

# Design of Embedded Systems Using 68HC12/11 Microcontrollers



RICHARD E. HASKELL

# ***Design of Embedded Systems Using 68HC12/11 Microcontrollers***

Richard E. Haskell

Computer Science and Engineering Department  
Oakland University  
Rochester, Michigan 48309



PRENTICE HALL  
Upper Saddle River, NJ 07458

## Library of Congress Cataloging-in-Publication Data

Haskell, Richard E.

Design of embedded systems using 68HC12/11 microcontrollers /

Richard E. Haskell

p. cm.

ISBN 0-13-083208-1 (pbk.)

1. Embedded computer systems—Design and construction.

2. Motorola 68HC11 (Microprocessor)

TK7895.E42H38 1999

004.2'1—dc21

99-16964

CIP

Publisher: *Tom Robbins*

Associate editor: *Alice Dworkin*

Production editor: *Audri Anna Bazlen*

Editor-in-chief: *Marcia Horton*

Executive managing editor: *Vince O'Brien*

Assistant managing editor: *Eileen Clark*

Vice-president of production and manufacturing: *David W. Riccardi*

Art director: *Jayne Conte*

Cover design: *Bruce Kenselaar*

Manufacturing buyer: *Pat Brown*

Marketing manager: *Danny Hoyt*

Editorial assistant: *Dan DePasquale*



© 2000 by Prentice Hall

Prentice-Hall, Inc.

Upper Saddle River, New Jersey 07458

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-083208-1

Prentice-Hall International (UK) Limited, London

Prentice-Hall of Australia Pty. Limited, Sydney

Prentice-Hall Canada Inc., Toronto

Prentice-Hall Hispanoamericana, S.A., Mexico

Prentice-Hall of India Private Limited, New Delhi

Prentice-Hall of Japan, Inc., Tokyo

Prentice-Hall (Singapore) Pte. Ltd., Singapore

Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

***Design of Embedded  
Systems Using 68HC12/11  
Microcontrollers***

*To Edie*

# *Preface*

Many people think of a computer as a PC on a desk with a keyboard and video monitor. However, most of the computers in the world have neither a keyboard nor a video monitor. Rather they are small microcontrollers—a microprocessor, memory, and I/O all on a single chip—that are embedded in a myriad of other products such as automobiles, televisions, VCRs, cameras, copy machines, cellular telephones, vending machines, microwave ovens, medical instruments, and hundreds of additional products of all kinds. This book is about how to program microcontrollers and use them in the design of embedded systems.

A popular microcontroller that has been used in a wide variety of different products is the Motorola 68HC11. Motorola has recently introduced an upgrade of this microcontroller, the 68HC12, that has new, more powerful instructions and addressing modes. This book emphasizes the use of the 68HC12 while at the same time providing information about the 68HC11. It can therefore be used in courses that use both 68HC12 and 68HC11 microcontrollers.

This book is the result of teaching various microcomputer interfacing courses over the past 20 years. While the technology may change, the basic principles of microcomputer interfacing remain largely the same and these basic principles are stressed throughout this book. However, microcomputer interfacing is a subject that is learned only by doing. The courses that I have taught using this material have all

been project-oriented courses in which the students design and build real microcomputer interfacing projects.

A definite trend in microcomputer interfacing and in digital design in general is a shift from hardware design to software design. Microcomputer interfacing has always involved both hardware and software considerations. However, the increasingly large-scale integration of the hardware together with sophisticated software tools for designing hardware means that even traditional hardware design is becoming more and more a software activity.

In the past most software for microcomputer interfacing has been written in assembly language. This means that each time a new and better microprocessor comes out the designer must first learn the new assembly language. The advantage of assembly language is that it is “closest to the hardware” and will allow the user to do exactly what he or she wants in the most efficient manner. While some feel that assembly language programs are more difficult to write and maintain than programs written in a high-level language, the major disadvantage of assembly language programs is related to the obsolescence of the microprocessor—when upgrading to a new or different microprocessor, all of the software has to be rewritten! Even when upgrading from a 68HC11 to a 68HC12, which is upward compatible at the source-code level, to get the best performance from the 68HC12 you will need to rewrite the code to use the newer, more powerful instructions and addressing modes.

