable definitions in a header file called `globals.h`. Each declaration in the file will look similar to:

```
EXT int myGlobalInt;
```

Preprocessor directives are placed at the top of the file to handle the `EXT` tag in each definition:

```
#ifdef MAIN
    #define EXT " "
#else
    #define EXT "extern"
#endif
```

Example 103: Using preprocessor directives to declare extern global variables

Each program source module will contain the line `#include <globals.h>`. At the top of the main source file, before `global.h` is included, the directive `#define MAIN` should appear. This keeps global variable declarations in one place and ensures that the `extern` keyword is only used when needed. The main source file contains definitions for the variables without the `extern` keyword.

12.1.6 The static Modifier

By default, all functions and variables declared in global space have external linkage and are visible to the entire program. Sometimes you require variables or functions which have internal linkage: they are visible within a single compilation unit. Use the `static` keyword to restrict the scope of variables:

```
static int myGlobalInt;
static int staticFunc(void);
```

Example 104: Using the static data modifier to restrict the scope of variables

These declarations create global identifiers which are not accessible by any other compilation unit. Any function within the same compilation unit as the `static` variable declarations can access these identifiers.

12.1.7 The visibility of static variables

The `static` keyword can be used to create permanent local variables. For example, consider the task of tracking the number of times a recursive function calls itself (the function's depth). You can accomplish this using a static variable:

```
int myRecurseFunc(void) {
    static int depthCount=1;
    depthCount += 1;
    if ( (depthCount > 10) || (!DONE) ) {
        myRecurseFunc();
    }
}
```

Example 105: Using static variables to track function depth

The function in **Example 105** contains an if statement which stops it from recursing too deeply. The static variable `depthCount` is used to keep track of the current depth. Normally, when a function is called the computer re-allocates memory for its automatic local variables. Memory for static variables, however, is only allocated once. The static variable `depthCount` retains its value between function calls and conserves memory because 8 bits is not allocated every time `myRecurseFunc()` calls itself.

Because `depthCount` is defined inside the `myRecurseFunc()` statement block, it is not visible to any code *outside* the function. Therefore, if you have another recursive function you can use the variable name `depthCount`²⁰.

12.1.8 The register Modifier

The `register` keyword is not often used in embedded systems programming because the target hardware does not have the variety of registers available on traditional C platforms.

When you declare a variable with the `register` modifier you inform the compiler to optimize access to the variable for speed. Traditionally, C programmers use this modifier when declaring loop counter variables:

```
{
    register int myCounter=1;
    while (myCounter<10) {
        ...
        myCounter += 1;
    } //end while
} // enclosed block enforces reallocation of myCounter
```

Example 106: Using the register data type modifier

²⁰ If functions are mutually exclusive use a global variable to save memory.

Unlike other storage class modifiers, `register` is simply a recommendation to the compiler. The compiler may use normal memory for the variable if it determines that such an allocation will allow the fastest access to the variable.

Because of the scarcity of registers on 8 bit machines and the desire for size optimization rather than speed, the `register` keyword is not very useful for embedded system programmers.

Notice the technique used in **Example 106** places the `register` variable declaration and its associated `while` loop inside a statement block. This forces the compiler to reallocate storage for `myCounter` as soon as the loop is finished – if the compiler uses a register to store `myCounter`, it will not tie up the register longer than necessary.

NOTE

If you use `register` ensure that the code for the variable declaration is close to the code where the variable is used. This minimizes the overhead expense of dedicating a register for storage of a single particular variable.

12.1.9 The auto Modifier

The `auto` keyword denotes a temporary variable. You may only use `auto` with variables because C does not support functions with local scope. Since all variables declared inside a statement block have no linkage by default, the only reason to use the `auto` keyword is for clarity:

```
int someFunc(NODEPTR myNodePtr) {
    extern NODEPTR TheStructureRoot;
    // global pointer to data structure root
    auto NODEPTR tempNodePtr;
    // temporary pointer for structure manipulation
    ...
}
```

Example 107: Using the auto data modifier

In this example, we declare `tempNodePtr` as an `auto` variable to make it clear that, unlike the global `TheStructRoot` pointer, `tempNodePtr` is only a temporary variable.

12.2 Data Type Modifiers

Data type modifiers alter the way information is recorded and retrieved. Type modifiers extend the basic data types available. Type modifiers apply to data only, not to functions. You can use them with variables, parameters, and returned data from functions.

Some type modifiers can be use with any data while others are used with specific types of data such as pointers.

12.2.1 Value Constancy Modifiers: `const` and `volatile`

The compiler's ability to optimize a program relies on several factors. One of these is the relative constancy of the data objects in your program. By default, variables used in a program change value when the instruction to do so is given by the developer.

`const`

Sometimes you want to create variables with unchangeable values. For example, if your code makes use of π , the constant PI, then you should place an approximation of the value in a constant variable:

```
const float PI = 3.1415926;
```

When your program is compiled, the compiler allocates ROM space for your PI variable and will not allow the value to be changed in your code. For example, the following assignment would produce an error at compile time:

```
PI = 3.0;
```

`volatile`

Volatile variables are variables whose values may change without a direct instruction. For example, a variable which contains data received from a port will change as the port value changes.

Using the `volatile` keyword informs the compiler that it can not depend upon the value of a variable and should not perform any optimizations based on assigned values.

12.2.2 Allowable Values Modifiers: signed and unsigned

You can direct the compiler to permit integer data types to contain negative as well as positive values. You can also restrict integer data types to positive values only. The sign value of an integer data type is assigned with the `signed` and `unsigned` keywords.

signed

The `signed` keyword forces the compiler to use the high bit of an integer variable as a sign bit. If the sign bit is set with the value 1 then the rest of the variable is interpreted as a negative value. By default, `short`, `int` and `long` data types are signed and the `signed` keyword need not be used. The `char` data type is unsigned by default. To create a signed `char` variable you must use a declaration such as:

```
signed char mySignedChar;
```

If you use the `signed` keyword by itself the compiler assumes that you are declaring an integer value. Since `int` values are signed by default, programmers rarely use the syntax: `signed mySignedInt;`

unsigned

To create unsigned `short`, `int`, or `long` data types use the `unsigned` keyword. You need never use the keyword with `char` values because they are unsigned by default. This keyword forces the computer to read the high bit as part of the variable value:

```
unsigned int myUnsignedInt;
```

If you use the `unsigned` keyword alone the compiler assumes the variable you are declaring is an `int`. C programmers often use the following syntax:

```
unsigned myUnsignedInt;
```

12.2.3 Size Modifiers: short and long

The `short` and `long` modifiers instruct the compiler how much space to allocate for an `int` variable. The resulting variable is interpreted as an `int`, but the number of bits used to store the variable value may change.

short

The `short` keyword declares an `int` of the same size as a `char` variable: usually 8 bits:

```
short int myShortInt;
```

On microcontrollers where the natural machine unit is the byte a `short int` is usually the same size as an `int`. Some compilers allow two byte `int` variables. In these cases, the `short int` remains 8 bits in size.

If you use the `short` keyword alone, the compiler assumes the variable is a `short int` type:

```
short myShortInt;
```

long

The `long` keyword declares an `int` twice as long as a normal `int` variable:

```
long int myLongInt;
```

On some computers a `long` is not twice the size of an `int`. However, `long` will *always* be the same size or larger than `int` and `short` will *always* be the same size or smaller than `int`.

On microcontrollers a `long int` occupies two bytes. If the compiler allows you to use 16 bit `int` data types, the `long` and `int` are usually the same size because of the fact that `long` data types always occupy two bytes.

12.2.4 Pointer Size Modifiers: near and far

The `near` and `far` keywords are common extensions to standard C. They allow different size pointers to address different areas of computer memory.

near

The `near` keyword creates a pointer which points to objects in the bottom section of addressable memory. These pointers occupy a single byte of memory, and the number of memory locations to which they can point is limited to the first 256 locations, or from `$0000` to `$00FF`.

```
int near * myNIntptr;
```

For efficient RAM access, most microcontrollers place user RAM in the low memory addresses. Thus, near pointers usually point to data stored in user RAM such as user defined variables.

far

The `far` keyword creates a pointer which can point to any data in memory:

```
int far * myFIntPtr;
```

These pointers take two bytes of memory which allows them to hold any legal address location from \$0000 to \$FFFF. `far` pointers usually point to objects in user ROM, such as user defined functions and constant variables.

12.2.5 Using near and far pointers

Each microcontroller has different memory usage and the specific implementation of near and far pointers will vary depending on the target platform. In general, microcontrollers fall into two groups:

- 1) Harvard architecture machines that maintain separate memory areas for data memory, RAM, and program memory, ROM.
- 2) Von Neumann architecture machines which arrange ROM and RAM into one contiguous address space.

Regardless of machine architecture, the compiler uses near pointers to point to commonly referenced data such as variables. The `far` pointers are harder to manipulate and are used for less common pointing tasks such as pointing to functions and constants.

12.2.6 Default pointer type

Since the implementation of near and far pointers varies from target to target the default method of creating pointers also varies. For example, what kinds of pointers do the following two declarations generate?

```
int * myIntPtr;  
const int * myConstIntPtr;
```

On most target machines, the compiler generates a near pointer for the first declaration and a far pointer for the second. Since the compiler *knows* that `const int` data is stored in ROM it knows a far pointer is needed.

The following declaration generates a far pointer to the void function `initPtr` knowing that the `*initPtr()` function will get stored in ROM.

```
void (*initPtr)(STATSTRUCT * statusPtr){  
    // contents of function  
};
```

Example 108: The far pointer type as default

If you use pointers extensively you must know the default pointer type. Many embedded developers do not use pointers extensively as they are very CPU intensive. This is especially true with the far pointer double byte values.

13. The C Preprocessor

Every C language environment has a preprocessor. As the name suggests, the preprocessor examines program code before it is processed by the compiler. The preprocessor reads a source code file line by line and performs the preprocessor directives it finds.

The preprocessor does not understand the C language. This can be a source of great trouble for program developers as it is easy to miss problems caused by passing the preprocessor invalid commands. Two common errors are including a semicolon to terminate a macro definition and placing a comment on the same line as a directive. Since the preprocessor does not understand the C interpretation of semicolons or comments it will attempt to read these things as part of the directive.

Some C environments support an option which invokes *only* the preprocessor for a source file. This has the advantage of letting you look at the preprocessor results *before* the source gets passed to the compiler.

13.1 Preprocessor Directive Syntax

Any source code line that begins with the hash character, #, is a command to the preprocessor and is called a preprocessor directive. It is good practice to justify these directives against the left hand margin to distinguish them from your C code. Historically, pre-ANSI compilers required preprocessor directives to begin in column one of a source code line. This practice should not be followed when you nest directives:

```
#if DEBUG
    #include <debug.h>
#endif
```

Example 109: Nesting preprocessor directives

The hash character must be the first non-white space character in a preprocessor directive. When a line begins with # the preprocessor assumes that the entire line is part of the same directive. To continue a single directive past a single line place the continuation character \ at the end of the line. When this character appears the preprocessor attaches the contents of the next line to the end of the current directive.

13.2 White Space in the Preprocessor

Unlike the C compiler, white space is very important to the preprocessor. For example, in C both the following function definitions are acceptable:

```
int smallest (int arg1, int arg2);
int largest(int arg1, int arg2);
```

The preprocessor is not so forgiving. Only one of the following two macros performs as expected:

```
#define SMALLEST (arg1,arg2) ((arg1)<(arg2)?(arg1):(arg2))
#define LARGEST(arg1,arg2) ((arg1)<(arg2)?(arg1):(arg2))
```

SMALLEST is defined as an object macro or *symbolic constant*, not as a function macro like LARGEST as intended. Thus a call to SMALLEST will be expanded by the preprocessor into the monstrosity:

```
(arg1,arg2)((arg1)<(arg2)?(arg1):(arg2))(oneInt,twoInt);
```

13.3 File Inclusion

The `#include` directive instructs the preprocessor to replace the directive with the contents of a specified file. That file need not contain C source code; for example, it can consist of nothing but preprocessor directives. In embedded system programs a header file which describes the resources of the target hardware is usually included:

```
#include <machine.h>
```

When the preprocessor sees this directive it will look for the file `machine.h` and replace the directive with the contents of `machine.h`. The preprocessor will then continue searching through source code. The next line it will look at will be the first line of the `machine.h` file.

If the preprocessor cannot find the specified file, it will give an error and quit processing. Where does the preprocessor look for the file?

13.3.1 File Inclusion Searches

`<filename.h>`

If you surround the file name with angle brackets the preprocessor will look for the file in a system dependent location determined by the compiler you are using.

In general, angle brackets produce two types of searching. On some systems, the preprocessor will look through a directory or list of directories you have specified as containing the library and header files for your compiler. On other systems the preprocessor will look through a directory or list of directories specified in the operating system environment as a location for commands.

`"filename.h"`

If you surround the file name with double quotes, the preprocessor behaviour is more complex.

- 1) The preprocessor looks for the file in a *system dependent* location. This may be the same location used for `<>` inclusion; however, it usually is not. If the preprocessor searches for include files in a single location, the preprocessor does not support `""` inclusion and treats it as `<>` inclusion.
- 2) If the file is not found, the preprocessor will retry the directive as if the file were surrounded by angle brackets.

In general practice, the double quotes signal the preprocessor to look for the file in the same place as the source code file containing the directive.

NOTE

If the preprocessor can not find the file in the place for `""` inclusion it will reprocess the directive as if it used `<>` inclusion syntax.

The common misconception that `""` inclusion refers to the current directory can lead to errors. You must check your compiler documentation to determine exactly where and how `""` and `<>` inclusion look for files.

13.4 Defining Symbolic Constants

The `#define` directive instructs the preprocessor to create a symbolic constant.

```
#define MAXINT +32768
```

This directive creates a symbolic constant `MAXINT` and associates it with a value of `+32768`. Here we intend `MAXINT` to stand for the largest 16 bit signed integer value the target hardware can represent.

When the preprocessor reaches the `#define` directive it places `MAXINT` into its list of defined symbols. The preprocessor will replace `MAXINT` with its defined value in any subsequent lines that contain the `MAXINT` symbol.

The association of this symbolic constant with its value is *not* passed on to the compiler. When the compiler examines the source file, the symbol `MAXINT` does not appear – the preprocessor has replaced it with the appropriate value. Symbol expansion does not happen within other preprocessor directives. You *can* use symbolic constants inside macro definitions, but the expansion of the symbol happens after the macro expansion. The symbol is *first* placed in the source code and then expanded.

There are two main reasons why symbolic constants are useful:

① Symbolic constants clarify ambiguous source code

You can place a meaningful word in your source code, instead of a potentially ambiguous value. For example, the number `3.0e+5` might not be clear. However, suppose we include the following directive:

```
#define LIGHTSPEEDkps 3.0e+5
```

You can see that the symbol might convey more meaning in the code than its value alone.

② Symbolic constants facilitate code maintenance

Symbolic constants, like variables, reduce typing errors. Once `MAXINT` is defined its value is assigned in a single location in your source code. If you need to change the value of `MAXINT` you need only edit the `#define` directive and recompile. Without the directive you would have to change every occurrence of the value in your program. Additional problems are encountered if the same value has different meanings.

13.4.1 The `#undef` directive

You may want to redefine the value of a symbolic constant. The preprocessor may give an error if you attempt to define a symbol that is already defined.

According to the ANSI standard you can redefine a symbolic constant with a replacement string which is exactly similar. Despite this, it is best to be scrupulous about using `#undef` for symbols before you redefine them.

You must tell the preprocessor to remove the symbol from its list before you can redefine it.

```
#undef MAXINT
#define MAXINT +127
```

Example 110: Redefining a constant using `#undef`

Suppose you have a small set of functions that you want to keep 8 bit portable, while allowing remaining functions to use 16 bit `int` values. The following directives would be used:

- | | |
|--------------------------------|------------------------------------|
| 1) Define MAXINT for 16 bit | <code>#define MAXINT +32768</code> |
| 2) Undefine MAXINT | <code>#undef MAXINT</code> |
| 3) Define MAXINT for the 8 bit | <code>#define MAXINT +127</code> |

Undefining a symbol has no effect if a symbol is not defined, the preprocessor simply ignores the `#undef` directive.

13.4.2 Defining “empty” symbols

Another useful feature of symbolic constants is that they do not have to be defined with associated values. For example:

```
#define 8BITINT
```

This directive instructs the preprocessor to place the symbol `8BITINT` into its symbol list with no associated value. If you use the symbol in your code the preprocessor replaces it with nothing. This can easily lead to compiler errors.

13.5 Defining Macros

Function macros are a powerful aspect of the C preprocessor. Macros are defined using the `#define` directive.

A function macro is a replacement macro with an argument list. When the preprocessor encounters a macro reference it performs a text replacement and retains the arguments listed with the macro in the source code. The preprocessor can provide a means for data abstraction – each invocation of a function macro deals with different values in a predictable way.

A simple example will clarify the behaviour of macros:

```
#define SMALLEST(arg1,arg2) ((arg1)<(arg2)?(arg1):(arg2))

// program code
someInt = SMALLEST(oneValue, twoValue);
```

Example 111: Defining and calling a macro

The `#define` in **Example 111** creates a macro called `SMALLEST` which returns the smaller of two arguments. The line which calls the function macro looks as follows after it has been processed by the preprocessor:

```
someInt = ((oneValue)<(twoValue)?(oneValue):(twoValue));
```

NOTE

Because a function macro looks similar to a function call it can be difficult to tell macro functions and regular functions apart. It is good coding practice to use upper case for all macro names so they are easily distinguished from functions code.

13.5.1 Macro Expansion

You can pass expressions as arguments to a function macro. There is a difference between passing expressions to macros and passing them to functions. When you pass expressions to functions they are first evaluated and the resulting *values* are received by the function. As the preprocessor simply performs text replacement; it does not evaluate expressions passed to a macro. For this reason you must use macros carefully. For example, here is a common macro error:

```
#define SQUARE(x) x * x
```

Consider the following call to `SQUARE`:

```
someInt = SQUARE(a+1);    // before expansion
someInt = a+1 * a+1;      // after expansion
```

C precedence rules produce an unintended result from this calculation. The use of parentheses is important in a macro definition using expressions. A better definition of `SQUARE` looks like:

```
#define SQUARE(x) ((x) * (x))
```

The parentheses around each parameter reference will preserve the expression's internal precedence and the parentheses around the macro will preserve its precedence with respect to other code.

Even with parentheses, using `SQUARE` as follows will produce unexpected results:

```
someInt = SQUARE(a++);    // before expansion
someInt = ((a++) * (a++)); // after expansion
```

Because `a` is not evaluated in the same manner as it would be in a function call, it is evaluated *twice* at compile time and `a` is incremented before the multiplication. If `SQUARE` were a function, `a` would have been evaluated once at compile time and the resulting value passed to the function. You can see the value in clearly distinguishing the function and macro names.

NOTE

Using *any* expression that causes side effects as an argument to a macro or a function call is not good practice and can cause unexpected results.

13.5.2 # and ## Operators

To expand macro parameters inside quotes you need to use the `#` and `##` operators

13.6 Conditional Source Code

The preprocessor supports directives which allow conditional compilation of your source code. You can bracket program portions and let the preprocessor decide whether or not to pass these portions of the code on to the compiler.