This has led to a trend of using high-level languages such as C or C++ for microcomputer interfacing. While this helps to solve the obsolescence problem—much of the same high-level code might be reusable with a new microprocessor—high-level languages come with their own problems. The development environment is not always the most convenient. One has to edit the program, compile it, load it, and then run it to test it on the real hardware. This edit-compile-test cycle can be very time consuming for large programs. Without sophisticated run-time debugging tools the debugging of the program on real hardware can be very frustrating. When designing microcomputer interfaces you would like to be as close to the hardware as possible.

What you would like is a computer language with the advantages of both a high-level language and assembly language, with none of the disadvantages. It would be nice if the language were also interactive so that you could sit at your computer terminal and literally “talk” to the various hardware interfaces. The language should also produce compact code so that you can easily embed the code in PROMS or flash memory for a stand-alone system. While you’re at it why not embed the entire language in your target system so that you can develop your program “on-line” and even upgrade the program in the field once the product is delivered. Impossible, you say? In fact, just such a language exists for almost any microprocessor you may want to use. The language is Forth and we will use a derivative of it in this book to illustrate how easy microcomputer interfacing can be.

We will use a unique version of Forth called *WHYP* (pronounced *whip*) that is designed for use in embedded systems. *WHYP* stands for *Words to Help You Program*. It is a subroutine threaded language which means that *WHYP* words are just the names of 68HC12(11) subroutines. New *WHYP* words can be defined simply by stringing previously defined *WHYP* words together.

A unique feature of Forth—and WHYP—is its simplicity. It is a simple language to learn, to use, and to understand. In fact, in this book we will develop the entire WHYP language from scratch. We will see that WHYP consists of two parts—some 68HC12 subroutines that reside on the target system (typically an evaluation board) and a C++ program that runs on a PC and communicates with the 68HC12 target system through a serial line. In the process of developing the WHYP subroutines on the target system you will learn 68HC12 assembly language programming. When you finish the book you will also know Forth. Previous knowledge of C++ will be helpful in understanding the C++ portion of WHYP that resides on the PC. The complete C++ source code is included on the disk that accompanies this book and is discussed in Chapters 16 and 17. However, these chapters are optional and are not required in order to use WHYP to program the 68HC12.

You will discover that you can develop large software projects using WHYP in a much shorter time than you could develop the same program in either assembly language or C. You might be surprised at the number of industrial embedded systems projects that have been developed in Forth. Many small companies and consultants that use Forth don't talk much about it because many consider it a competitive advantage to be able to develop software in a shorter time than others who program in assembly language or C.

In Chapter 1 you will learn about the architecture of the 68HC12 and how to write a simple assembly language program, assemble it, download it to the target board, and execute it. You will see how to write 68HC12 subroutines in Chapter 2 where you will learn how the system stack works. We will then develop a separate data stack, using the 68HC12 index register, *X*, as a stack pointer. This data stack will be used throughout the book to pass parameters to and from our 68HC12 subroutines (WHYP words). We will see in Chapter 2 that this makes it possible to access our 68HC12 subroutines interactively, by simply typing the name of the subroutine on the PC keyboard.

In Chapter 3 we will study 68HC12 arithmetic with emphasis on the new 16-bit signed and unsigned multiplication and division instructions available on the 68HC12. We will use these instructions to create WHYP words for all of the arithmetic operations.

The power of WHYP comes from the fact that you can define new WHYP words in terms of previously defined words. This makes WHYP an extensible language in which every time you write a WHYP program you are really extending the language by adding new words to its dictionary. You will learn how to do this in Chapter 4.

In Chapter 5 we will look at the 68HC12 branching and looping instructions and see how we can use them to build some high-level WHYP branching and looping words such as an *IF . . . ELSE . . . THEN* construct and a *FOR . . . NEXT* loop. We will also see in this chapter how we can do recursion in WHYP, that is, how we can have a WHYP word call itself.

After the first five chapters you should have a good understanding of the 68HC12 instructions and how they are used to create the WHYP language. The next six chapters will use WHYP as a tool to explore and understand the I/O capabilities of the 68HC12 (and 68HC11). The important topic of interrupts is introduced in



Chapter 6 and specific examples of using interrupts in conjunction with various I/O functions are given in Chapters 7–11.

Parallel interfacing will be discussed in Chapter 7 where examples will be given of interfacing a 68HC12 to seven-segment displays, hex keypads, and liquid crystal displays. Real-time interrupts are used to program interrupt-driven traffic lights.

Chapter 8 will cover the 68HC12 Serial Peripheral Interface (SPI) where it will be shown how to interface keypads and seven-segment displays using the SPI. The 68HC11 and 68HC12 Analog-to-Digital (A/D) converter is described in Chapter 9 where an example is given of the design of a digital compass.

The 68HC12 programmable timer is discussed in Chapter 10 where examples are given of using output compares, input captures, and the pulse accumulator. Examples of using interrupts include the generation of a pulse train and the measurement of the period of a pulse train. An example of storing hex keypad pressings in a circular queue using interrupts is also included in Chapter 10. As a final example of using interrupts a design is given of a sonar tape measure using the Polaroid ultrasonic transducer.

Chapter 11 deals with the Serial Communication Interface (SCI) which is the module used by the 68HC12 to communicate with the PC.

Chapters 1–11 provide all the basic material needed to program a 68HC12 microcontroller for most applications. These chapters can form the basis of a one-term projects-oriented capstone design course at the senior/graduate level.

The material in Chapters 12 and 13 will be of interest to those who want access to more advanced topics related to programming in WHYP. Chapter 12 describes how to convert ASCII number strings to binary numbers and vice versa. Chapter 13 shows how you can create defining words using the *CREATE . . . DOES*> construct. These defining words are used to create jump tables and various data structures in WHYP.

The 68HC12 has special instructions that facilitate the implementation of fuzzy control. Chapter 14 discusses fuzzy control and shows how to design a fuzzy controller using WHYP on a 68HC12.

A number of special topics related to the 68HC12 are covered in Chapter 15 and as mentioned above Chapters 16 and 17 describe the C++ program for that part of WHYP that runs on the PC. The appendices contain the 68HC12 and 68HC11 instruction sets, plus useful information about WHYP, including procedures for installing WHYP on various evaluation boards.

Chuck Moore invented Forth in the late 1960s while programming minicomputers in assembly language. His idea was to create a simple system that would allow him to write many more useful programs than he could using assembly language. The essence of Forth is simplicity—always try to do things in the simplest possible way. Forth is a way of thinking about problems in a modular way. It is modular in the extreme. Everything in Forth is a word and every word is a module that does something useful. There is an action associated with Forth words. The words execute themselves. In this sense they are object oriented. We send words parameters on the data stack and ask the words to execute themselves and send us the answers back on

the data stack. We really don't care how the word does it—once we have written it and tested it so we know that it works.

Forth has been implemented in a number of different ways. Chuck Moore's original Forth had what is called an *indirect-threaded* inner interpreter. Other Forths have used what is called a *direct-threaded* inner interpreter. These inner interpreters get executed every time you go from one Forth word to the next, that is, all the time. WHYP is what is called a *subroutine-threaded* Forth. This means that the subroutine calling mechanism that is built into the 68HC12 is what is used to go from one WHYP word to the next. In other words, WHYP words are just regular 68HC12 subroutines. This both simplifies the implementation and speeds up the execution, at the expense of using somewhat more memory. In WHYP a word is compiled as a 3-byte jump-to-subroutine instruction while direct-threaded Forths need to store only the 2-byte address in memory. The inner interpreter takes care of reading the next address and executing the code at that address. Indirect-threaded Forths have an additional level of indirection. The 2-byte address in memory points not to the code to be executed, but to a location containing the address of the code to be executed. WHYP avoids these complications by being subroutine threaded and using the subroutine structure built into the 68HC12.