13.6.1 #if and #endif

The `#if` and `#endif` directives include code when the `#if` expression evaluates to a non-zero integer value:

```
#define DEBUG 1
#if DEBUG
    #include <debug.h>
#endif
```

Example 112: Using `#if` and `#endif` to conditionally compile code

Blocks of code such as that in **Example 112** are often used to produce both a debugging and final version of a program. The first line defines the `DEBUG` symbol with the value 1. The `#if` directive tests its argument expression to see if it has a non-zero constant integer value. When `DEBUG` has a non-zero value,

the preprocessor will `#include` a header file created for debugging called `debug.h`.

Because `#if` accepts an expression as an argument, you can also do the followings to check for the value assigned a symbolic constant:

```
#define DEBUG_STATE 1
#if DEBUG_STATE == 1
    #include <debug1.h>
#endif
```

Example 113: Using expressions in `#if` directives for conditional compilation

13.6.2 The `defined( )` Function

The constant integer expression tested by `#if` cannot contain the `sizeof( )` function, type casts, or enum constants. However, you can use the `defined( )` function with `#if` directives. The `defined( )` function returns 1 if its argument is a defined symbol. If the symbol is not defined, it returns 0. Therefore, we can rewrite **Example 113** as follows:

```
#define DEBUG
#if defined(DEBUG)
    #include <debug.h>
#endif
```

Example 114: Using the `defined()` function for conditional compilation

You can also use `!defined( )` to test if a symbol has not been defined. It will return 1 if its argument is *not* a defined symbol and 0 if the argument is defined:

```
#if !defined(DEBUG)
    #include <machine.h>
#endif
```

Example 115: Using `!defined()` to test if a symbol has not been defined

13.6.3 The `#else` and `#elif` Directives

The C preprocessor includes the ability to choose between two compilation blocks using the `#else` directive. For example, suppose that the debug header file includes descriptions of target resources. To avoid including these twice, you could write:

```
#define DEBUG 1
#if DEBUG == 1
    #include <debug.h>
#else
    #include <machine.h>
#endif
```

Example 116: Using #else and #elif to choose between compilation blocks

If you want to build a switch-like structure of compilation blocks, use the #elif directive inside a #if and #endif pair. You can use as many #elif directives as necessary but you can only have one #else, which must come after the #elif directives.

```
#define STATE DEBUG
#if STATE == DEBUG
    #include <debug.h>
#elif STATE == TESTING
    #include <testing.h>
#elif STATE == RELEASE
    #include <machine.h>
#endif
```

Example 117: Using #elif, #if and #endif for conditional compilation

13.6.4 #ifdef and #ifndef

If you do not use the defined or !defined operators in a directive, you can use the directives #ifdef or #ifndef. #ifdef FOO is equivalent to #if defined(FOO) while #ifndef FOO is equivalent to #if !defined(FOO):

```
#define DEBUG
#ifdef DEBUG
    #include <debug.h>
#endif
#ifndef DEBUG
    #include <machine.h>
#endif
```

Example 118: Using #ifdef and #ifndef

13.7 Producing Error messages

The #error directive halts the preprocessor and produces a specified error message. Most compilers provide additional information with your message,

such as the name of the source file and the position of the error directive within that file:

```
#if STATE == DEBUG
#include <debug.h>
#elif STATE == RELEASE
#include <machine.h>
#else
#error Bad or missing STATE value: need DEBUG or RELEASE
#endif
```

Example 119: Using the #error directive

13.8 Defining Target Hardware

The standard C environment allows the definition of compiler-specific extensions with the #pragma preprocessor directive. The preprocessor may deal with #pragma directives in your source code or it may be the compiler which acts upon these directives.

ANSI C has one prescribed rule about #pragma directives – if a #pragma directive is not recognised, it is ignored and passed on. This ensures that #pragma directives that are unknown will not affect your code.

The #pragma directive is used most commonly in embedded development to describe specific resources of your target hardware such as available memory, ports, and specialized instruction sets.

13.9 In-line Assembly Language

While not required by ANSI C, most embedded development compilers provide a means of incorporating assembly language in C programs. One common way of accomplishing this is using preprocessor directives.

13.9.1 The #asm and #endasm Directives

Some compilers use #asm and #endasm directives to signal assembly language code boundaries. Everything lying between the directives is assumed to be assembly language code and will be processed by a macro assembler which is either built-in to the compiler or a secondary program called by the compiler.

14. Libraries

Technically, a library in C is simply a collection of C functions. Libraries usually contain functions which serve a common purpose, such as interfacing to an LCD, using a timer, providing mathematical capabilities, or converting data types. The functions within a library are a collection of the basic operations defined by the scope of the library. For instance a math library would contain routines for multiplication, division, and modulus.

Because high level languages are very portable, libraries written in high level languages are also very portable. Portability is made possible by the standardization of high level languages such as C. C language code written on a PC will compile and run on MAC or UNIX machines often with little or no alternation. Similarly, C code written for a specific 8 bit microcontroller can be compiled and run on a different microcontroller with very minor changes to the code.

Although libraries for math and data type conversion are useful, they are not the libraries most useful in embedded systems development. By definition a microcontroller embedded within a system needs to receive data in and sends data out. This is most often done with devices such as keyboards, LCD displays, serial interfaces, and I/O ports. At times it is necessary to convert this data to a specific format so that it can be understood. Devices such as Analog to Digital and Digital to Analog converters provide such conversion capabilities. Libraries which support peripheral devices are very useful in embedded systems development.

14.1 Portable Device Driver Libraries

C's portability allows us to implement **Portable Device Driver** libraries. A portable device driver is a standard technique for using a specific peripheral device with a range of different microcontrollers, both between and within microcontroller families. Why would we want to do this? The main reason is to save development time. In the embedded marketplace time to market is probably the most important mitigating factor in the design process.

The advantages realized with portable device driver libraries are:

- 1) We do not need to “reinvent the wheel”. Device drivers would not have to be rewritten for every new project.

- 2) The libraries have been thoroughly tested and debugged allowing faster hardware/software integration
- 3) The embedded programmer does not need to know the low level hardware details of how the device operates.
- 4) Support for multi-controller systems which use microcontrollers from different families. C source code can be ported between families by changing the included header file. This saves the embedded programmer from having to learn implementations on different microcontroller.
- 5) Software reusability is maximized.

Some useful portable libraries would provide routines for:

- 1) SPI (Serial Peripheral Interface)
- 2) Microwire
- 3) SCI (Serial Communications Interface)
- 4) UART (Universal Asynchronous Receiver Transmitter)
- 5) USART (Universal Synchronous Asynchronous Receiver Transmitter)
- 6) Analog to Digital conversion and Digital to Analog Conversion
- 7) I/O ports
- 8) LCD displays
- 9) PWM (Pulse Width Modulation)
- 10) Timers

14.2 An Example Development Scenario

Suppose you have been given the task of implementing a SPI serial interface between a Microchip PIC16C74, National COP8SAA and a Motorola 68HC05C8. You have only programmed for the Microchip PIC and you are not familiar with SCI serial interfaces. You could learn how SPI works, find out how it is implemented on the different chips, learn how to code for the different chips, write drivers for each chip, and then finally debug the hardware. This development process could take a very long time! By drawing on a portable library for the SPI you can write C code using library functions and avoid delays in project development.

14.2.1 How SPI Works

SPI is a synchronous a three wire serial communications interface based on a master/slave relationship. The master and slave both contain serial shift registers that are connected to form a circular shift buffer. The master supplies the clock which is used to shift data out of the master and into the slave and simultaneously out of the slave and into the master.

SPI is implemented in many different ways, but the same basic functionality holds for each implementation. For example, the COP8SAA7 has a **Microwire Plus** serial interface which SPI compatible. The 68HC05C8 contains the SIOP serial interface which is also SPI compatible. The Microchip PIC16C74 calls its SPI device SPI. Each of these devices has specific names and techniques for SPI serial communication. Using portable libraries we can avoid the confusion involved in using many different device-specific routines.

The following C program performs master functions. The example shown is configured for the Microchip PIC16C74:

```
#define NOLONG //unique to the MPC series of header files
#define REC_SIZE 5
//Use the proper header and driver for the COP8 and 68HC05
#include "16c74.h"
#include "SPI.MPC"

char SPI_in[REC_SIZE];
const char o[] = {0b10000001,0b10000010,0b01000100,
                  0b00001000, 0b00010000};

void main(void){
    SPI_array_get(SPI_in); //set the array to store data in
    SPI_array_send(o); //sets the array to send data from
    //The following statement configures the SPI
    //The argument that is passed depends on the desired
    //configuration. The instructions on how to set this
    //are found in the device driver headers
    SPI_set_master(0b00100000);
    SPI_flush(); //send a byte to get everything synched
    SPI_send_rec(0,4); //initiate the send/receive function

    while(1){
    }
}
```

Example 120: Master function for PIC16C74 SPI communication

The master source code in **Example 120** can be compiled for different chips with very minor changes and the library calls would work as expected.

The library calls are those which begin with the letters SPI such as `SPI_array_get(SPI_in), SPI_array_send(o);`, `SPI_set_master(0b00100000);`, `SPI_flush();` and `SPI_send_rec(0,4);`. We will now examine some of these functions in detail by looking at excerpts from specific device libraries.

14.2.2 SPI_set_master(ARGUMENT);

This function configures the SPI. The following sections describe how it is implemented in the libraries for the individual chips.