The way you program in Forth is bottom up—even though you may design the overall solution top down. You define a simple little word (subroutine) and test it out interactively at the keyboard. You put values on the data stack by simply typing them on the screen, separated by spaces, followed by the name of the word. When you press <enter>, the word (subroutine) is executed immediately and it leaves the answer(s) on the data stack which you can then display. This will all be explained in detail in the first five chapters of this book.

You should think of WHYP as your personal language that will allow you to write programs for the 68HC12 incrementally and interactively. Because we develop WHYP from scratch in this book there will be no mystery as to how it works. The entire source code—both the assembly language and the C++ parts—are included on the disk that comes with this book. In the true spirit of Forth this will give you complete control over your programming environment. Remember, Forth is an extensible language—and WHYP is your personal language that you will be able to extend and modify to suit your needs.

## Acknowledgment

The material in this book is based on many years of teaching Forth in a senior graduate course on embedded systems. My interest in and knowledge of Forth has benefited greatly from the Forth Interest Group (<http://www.forth.org/fig.html>) and many enjoyable years attending the annual FORML Conference in Pacific Grove, CA, and the annual Rochester Forth Conference in Rochester, NY. Many colleagues and students have influenced the development of this book. Their stimulating discussions, probing questions, and critical comments are greatly appreciated. I wish to thank Darrow F. Dawson of the University of Missouri-Rolla who reviewed the manuscript and made important suggestions that improved the book.

# *Contents*

Preface	xv
<b>Chapter 1 Introducing the 68HC12</b>	<b>1</b>
1.1 From Microprocessors to Microcontrollers	1
1.2 The 68HC12 Registers	8
1.2.1 The 68HC12 Accumulators	8
1.2.2 Index Registers, <i>X</i> and <i>Y</i>	11
1.2.3 Stack Pointer, <i>SP</i>	12
1.2.4 Program Counter, <i>PC</i>	13
1.2.5 The Condition Code Register	13
Carry ( <i>C</i> )	14
Zero Flag ( <i>Z</i> )	14
Negative Flag ( <i>N</i> )	15
Overflow Flag ( <i>V</i> )	15
Half-Carry ( <i>H</i> )	15
Interrupt Mask Flag ( <i>I</i> )	15
X-Interrupt Mask Flag ( <i>X</i> )	15
Stop Disable Flag ( <i>S</i> )	15

1.3	Writing Programs for the 68HC12	16
1.3.1	Editing and Assembling an .ASM File	16
1.3.2	Downloading and Executing the Program	18
1.4	Addressing Modes	22
1.5	Summary	24
	Exercises	24
<b>Chapter 2</b>	<b>Subroutines and Stacks</b>	<b>29</b>
2.1	The System Stack	29
2.2	Subroutines	31
2.3	A Data Stack	35
2.4	Making Subroutines Interactive	39
2.5	Stack Manipulation Words	40
2.6	The Return Stack	46
2.7	Summary	50
	Exercises	51
<b>Chapter 3</b>	<b>68HC12 Arithmetic</b>	<b>53</b>
3.1	Addition and Subtraction	53
3.1.1	Increment and Decrement Instructions	54
3.1.2	Double Numbers	55
3.1.3	Displaying Single and Double Numbers on the Screen	57
3.2	Multiplication	60
3.3	Division	63
3.3.1	16-Bit Signed Division: <i>IDIVS</i>	63
3.3.2	Extended Unsigned Division: <i>EDIV</i>	66
3.3.3	Extended Signed Division: <i>EDIVS</i>	69
3.3.4	Integer and Fractional Divide: <i>IDIV</i> and <i>FDIV</i>	71
	The Instruction <i>IDIV</i>	71
	The Instruction <i>FDIV</i> , Fractional Divide	73
	A 68HC11 Version of <i>UM/MOD</i>	74
3.4	Shift and Rotate Instructions	75
3.4.1	Logical Shift Instructions	76
	Logic Shift Left	76
	Logic Shift Right	76
3.4.2	Arithmetic Shift Instructions	77
3.4.3	Rotate Instructions	77
	Rotate Left	78
	Rotate Right	78
3.4.4	Shifting 16-Bit Words	78
3.5	Summary	79
	Exercises	82