On the Microchip PIC16C74

The SPI functions may be used when the synchronous serial port on the Microchip PIC is configured in SPI mode. You must configure the SSPCON register when using SPI. The SSPCON set up for SPI is:

SSPM<3:0>

- 0000 SPI master clock = $osc/4$
- 0001 SPI master clock = $osc/16$
- 0010 SPI master clock = $osc/64$
- 0011 SPI master clock = $TMR2_output/2$
- 0100 SPI slave mode, clock = SCK pin, SS pin control enabled
- 0101 SPI slave mode, clock = SCK pin, SS pin control disabled, SS can be used as I/O pin

CKP<4>

- 1 Transmit on falling edge, receive on rising edge. Idle clock is high
- 0 Transmit on rising edge, receive on falling edge. Idle clock is low

SSPEN<5>

- 1 Enable serial port, configure SCK, SDO and SDI as serial port pins
- 0 Disable serial port, configure pins as I/O

SSPOV<6>

- 1 A new byte is received while SSPBUF register still holds previous data. If an overflow occurs the data in SSPSR is lost. Overflow can only occur in slave mode. The user must read SSPBUF to avoid setting the overflow. In master mode the overflow bit is not set since each new reception and transmission is enacted by writing to SSPBUF

WCOL<7>

- 1 The SSPBUF register is written while transmitting the previous word. Must be cleared in software.
- 0 No Collision

```

/*=====
This function configures the SPI and sets up the proper
pins for serial port operation.
ARGUMENTS:
temp, The byte to set the SPI
=====*/

void SPI_set_master(registerw temp){
    SSPCON = temp;
    TRISC.SDI = 1; //configure TRIS register for serial
    TRISC.SDO = 0;
    TRISC.SCK = 0;
}

```

Example 121: Setting up the SPI on the Microchip PIC16C74**On the Motorola 68HC05**

```

/*=====
This function configures the SPI and sets up the proper
pins for serial port master mode operation.
ARGUMENTS:
NONE
=====*/

#define SIOP_set_master() SCR.SPE = 1; SCR.MSTR = 1;

//this is used to create a uniform interface
#define SPI set master(ARG) SIOP set master()

```

Example 122: Setting up SPI on the Motorola 68HC705C8**On the National Semiconductor COP8**

```

/*****
This function sets the MW in master mode and sets the SK
clock time and sets the SO and SK pins on port G.
ARGUMENTS: ARG1.0 = CNTRL.SL0, ARG1.1 = CNTRL.SL1
CONFIGURATION      SK Cycle Time
CNTRL.SL0=0 CNTRL.SL1=0  2Tc
CNTRL.SL0=0 CNTRL.SL1=1  4Tc
CNTRL.SL0=1 CNTRL.SL1=X  8Tc
*****/

```

```
#define MW_set_master(ARGL) {CNTRL.SL0= ARG & 0b00000001;\
                             CNTRL.SL1= 0b00000010 & ARG;\
                             master();}

void master(void){
    PORTGC.4 =1;
    PORTGC.5 =1;
    PORTGC.6 = 0;
    PORTGD.6 = 1;
    CNTRL.MSEL = 1;
}
//an alias to create a uniform library
#define SPI_set_master(ARG) MW_set_master(ARG)
```

Example 123: Setting up SPI on the National COP8SAA7

14.2.3 SPI_send_rec(0,4);

This function initiates the send/receive function. The following sections show the device specific functions. The function starts at array index 0 of the receive and transmit arrays and transfers information up to index 4. With SPI information is received and transmitted at the same time.

On the Microchip PIC16C74

```
/*=====
This function sends several data bytes from  ARRAY_SEND
and places the contents in the ARRAY_GET array. This
function uses polling. This function assumes that the
returned data is important and stores it in an array

ARGUMENTS:
ARG2 is a pointer to the array or data you wish to send
ARG3 n is the array index to start from
ARG4 offset is the array index to go up to
=====*/

#define SPI_send_rec(ARG2, ARG3, ARG4) \
        SPI_array_send(ARG2); \
        SPI_send_rec2(ARG3, ARG4);

void SPI_send_rec2(n, offset){
    offset = offset+1;
    ARRAY_SEND = ARRAY_SEND + n;
    while(n != offset){
        SSPBUF = *ARRAY_SEND; //SPI_out[n]; // load SSPBUF
        // wait for the BF flag to indicate
        // transmission is done
        while(SSPSTAT.BF == 0){
        }
    }
}
```

```

        *(ARRAY_GET+n) = SSPBUF; //store returned byte
        ARRAY_SEND = ARRAY_SEND + 1 ;
        n=n+1;
    }
}

```

Example 124: Initiating SPI send/receive on the Microchip PIC16C74

On the Motorola 68HC05

```

/*=====
This function sends several data bytes from ARRAY_SEND
and places the contents in the ARRAY_GET array. This
function uses polling. This function assumes that the
returned data is important and stores it in an array
ARGUMENTS:
n, ARG3 is the array index to start from
offset, ARG4 is the array index to go up to
ARG2 is the array you wish to send from
=====*/

#define SIOP_send_rec(ARG2,ARG3,ARG4)\
        SPI_array_send(ARG2); \
        SPI_send_rec2(ARG3, ARG4);

void SIOP_send_rec2(n, offset){
    offset = offset+1;
    ARRAY_SEND = ARRAY_SEND + n;

    while(n != offset){
        SDR = *ARRAY_SEND;//
        SPI_out[n]; // load the SSPBUFF
        //SPIF flag indicates transmission is done
        while(SSR.SPIF == 0){
        }
        *(ARRAY_GET+n) = SDR; //store the returned byte
        ARRAY_SEND = ARRAY_SEND + 1 ;
        n=n+1;
    }
}
//note the use of an alias!
#define SPI_send_rec(ARG2,ARG3,ARG4) \
        SIOP_send_rec(ARG2,ARG3,ARG4)

```

Example 125: Initiating SPI send/receive on the Motorola 68HC705C8

On the National Semiconductor COP8

```

/*=====

```

This function sends several data bytes from `ARRAY_SEND` and places the contents in the `ARRAY_GET` array. This function uses polling. This function assumes that the returned data is important and stores it in an array

ARGUMENTS:

`n` is the array index to start from

`offset` is the array index to go up to

=====*/

```
#define MW_send_rec(ARG2, ARG3, ARG4) \
    MW_array_send(ARG2); \
    MW_send_rec2(ARG3, ARG4);

void MW_send_rec2(n, offset){
    offset = offset+1;
    ARRAY_SEND = ARRAY_SEND + n;
    while(n != offset){
        SIOR = *ARRAY_SEND; // load the SIOR
        PSW.BUSY = 1;
        //BUSY flag to indicate transmission done
        while(PSW.BUSY == 1){
            // transmission is complete,
        }
        *(ARRAY_GET+n) = SIOR; //store returned byte
        ARRAY_SEND = ARRAY_SEND + 1 ;
        n=n+1;
    }
}
//note the use of an alias!
#define SPI_send_rec(ARG2, ARG3, ARG4) \
    MW_send_rec(ARG2, ARG3, ARG4)
```

Example 126: Initiating SPI send/receive on the National COP8SAA7

14.3 Device Driver Library Summary

As we can see from the individual functions, the library prevents the user from having to know the specific hardware configuration of each processor. In particular, the use of aliases allows the user to refer to the functions in the most familiar way possible. One user might be most familiar with the Microchip PIC and wish to refer to the functions as SPI. However, another user might be most familiar with the National COP8 and wish to refer to the functions as MW (Microwire).

15. Sample Project

This section covers a sample embedded system project. The project interfaces a microcontroller with a SPI (UART) peripheral to a PC via the RS-232 port. The most common and easiest technique for interfacing to a PC is to use the parallel port where there are eight parallel bits for input and output. However, it is very easy to damage the parallel port. On PCs with the parallel port on the motherboard a damaged parallel port can require a new motherboard.

The serial port is more complicated but it is a much better tool for interfacing to a desktop PC. It is very difficult to damage your computer by manipulating the serial port. Also, the hardware is almost universally standard. Once you build an embedded system with RS-232 support you can hook it up to a PC, MAC, or another embedded system merely by changing the interface software

This project will introduce some key embedded system programming concepts such as interrupts, registers, and peripherals.

15.1 Project Specifics

The project uses the portable device driver libraries discussed in Section 14, Libraries. The specific hardware implementation will be on a Microchip PIC 16C74. The code is written using Borland C functions. If you do not use Borland these functions are most likely supported by your favourite compiler, where they may have slightly different names.

15.2 Project Foundations

The concepts and terminology necessary for this project are discussed in the following sections.

15.2.1 Asynchronous

Devices that are synchronized in the electronics world use the same clock and their timing is in synchronization with each other. Things that are asynchronous have their own timing and clocks. In the world of serial communications it is easy to tell if something is synchronous or not: if there is a clock line it is synchronous, if there is no clock line it is asynchronous.

15.2.2 SCI

SCI is an asynchronous serial interface also known as UART (Universal Asynchronous Receiver Transmitter). You may also see chips with a USART or SPI with synchronous modes, this is still fundamentally the same as the SCI interface but with the additional option of selecting a synchronous interface. The timing of this signal is compatible with the RS-232 serial standard but the electrical specifications are not compatible and will require a transceiver.

15.2.3 RS-232

Computers like to operate with parallel data. Serial transfers occur by transferring parallel data to serial and then transferring it back into parallel data. There is a component called a **shift register** that can perform these transformations. The shift register uses an internally generated clock to shift data in and out. It can shift in serial and shift out parallel or it can shift in parallel and shift out serial data.

How do the receiver and the transmitter keep the same clock rate? The answer is that they both agree ahead of time on a **baud** rate. The baud rate is the number of times per second that the serial port changes its state. The receiver and transmitter must use the same baud rate.

In order to explain how the receiver and transmitter stay synchronised we must examine a typical RS-232 signal which represents the byte 01010011 in serial format:

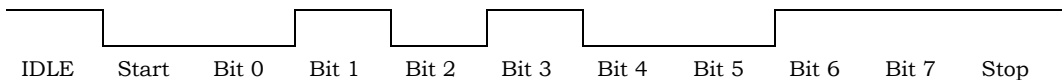


Figure 10: RS-232 signal

An idle serial line going from high to low is a signal to start receiving data. By using the baud rate, the receiver knows exactly how long each bit will be, so it can distinguish bits from each other. After 8 bits are received the line goes high again and the receiver waits for the next start bit. After a byte has been received, it can be taken from the serial port receive register and used by the computer.

The transmitter hardware handles the start and stop bits. Usually all we have to do is load up the serial port transmit register and wait for an interrupt or use device polling to determine when the transmission is complete.

15.3 Electrical Specifications

We mentioned that we need a transceiver to connect the PIC17C74 to an RS-232 serial port. This is because the RS-232 standard specifies voltages that are much different from the 0-5 volts typically used by microcontrollers. RS-232 uses what is called a push-pull system to transmit information. This push pull ranges from sending a 1 (called a mark) as -3 to -25 volts and a 0 (called a space) as +3 to +25 volts. These voltages allow for less distortion and longer cable lengths.

15.4 PIC Implementation

The PIC 16C74 contains a hardware SPI port that lets us transmit asynchronous serial data. We can be notified by interrupt or by polling when the chip has finished sending or receiving a byte. We will examine the serial port on the PC in detail.

15.4.1 Anatomy of a PC serial port

The concept of memory mapped peripherals on microcontrollers was discussed in **Section 2 Microcontroller Overview** and **Section 3 The Embedded Environment**. You will recall that the input and output devices are accessed as memory locations. The PC works exactly the same way. It has four serial ports known as:

NAME	ADDRESS	IRQ
COM1	0x03F8	4
COM2	0x02F8	3
COM3	0x03E8	4
COM4	0x02E8	3

Table 10: PC serial port addresses and interrupts

The next time your PC boots, examine the screen which contains the BIOS information. The BIOS will tell you what serial ports you have and will display their hex address. The column “IRQ” in **Table 10** is the name of the interrupt that is associated with the port.

15.4.2 A Note On Chip Sets

Depending on the vintage of your computer you may have any one of the following UART chips:

CHIP	COMMENTS
8250,8250A, 8250B	These were the first UARTS.
16450, 165501, 6550A	These are what the majority of you will have. 16450 was used in AT's but is still quite common. The 16550 had some problems and was replaced by the 16650A which has a 16 byte FIFO
16650	The newest UART.

Table 11: UART chips

What is a FIFO?

A FIFO is a buffer. FIFO stands for First In First Out. A UART with a FIFO can store data and therefore does not have to interrupt the CPU as often because it can transfer many bytes at each interrupt service.

The variety of UART chips does not affect software development a great deal. The UART chips are all supersets of previous UART's. Unless you are interested in super high performance communications, you can program these chips in exactly the same way. Of course, if you run code for FIFO chips on FIFOless chips the FIFO will not be working. For reasons of simplicity and portability the code in this book will not use a FIFO.

15.4.3 IRQ

Everything you know about interrupts from embedded systems holds true for larger computers. However, the memory address range is much bigger so vectors will be several bytes.

The original PC was designed with 256 interrupt vectors for both hardware and software. These were each 4 bytes in length for a total of 1024 (256×4) bytes in memory. As a whole this areas of memory is called the interrupt vector table. For example, INT 0 uses memory locations 0x00000, 0x00001, 0x00002 and 0x00003 while INT8 uses the four bytes at 0x0020, 0x0021, 0x0022 and 0x0023.

Eight hardware interrupts beginning at INT8 are reserved. They are called IRQ0-IRQ7, thus IRQ0 corresponds to INT8, IRQ1 to INT9 and so on.

Now that we know about the vector table we have to examine a few other registers:

Address BASE +	Read/Write	Abbreviation	Name
0 (DLAB = 0)	W		Transmit Holding Buffer
0 (DLAB = 0)	R		Receiver Buffer
0 (DLAB = 1)	R/W		Divisor Latch Low Byte
1 (DLAB = 0)	R/W		Divisor Latch High Byte
1 (DLAB = 1)	R/W	IER	Interrupt Enable Register
2	R	IIR	Interrupt Identification Register
2	W	FCR	FIFO Control Register
3	R/W	LCR	Line Control Register
4	R/W	MCR	Modem Control Register
5	R/W	LSR	Line Status Register
6	R	MSR	Modem Status Register
7	R/W		Scratch Register

Table 12: COM port registers

The table depicts the registers associated with each COM port. The registers are located at the base port address plus an offset. For example, the Line Status Register for COM1 is at $0x03FD = (0x03F8 + 5)$. The DLAB bit is similar to a paging bit, it allows the access of different registers at the same address. For example, to access the IER set the DLAB bit and access BASE +1.

The following paragraphs describe each register:

Transmit Holding Buffer

Used to read a byte off the UART.

Receive Holding Buffer

A write to the receive holding buffer is used to transmit a byte on the UART.

Divisor Latch High/Low

These two registers allow us to select a baud rate. On the UART there is a 1.8432 Mhz crystal, which the UART divides by 16. If we used this frequency the baud rate would be 115200 hertz. This rate is too fast to communicate with 300 BAUD modems. To get different speeds we can write a 16 bit number to

the Divisor Latch Low/High registers and the baud rate is changed to 115200 / Divisor. For example, for a 2400 BAUD rate, we want the divisor to be 48 ($115200/2400 = 48$). We write 48 (or 0x30) into these two registers by placing 0x00 in the high byte register and 0x30 in the low byte register.

Interrupt Enable Register

Bit Number	Description
BIT 0	Enable Received Data Available Interrupt If we set this bit, the UART will issue an interrupt when received data is available. NOTE: if you have a 16550 or newer UART this enables FIFO time out interrupts
BIT 1	Enable Transmitter Holding Register Empty Interrupt. This will interrupt when the transmit register is empty.
BIT 2	Enable Receiver Line Status Interrupt: Not used in this Project
BIT 3	Enable Modem Status Interrupt: Not used in this Project
BIT 4	RESERVED
BIT 5	RESERVED
BIT 6	RESERVED
BIT 7	RESERVED

Table 13: Interrupt enable register bits

Interrupt Identification Register

Bit Number		Description
BIT 0		0 Interrupt Pending 1 No Interrupt Pending
BIT 1	BIT 2	
0	0	Modem Status Interrupt
1	0	Transmitter Holding Register Empty Interrupt
0	1	Received Data Available Interrupt
1	1	Receiver Line Status Interrupt
BIT 3		0 Reserved on 8250 16450 1 16550 Time-out Interrupt Pending
BIT 4		Reserved
BIT 5		Reserved
BIT 6	BIT 7	
0	0	No FIFO
0	1	FIFO Enabled but Unusable
1	1	FIFO Enabled

Table 14: Interrupt identification register

FIFO Control Register

Bit Number		Description
BIT 0		Enable FIFO - turn on the FIFO.
BIT 1		Clear Receive FIFO - erase the receive buffer
BIT 2		Clear Transmit FIFO - erase the Transmit Buffer
BIT 3		DMA mode select - Not used by this project
BIT 4		Reserved
BIT 5		Reserved
BIT 6	BIT 7	Interrupt Trigger Level - Only if chip has a FIFO
0	0	1 Byte
0	1	4 Bytes
1	0	8 Bytes
1	1	14 Bytes

Table 15: FIFO control register

Line Control Register

Bit Number			Description
BIT 1	BIT 0		Word Length - select how many bits to send in each message.
0	0		5 bits
0	1		6 bits
1	0		7 bits
1	1		8 bits
BIT 2			Length of Stop Bit
	0		One Stop Bit
	1		2 Stop bits for words of length 6,7,8 bits or 1.5 Stop bits for word lengths of 5 bits
BIT 3	BIT 4	BIT 5	Parity Select
0	DC	DC	None
1	0	0	Odd
1	1	0	Even
1	0	1	High Sticky
1	1	1	Low Sticky
BIT 6	Set Break Enable		
BIT 7	1		Divisor Latch Access Bit - DLAB Remember this guy
	0		Access to Receive

Table 16: Line Control Register

Modem Control Register

Bit Number	Description
BIT 0	Reserved
BIT 1	Reserved
BIT 2	Reserved
BIT 3	Loop Back Mode
BIT 4	Aux Output 2
BIT 5	Aux Input 1
BIT 6	Force Request to Send
BIT 7	Force Data Terminal Ready

Table 17: Modem Control Register

Line Status Register

Bit Number	Description
BIT 0	Error in Received FIFO
BIT 1	Empty Data Holding Registers
BIT 2	Empty Transmitter Holding register
BIT 3	Break Interrupt
BIT 4	Framing Error
BIT 5	Parity Error
BIT 6	Overrun Error
BIT 7	Data Ready

Table 18: Line Status Register

Modem Status Register

Bit Number	Description
BIT 0	Carrier Detect
BIT 1	Ring Indicator
BIT 2	Data Set Ready
BIT 3	Clear To Send
BIT 4	Delta Data Carrier Detect
BIT 5	Trailing Edge Ring Indicator
BIT 6	Delta Data Set Ready
BIT 7	Delta Clear to Send

Table 19: Modem Status Register

15.5 Programming Interrupts

The serial port has two associated IRQs, IRQ3 and IRQ4. To refer to these in a program we must refer to them by interrupt vector table entry: 0x0B for IRQ3 and 0x0C for IRQ4. (IRQ0 is located at 0x08). There are two useful macros provided by Borland called `enable()` and `disable()`. These functions enable and disable all interrupts which is useful for when we are carrying out interrupt related programming and do not wish our program to be interrupted by other interrupt service requests.