<b>Chapter 4</b>	<b>WHYP—An Extensible Language</b>	<b>84</b>
4.1	A Closer Look at WHYP	84
4.2	Defining New WHYP Words	86
4.2.1	Defining New Words Interactively	86
4.2.2	<i>SEE</i> and <i>SHOW</i>	89
4.2.3	Single-Stepping through Colon Definitions	90
4.2.4	Loading WHYP Words from a File	91
4.2.5	<i>CR</i> and <i>.</i>	92
4.3	Variables	94
4.3.1	Fetch and Store	95
4.3.2	System Variables	99
4.3.3	Arrays	100
4.4	Constants	101
4.4.1	Tables	102
4.5	EEPROM	105
4.5.1	Erasing the EEPROM	106
4.5.2	Programming the EEPROM	109
4.5.3	Storing WHYP Programs in the EEPROM	110
4.6	Summary	114
	Exercises	116
<b>Chapter 5</b>	<b>Branching and Looping</b>	<b>120</b>
5.1	68HC12 Branch Instructions	120
5.1.1	Short Conditional Branch Instructions	121
5.1.2	Unconditional Branch and Jump Instructions	123
5.1.3	Branching Examples	123
	Branching on the Zero Flag <i>Z</i>	123
	Branching on the Negative Flag <i>N</i>	126
	Branching on the Carry and Overflow Flags, <i>C</i> and <i>V</i>	127
5.1.4	Bit-Condition Branch Instructions	127
5.1.5	Unsigned and Signed Branch Instructions	128
5.2	WHYP Branching and Looping Words	130
5.2.1	WHYP Conditional Words	131
5.2.2	WHYP Logical Words	133
5.2.3	<i>IF ... ELSE ... THEN</i>	134
5.2.4	<i>FOR ... NEXT</i> Loop	137
	A <i>FOR ... NEXT</i> Delay Loop	139
5.2.5	<i>BEGIN ... AGAIN</i>	141
5.2.6	<i>BEGIN ... UNTIL</i>	142
5.2.7	<i>BEGIN ... WHILE ... REPEAT</i>	145
	Sine and Arcsine	147
5.2.8	<i>DO ... LOOP</i>	150
	The Word <i>LEAVE</i>	151

5.3	Recursion in WHYP	153
5.4	Summary	155
	Exercises	157
<b>Chapter 6</b>	<b>Interrupts</b>	<b>162</b>
6.1	68HC12 Interrupts	163
6.1.1	68HC12 Nonmaskable Interrupts	163
	Reset	163
	COP (Computer Operating Properly)	163
	Unimplemented Instruction Trap	164
	Software Interrupts ( <i>SWI</i> )	164
	Nonmaskable Interrupt Request ( <i>XIRQ</i> )	165
6.1.2	68HC12 Maskable Interrupts	165
6.2	68HC11 Interrupts	167
6.3	Interrupt Vector Jump Tables	168
6.3.1	68HC711E9	168
6.3.2	D-Bug12	170
6.4	Writing WHYP Interrupt Service Routines	172
6.5	Real-Time Interrupts	173
6.5.1	Real-Time Interrupt on a 68HC11	176
6.6	Writing Assembly Language Interrupt Service Routines	178
6.7	Summary	181
	Exercises	181
<b>Chapter 7</b>	<b>Parallel Interfacing</b>	<b>183</b>
7.1	Parallel I/O Ports	183
7.1.1	The MC68HC812A4 Parallel Ports	184
7.1.2	The MC78HC912B32 Parallel Ports	186
7.1.3	The MC68HC711E9 Parallel Ports	188
	Port A	188
	Port B	188
	Port C	189
	Port D	191
	Port E	191
7.2	Using Parallel Ports	191
7.2.1	Using Parallel Port Outputs	191
7.2.2	Using Parallel Port Inputs	194
7.3	Seven-Segment Displays	195
7.3.1	Common-Anode Displays	196
7.3.2	Common-Cathode Displays—The MC14495-1	198
7.4	Keypad Interfacing	199
7.4.1	4 × 4 Hex Keypad	199
7.4.2	The 74C922 16-Key Encoder	204
7.4.3	Interfacing a 16 × 1 Hex Keypad Using a 74154 Decoder	205