Good programming practice dictates that we return the contents of the vector location when our program is finished. If we do not, programs which use the serial port may not run because they will be directed to our interrupt service routine.

Programming interrupts on a PC is much like programming them on an embedded system. There is a definite series of steps one must use:

① Change the vector location

First declare a pointer to an interrupt function:

```
void interrupt far (*old_function)();
```

Store the old vector address for our interrupt, because we must restore this when we are done. For example if we are using COM1, `IRQ_location` will be 0x0C.

```
old_function = getvect(IRQ_location);
```

Now we can place the location of our interrupt service routine in the vector table:

```
setvect(IRQ_location, our_int_serv);
```

If we were using COM1, `IRQ_location` is 0x0C and `our_int_serv` is the name of the function that we have written for interrupt service.

Finally we reset the old vector to the table:

```
setvect(IRQ_location, old_function);
```

② Unmask the Interrupt

The interrupt is unmasked by clearing the bit corresponding to our IRQ in location 0x021.

```
//leave other bits alone and change our bit  
outportb(0x021, inportb(0x021 & ~IRQ_location);
```

③ Write an Interrupt Service Routine

Now we can write our ISR. The routine can perform any action we wish. For example, the computer can log a time, log data, or emit a beep. The only condition is that we must tell the computer that the interrupt has been processed. This is easily done using the following call:

```
outportb(0x20, 0x20);
```

```
//this tells the computer that the interrupt has been
//processed
```

A general format for ISR is as follows:

- 1) Disable any further interrupts with `disable()` ;
- 2) Do what ever is required
- 3) Indicate that the interrupt was processed to the PC
- 4) Enable interrupts with `enable()` ;
- 5) Return the interrupt mask to its original configuration (set the bit corresponding to our interrupt).

NOTE

You may encounter problems attempting to perform some operations in an interrupt service routine. For instance if you try to write data to disk, the system may hang because the disk drive is trying to use an interrupt but is unable to do so.

15.6 The Sample Project Code

The following is a simple implementation of a serial port connection, the PC sends the message `Hello Mr. PIC!` and the PIC16C74 sends the message `Hello Mr. PC!`. On both the PIC16C74 and the PC the string is stored in an array. The PIC16C74 will initiate the transfer. The PIC will take advantage of a portable device driver library. Both the PIC and the PC are configured to run at 9600 baud with 8 bits of data, no parity, and 1 stop bit.

15.6.1 PIC16C74 Code

```
#pragma option v;
#define NOLONG

//header files contain memory and port declarations,
//and device library functions

#include "16c74.h"
#include "port.mpc"
#include "MPCsci2.h"

const char cout[] = "Hello, Mr. PC!";
char cin[15];

void main(void){
```

```
        SCI_array_get(cin); // The array to store the data
        SCI_setup(0x019); // Set up the SCI port on the PIC
        SCI_string_int(cout); // Transmit a string using int.

        while(1){
        }

}

void __INT(void){
    INTCON.GIE = 0;
    if(PIR1.TXIF == 1){
        SCI_int_svcst(); // macro from the device library
    }
    RestoreContext;
    return;
}
```

Example 127: Serial port connection example for the PIC16C74

15.6.2 PC Code

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define PORT1 0x03f8 //we want to use COM 1
#define COM1 0x03f8
#define IRQ 0x0C //the INT for COM1 (IRQ3)

unsigned char chout[] = "Hello Mr. PIC";
unsigned char chin[15]; //store the received message here

void interrupt far (*oldfunc)(...); // store the old ISR

int gn; // this is a global counter, to tell us how many
        //times the interrupt has been serviced
int int_done = 0;

// our interrupt service routine
void interrupt int_svc(...){
    unsigned char c;
    disable(); //turn off interrupts
    // tell the PC that the interrupt has been serviced

    outportb(0x20,0x20);
    //if the first bit of the LSR is set, it means
    //that the UART has received information

    do { c = inportb(PORT1 + 5);
        if (c & 1){ //so we get it!
            chin[gn] = inportb(PORT1);
            printf("%c", chin[gn]); // print message
```

```
        }
    }while (c == 1);
    int_done = 1;
    gn = gn + 1;
    enable(); // turn back on the interrupts
}

int main(void){
    int n = 0;
    outportb(PORT1 + 1 , 0); //Turn off COM1 interrupts
    disable(); //Borland macro to turn off all interrupts
    oldfunc = getvect(IRQ); //store old interrupt vector
    setvect(IRQ, int_svc ); // Set new interrupt vector

    //We must now configure the serial port, refer to the
    //charts at the beginning of the book to determine
    //what is being set. We will configure for 9600 baud,
    //8 bit words, 1 stop bit, and no parity bit. The
    //FIFO (if it exists) is set to one byte.

    //Communication Settings
    outportb(PORT1 + 3 , 0x80); // SET DLAB ON
    outportb(PORT1 + 0 , 0x0C); // Set Baud rate Low Byte
    outportb(PORT1 + 1 , 0x00); // Set Baud rate Hi Byte
    outportb(PORT1 + 3, 0x00); // The DLAB is zero

    // 8 Bits, No Parity, 1 Stop Bit
    outportb(PORT1 + 3 , 0x03);
    outportb(PORT1 + 2 , 0x07); // FIFO Control Register
    outportb(PORT1 + 4 , 0x0B); // Turn on DTR, RTS, OUT2

    // Interrupt when data received
    outportb(PORT1 + 1 , 0x01);

    // Set Programmable Interrupt Controller,
    // i.e. unmask our interrupt
    outportb(0x21, (inportb(0x21) & 0xEf));

    enable(); //Borland macro to turn on interrupts
    while(n<=15){
        outportb(PORT1, chout[n]);
        while(int_done == 0)
        {}
        //output our message after receiving a byte
        outportb(PORT1, chout[n]);
        int_done = 0;
        n=n+1;
    }
    disable();
    //set mask bit
    outportb(0x21, (inportb(0x21) | 0x10));
```

```

    setvect(IRQ,oldfunc); //restore interrupt setting
    enable();
    printf("\nThe received string is \n %c\n" , chin);
    return 0;
}

```

Example 128: Serial port connection example for the PC

Now all that we require is the hardware necessary to turn the electrical signals from the PIC into RS-232 levels. This is quite easy and there are a number of chips that can turn 5 volt TTL levels into RS-232 levels off of a standard 5 volt power supply. They use a “bucket brigade” of capacitors to build the needed potential difference. This project uses a MAX232A, but you can use any of the many alternatives as long as you follow the schematic:

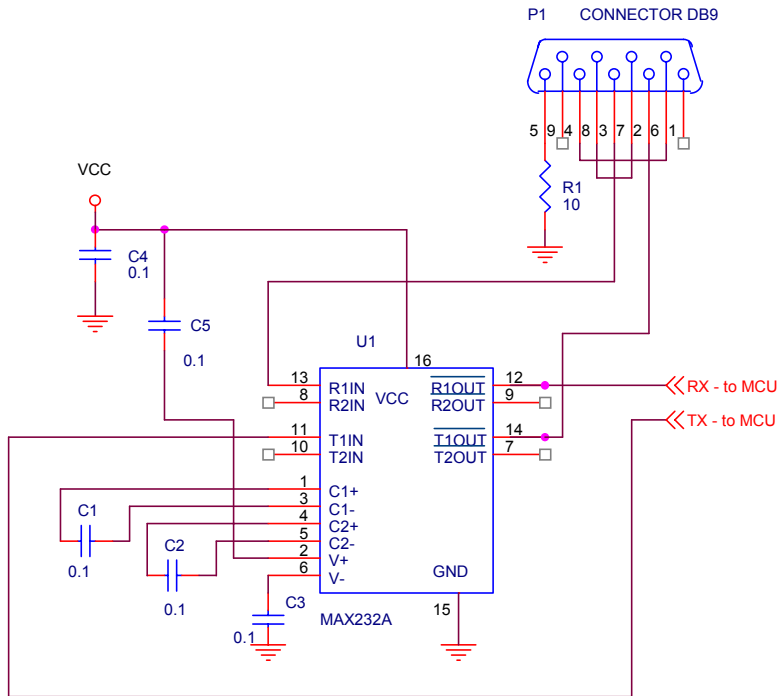


Figure 11: Project schematic

It is useful to examine the pin outs of the RS232 port:

CONNECTOR PIN #		NAME
DB9	DB25	
3	2	TD -Transmit Data
2	3	RD - Receive Data
5	7	SG - Signal Ground
4	20	DTR - Data Terminal Ready
6	6	DSR - Data Set Ready
1	8	CD - Carrier Detect
7	4	RTS - Request to Send
8	5	CTS - Clear to Send

Table 20: Pin outs on the RS232 port

There are two types of serial connectors: the DB9 has 9 pins and the DB25 has 25 pins. The pins are usually marked on the connector so it is easily determined which pins are which. The interface to the microcontroller can be thought of as a DCE, or Data Communications Equipment, and therefore needs only a straight through cable.

The RS-232 protocol defines two types of devices, DTE, Data Terminal Equipment and DCE, Data Communication Equipment. DTE is generally used with PCs and DCE is usually found on modems. The pins, DTR, DSR, CD, RTS, and CTS are only useful with a modem, i.e. connecting a DTE to a DCE. We will just loop these pins back and trick the PC into thinking it is talking to a modem. This way data can flow freely on the TX and RX pins.

NOTE

Ensure that you connect the ground on both parts of the circuit together or it will not work because the electrical signals will not be able to complete a circuit.

You can make many interesting and fun projects using the serial port. The project described in this section can be very useful even if you do not have any micro controllers to interface with. It is very useful for learning key embedded programming concepts like interrupts, registers, serial communications, and timing. You can easily hook up two PCs to transfer files and run dumb terminals. If you connect two PC's together remember, that you are connecting two DTEs, which will require a null modem cable.

16. C Precedence Rules

Expression type	Operators		
Primary	Identifier		
	Constant		
	String		
	Expression		
Postfix	a[b]	f()	a.b
	a--	a++	
Unary	++a	--a	
	sizeof a	sizeof(a)	
	&a	*a	~a
	!a	+a	-a
Cast	(type) a		
Multiplicative	a * b	a / b	a % b
Additive	a + b	a - b	
Shift	a << b	a >> b	
Relational	a < b	a > b	
	a <= b	a >= b	
	a == b	a != b	
Bit AND	a & b		
Bit EOR	a ^ b		
Bit OR	a b		
Logical AND	a && b		
Logical OR	a b		
Conditional	a ? b : c		
Assignment	a = b		
	a += b	a -= b	
	a *= b	a /= b	a %= b
	a &= b	a ^= b	a = b
	a <<= b	a >>= b	
Comma	a, b		

Table 21: Rules of operator precedence

Operations higher up in the table have precedence over those lower. Those at the same level execute in the order they appear. The optimizer often regroups sub-expressions that are both associative and commutative in order to improve the efficiency of generated code. The order of any side-effects, such as

assignment, or action taken by a function call, is also subject to alteration by the compiler.

17. ASCII Chart

HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	\$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Table 22: ASCII characters

18. Glossary

accumulator

Also **AC**, **ACC**. A register which holds the resulting values of ALU operations.

a/d

Analog to digital.

address

A number which indicates the storage location of data in memory.

addressing mode

The syntax used to describe a memory location to the CPU.

algorithm

A solution to a problem.

ALU

Arithmetic Logic Unit. Performs basic mathematical manipulations such as add, subtract, complement, negate, AND, OR.

analog

A continuous range of voltage values.

AND

Logical operation where the result is 1 iff ANDed terms both have the value 1.

ANSI C

American National Standards Institute standards for C language.

array

A group of data elements indexed and stored in contiguous memory.

ASCII

The American Standard Code for Information Interchange is used to represent characters.

assembler

Program that converts a machine's assembly language into object code.

assembly language

Mnemonic form of a specific machine language.

assignment

Store a value in a variable.

asynchronous

Unclocked or not synchronous with CPU timing.

bank

A logical unit of memory (64k).

baud

The number of bits transmitted per second

binary

Base 2 number system which contains only the numbers 0 and 1.

bit

Binary digit which is either 0 or 1.

bit field

A group of contiguous bits considered as a unit.

block

Any section of C code enclosed by braces {}.

breakpoint

A set location to stop executing program code. Breakpoints are used in debugging programs.

bus

Path for signals between components of a computer system .

byte

Eight bits.

C

High level programming language.

cast

Also **Coerce**. Convert a variable from one type to another.

checksum

A value which is the result of adding specific binary values. A checksum is often used to verify the integrity of a sequence of binary numbers.

clear

Set a bit to 0.

clock

Fixed-frequency signal that triggers or synchronizes CPU operation and events. A clock has a frequency which describes its rate of oscillation in MHz.

comment

Non-executed text included in a program in order to explain what the executable statements in the program are doing.

compiler

Program that converts a high level language to object code.

computer operating properly (COP)**control statement**

Statement which controls the execution of other statements based on conditions provided by the programmer.

CPU

Central Processing Unit. It fetches, decodes and executes instructions.

cross assembler

An assembler that runs on one type of computer and assembles the source code for a different target computer. For example, an assembler that runs on a 80486 and generates object code for Motorola's 68HC05.

cross compiler

A compiler that runs on one type of computer and compiles source code for a different target computer. For example, a compiler that runs on a 80486 and generates object code for Motorola's 68HC05.

crystal

A quartz crystal which provides a frequency for clock timing.

debugger

A program which helps with system debugging where program errors are found and repaired. Debuggers support such features as breakpoints, dumping, memory modify.

decision statement

Statement which controls the program flow based on the result of testing a condition.

declaration

A specification of the type, name and possibly the value of a variable.

decoder

The unit which decodes bits into mutually exclusive outputs.

dereference

Also *. Access the value pointed to by a pointer.

directive

A command given to the preprocessor which begins with a #.

EEPROM

Electrically erasable programmable read only memory.

embedded

Fixed within a surrounding system or unit.

escape character

The `\` character in C can be used as an escape character.

executable

A file which contains code which can be run on a specific target device.

fixed point

Integer representation where the decimal is in a fixed position.

floating point

The integer representation of decimal numbers using a mantissa field and an exponent field.

global variable

Variable that can be read or modified by any part of a program.

header file

Source code which is inserted into another source file using the `#include` preprocessor directive.

hexadecimal

Also Hex. Base 16 numbering system which uses the digits 0-9 and the letters A-F.

include file

A file which is included by the preprocessor due to the use of the `#include` directive.

index register

Also X. Register used to hold an increment which can be added to an address when indirect addressing is used.

integer

A number with no decimal, a whole number.

interrupt

A signal sent to the CPU to request service. The CPU saves its state and branches to a routine to handle the interrupt. After the interrupt has been handled the saved state is restored.

library

Collection of functions which are available for use by other programs.

linker

A program which combines separate object files together in order to create an executable file.

local variable

Variable that can only be used by a specific module or modules in a program.

logical operator

Operators which perform logical operations on their operands. For example, !, &&, ||.

machine language

Binary code instructions which can be understood by a specific CPU.

macro

Source code which is given a unique label. If the compiler sees the label in following source code it will replace it with the body of the macro.

mask

A group of bits designed to set or clear specific locations in another group of bits when used with a logical operator.

maskable interrupt

Interrupts which software can activate and deactivate.

memory mapped

A virtual address or device is associated with an actual address in memory.

microcontroller

Also **MCU**. Single chip which controls another device and contains a CPU, memory and I/O ability. A type of embedded controller.

microprocessor

Also **μP**. A single chip CPU.

module

A logically united part of a program which is in the same source code file.

nibble

A four bit binary number.

NOP

No operation. An instruction which is used to create a delay.

not

Logical negation. A 0 becomes a 1 and a 1 becomes a 0.

object code

Machine language instructions represented by binary numbers not in executable form. Object files are linked together to produce executable files.

octal

Base 8 number system.

operator

A symbol which represents an operation to be performed on operands. For example, +, *, /.

or

A Boolean operation which yields 1 if any of its operands is a 1.

paging

A page is a logical block of memory. A paged memory system uses a page address and a displacement address to refer to a specific memory location.

parameter

A variable used to pass information to and from a function.

pointer

An address of a specific object in memory which is used to refer to that object.

port

A physical I/O connection.

preprocessor

A program which prepares data for processing by the compiler.

program

Collection of instructions for a computer written in a programming language which implement an algorithm.

program counter

Also **PC**. A register which holds the address of the next instruction to be executed. The program counter is incremented after each instruction is fetched.

PROM

Programmable read-only memory. ROM that can be programmed.

RAM

Random Access Memory. RAM is read/write memory.

real number

A number which can have a decimal place.

real time

A system which reacts at a speed commensurate with the time an actual event occurs.

recursive

A function which calls itself.

register

A byte or word of memory which can be directly accessed by the processor. Registers are accessed more quickly than other memory locations. Some registers are CPU registers which means that they exist within the CPU.

reset

To return to a selected beginning point.

return

An instruction which terminates a function.

ROM

Read Only Memory.

ROMable

Code which will execute when placed in ROM memory.

scope

A variable's scope is the areas of a program in which it can be accessed.

sequencer

A module which provides the next program address to memory.

serial

Sequential transmission of one bit at a time using a single line.

set

Give a bit the value 1.

shift

Move the contents of a register to the left or right.

side-effect

An unintentional change to a variable.

simulator

A program which has the same input and output behaviour as a specific device. Timing considerations can not be tested with a simulator.

source code

A program in assembly language or a high level language before it passes through an assembler or compiler.

stack

A section of RAM which is used to store temporary data. A stack is a last-in-first-out (LIFO) structure which contains information which is saved and restored.

stack pointer

A register which contains the address of the top of the stack.

static

A variable that is stored in a reserved area of RAM instead of in the stack. The area reserved cannot be used by other variables.

synchronous

Operations which are controlled by a clock pulse.

timer**UART**

Universal asynchronous receiver/transmitter. A serial-to-parallel and parallel-to-serial converter.

USART

Universal Synchronous/Asynchronous Receiver/Transmitter. A chip which handles synchronous data communications.

variable

A symbolically named address or range of addresses which can be assigned values.

void

A C data type.

word

A 16 bit binary number.

19. Bibliography

Oualline, Steve. *Practical C Programming*. Sebastopol, CA: O'Reilly & Associates, 1991.

20. Index

!

! · *See* not operator

!= · *See* inequality operator

#

#define · 57

#include · 57

#pragma · 58

&

& operator

precedence · 185

&& · *See* and operator

* operator

precedence · 185

{

{ · *See* braces

<

< · *See* less-than operator

<= · *See* less-than-or-equal

=

= · *See* assignment operator

== · *See* equality operator

>

> · *See* greater-than operator

>= · *See* greater-than-or-equal

A

accumulator · 14

algorithm · 83

ALU · 13

and operator · 93

arithmetic logic unit · *See* ALU

arithmetic operators · 88

ASCII · 71, 76

assembler · 47

assembly language · 46

assignment operator · 87

assignment statement · 60

asynchronous · 16

B

baud rate · 16

binary · 44

binary notation · 79

binary operators · 85

bits · 45

block · 69

braces · 59, 69, 70, 102

bus · 6, 19

C

central processing unit · *See* CPU
character · 71
character data type · 76
 assigning · 76
clock · 11
collating sequence · 77
comma operator · 87
comments · 56
 C++ · 56
compiler · 49, 50, 66, 69
 cross compiler · 51
constant · 67
 defining with `#define` · 57
control statement · 60
control structure · 99
CPU · 6, 19
cross compiler · *See* compiler

D

data abstraction · 75, 113
data type · 71
 character · 76
 double · *See* double data type
 float · 81
 function · 75
 integer · *See* integer data type
 long · *See* long data type
 long double · *See* long double data type
 modifiers · *See* modifiers
 parameter · 76
 short · *See* short data type
dead code · 94
decimal notation · 79
decoder · 13
decrement
 operator · 89
 postfix · 89
 prefix · 90
development platform · 5
directives · *See* preprocessor directives
division
 integer · 88

 operator · 88
double data type · 81
double underscore · 1

E

else statement · 101
 matching with if · 102
emulator · 51
equality operator · 92
equality operators · 91
escape sequence · 77
expression
 evaluation · 84
expressions · 84
 compared to statements · 84

F

floating point numbers · 80
function
 body · 60
 header · 60
 identifier · 68
 prototype · 59
functions · 58, 65

G

GIE · 18
global interrupt enable · *See* GIE
greater-than operator · 93
greater-than-or-equal · 93

H

Harvard architecture · 7
header file · 57, 81
hexadecimal · 46
hexadecimal notation · 79

I

identifier · *See also* variable
identifier
 and significant characters · 67
 constant · 67
 memory allocation · 66
 naming rules · 66
identifiers · 65
if · 61
if statement · 100
 matching with else · 102
increment · 83
 operator · 89
 postfix · 89
 prefix · 90
index register · 14
inequality operator · 92
initialization code · 111
integer data type · 71, 78
 assigning to a float · 81
integer variables · 80
interrupt · 6, 18
 maskable · 18
 non-maskable · 18
intger data type
 sign bit · 78

K

keywords · 66

L

LED · 55
less-than operator · 93
less-than-or-equal · 93
linker · 50
local variables · *See* variables, local
logical operators · 91
long data type · 72, 79
long double data type · 81
loop · 61, 92
 infinite · 61

M

machine code · 73
machine language · 46, 50
main() · 58
maskable interrupt · *See* interrupt
memory
 allocation for variables · 73
microcontroller · 5
 standard · 2
microprocessor · 5
modifiers · 80
module · 57
modulus
 operator · 88

N

nesting
 if statements · 101
not operator · 93

O

octal notation · 79
operator
 binding · 85
 precedence · 86
operator precedence · 185
operators · 83
 arithmetic · 88
 binary · 85
 postfix · 85
 prefix · 85
 ternary · 86
 unary · 85
or operator · 93

P

parameter · 59
parameters · 76

platform
 development · 5
 target · 6
port · 16, 58
precedence
 of operators · 86
preprocessor · 50
preprocessor directives · 56, 65
processor clock · *See* clock
processor oscillator · *See* clock
programmer · 52
prototype · *See* function prototype

R

RAM · 6, 67, 68, 73
random access memory · *See* RAM
read only memory · *See* ROM
readability
 improving · 56, 57
real numbers · *See* floating point
register · 13. *See also* individual registers
relational operators · 91
reserved word · 58
ROM · 6, 68

S

scope · 73
 local · 74
 overlap · 74
semicolon · 69
sequencer · 13
short data type · 78
short-circuit evaluation · 94
side-effects · 90
sign bit · 78, 80
significant characters · 67
simulator · 51
stack pointer · 14
standard
 microcontroller · 2
statement
 terminator · 69

stdio library · 55
symbol table · 66, 73
synchronous · 16

T

target platform · 6
timer · 6, 17
trinary operator · 86
typographical conventions
 courier font · 1
 italic courier font · 1
typographical conventions · 1
 bold · 1

U

unary operators · 85

V

variable
 initializing · 72
 scope · *See* scope
variable · 65
 declarations · 73
 external · 74
 global · 74
 local · 74
 multivariable declaration · 72
variables
 local · 68
vector · *See also* interrupt
visibility · *See* scope
void · 59, 75, 84
Von Neumann architecture · 7

W

watchdog timer · *See* COP
while · 61, 70
white space · 69

Byte Craft Limited

Code Development Systems

CATALOG

05/2001

Byte Craft Limited specializes in embedded systems software development tools for single-chip microcontrollers. Byte Craft Limited was the first company to develop a C compiler for the Motorola 68HC05 and the National Semiconductor COP8™. Our compilers and related development tools are now being used by a wide range of design engineers and manufacturers in areas of Commerce, Industry, Education, and Government.

MPC

Supports all Microchip PIC 12x/14x/16x/17x families, 8K and Flash parts
Named address space supports variable grouping
Works with Microchip's PICMASTER, ICE 2000 emulator, MPLAB-SIM simulator, Advanced Transdata, Tech-Tools Mathias, Clearview, iSystem
Supports setting configuration fuses through C
Demo at www.bytecraft.com/impc.html

for DOS or Windows COP8C

Supports the Feature Family, and SGR/SGE
Supports LOCAL memory reuse, SPECIAL memory through software
Supports SREG memory management
Support for symbolic debugging with emulators including MetaLink
Supports setting configuration fuses through C
Demo at www.bytecraft.com/icop.html

C6805

Supports all 68HC05 variants
Supports LOCAL memory reuse, SPECIAL memory through software
Support for symbolic debugging with many emulators including MMDS05, MMEVS, and Metalink iceMASTER
E6805 available to support Motorola EVM, EYS
Supports setting Mask Option Register through C
Demo at www.bytecraft.com/i05.html

C38

Supports all MELPS740 variants, including 7600 series, M509xx, M371xx, M374xx and M38xxx
Supports MUL, 7600
Supports processor-specific instructions BRK, CLC, CLD, CLI, CLT, CB, NOP PHA, PLA, PLP, ROL, ROR, RRF SEC, SED, SEI, SET, STP, WIT
Allows direct access to AC, X, Y, CC registers
Demo at www.bytecraft.com/ic38.html

C6808

Supports all 68HC08 variants
Supports LOCAL memory reuse, SPECIAL memory through software
Supports 6808 extended addressing, instructions
Support for symbolic debugging with many emulators including Motorola MMDS08 and MMEVS08, and the Ashling CT68HC08
Supports setting Mask Option Register through C
Demo at www.bytecraft.com/i08.html

for DOS or Windows SXC

Supports all SX variants, including SX48 and SX52
Supports LOCAL memory reuse, SPECIAL memory through software
Supports virtual device drivers within C
Data types include bit, bits, char, short, int, int8/16/24/32, long, float and fixed point
Support for assembly source-level debugging with Parallax SX-Key
Demo at www.bytecraft.com/isxc.html

Z8C

Supports all Zilog Z8 and Z8+ variants
Supports instruction set variants C94, C95, HALT, MUL, STOP, WAIT
Supports processor-specific instructions DI, EI, HALT, NOP, RCF SCF, STOP, WAIT, WDT, WDH
Generates information required for source-level debugging
Demo at www.bytecraft.com/iz8c.html

Fuzz-C™

Transforms fuzzy logic to plain C; call between C and fuzzy functions
Accepts fuzzy logic rules, membership functions and consequence functions
Standard defuzzification methods provided; add new defuzzification methods easily
Includes plots of membership and consequence functions in generated comments
Works with all Code Development Systems

Features

Both DOS and Windows versions include an **Integrated Development Environment**. The DOS IDE provides source-level error reporting. The Windows IDE maintains projects, gives access to online help, and can control third-party tools.

The compilers generate tight, fast, and efficient executables, as well as listing files that match the original C source to the code generated. Several optional reports (symbol information, nesting level, register contents) can appear in the listing file.

Header files describe each processor derivative. **#pragma** statements configure the compiler for available interrupts, memory resources, ports, and configuration registers. Convenient **#defines** make your programs portable between members of a processor family.

C extensions include: **bit** and **bits** data types, binary constants, **case** statement extensions, direct register access in C, embedded assembly, initialization control, direct variable placement, interrupt support in C.

Two forms of linking are available: **Absolute Code Mode** links library modules into the executable during compilation. The **BLink linker** uses a more traditional linker command file and object files. Either route provides optimization at final code generation.

You can include **Macro Assembler** instructions within C code, or as separate source files. Embedded assembly code can call C functions and access C variables directly. You can also pass arguments to and from assembly code.

Availability

Byte Craft Limited products are available world-wide, both directly from Byte Craft Limited and through our distributors. Demonstration versions of the Code Development System are available.

For more information, see www.bytecraft.com.

Upgrade Policy

Registered customers receive free upgrades and technical support for the first year. All other **registered users** may purchase major releases for a fraction of the full cost. Along with our version upgrades, Byte Craft Limited remains committed to maintaining a high level of technical support.



Byte Craft Limited
A2-490 Dutton Drive
Waterloo, Ontario
Canada • N2L 6H7
phone: 519-888-6911
fax: 519-746-6751

info@bytecraft.com
www.bytecraft.com

COP8C

C6805

C6808

SXC

Z8C

C38

MPC

About Byte Craft Limited

Byte Craft Limited is a software development company specializing in embedded systems software development tools for single-chip microcomputers. We provide innovative solutions for developers, consultants and manufacturers around the world. Our main products are C cross-compilers targeted to a variety of microcontroller families.

www.bytecraft.com



Byte Craft Limited
A2-490 Dutton Drive
Waterloo, Ontario, Canada
N2L 6H7
phone: +1 519.888.6911
fax : +1 519.746.6751
<info@bytecraft.com>