7.5	Liquid Crystal Displays	207
7.6	Interrupt-Driven Traffic Lights	212
7.7	Summary	215
	Exercises	216
<b>Chapter 8 The Serial Peripheral Interface (SPI)</b>		<b>220</b>
8.1	Operation of the SPI	221
8.1.1	The SPI Registers	222
8.1.2	Programming the SPI in WHYP	225
8.2	Keypad Interfacing with 74165 Shift Registers	226
8.3	Four-Digit Seven-Segment Display Using a MC14499	228
8.4	The 68HC68T1 Real-Time Clock	232
8.5	Summary	236
	Exercises	237
<b>Chapter 9 Analog-to-Digital Converter</b>		<b>243</b>
9.1	Analog-to-Digital Conversion	243
9.2	The 68HC11 A/D Converter	245
9.2.1	WHYP Words for the 68HC11 A/D Converter	248
9.3	The 68HC12 A/D Converter	250
9.3.1	WHYP Words for the 68HC12 A/D Converter	254
9.4	Design of a Digital Compass	256
9.5	Summary	260
	Exercises	260
<b>Chapter 10 Timers</b>		<b>264</b>
10.1	The 68HC12 Programmable Timer	265
10.1.1	The 68HC11 Timer Registers	268
10.2	Output Compares	269
10.2.1	Pulse Train Example	271
10.2.2	Output Compares on a 68HC11	274
10.3	Input Capture	275
10.3.1	Input Captures on a 68HC11	277
10.4	Pulse Accumulator	277
10.4.1	The Pulse Accumulator on a 68HC11	280
10.5	Timing Interrupt Service Routines	280
10.6	A Circular Queue Data Structure	282
10.7	Keypad Interfacing Using Interrupts	284
10.8	Pulse Train Using Interrupts	286
10.9	Measuring the Period of a Pulse Train Using Interrupts	289
10.10	The Polaroid Ultrasonic Transducer	292
10.11	Summary	295
	Exercises	296

<b>Chapter 11 The Serial Communication Interface (SCI)</b>	<b>302</b>
11.1 Asynchronous Serial I/O	302
11.2 The 68HC12 SCI Interface	304
The Data Register	305
The Status Register	306
The Control Registers	307
The Baud Rate Control Register	309
11.2.1 Programming the SCI Port	310
11.2.2 The 68HC11 SCI Registers	312
11.3 Programming the SCI in WHYP	314
11.3.1 Communicating with a PC	316
11.3.2 Testing SCI1 with the LOOP Function	317
11.3.3 Sending Register Values to the PC	319
11.4 SCI Interface Using Interrupts	322
11.4.1 Master-Slave SCI Communications	324
11.5 Summary	326
Exercises	327
 <b>Chapter 12 Strings and Number Conversions</b>	 <b>331</b>
12.1 WHYP Strings	332
12.2 ASCII Number String to Binary Conversion	333
12.3 Binary Number to ASCII String Conversion	335
12.3.1 Examples of Converting Numbers to ASCII Strings	336
12.4 The WHYP Words <i>CMOVE</i> and <i>CMOVE&gt;</i>	339
12.5 Summary	340
Exercises	342
 <b>Chapter 13 Program Control and Data Structures</b>	 <b>344</b>
13.1 <i>CREATE . . . DOES&gt;</i>	344
13.2 Program Control	347
13.2.1 A Simple Jump Table	347
13.2.2 A Jump Table with WHYP Words	348
13.3 Data Structures	350
13.3.1 Arrays	351
13.3.2 Linked Lists	352
13.4 Summary	357
Exercises	357
 <b>Chapter 14 Fuzzy Control</b>	 <b>361</b>
14.1 Fuzzy Sets	361
14.2 Design of a Fuzzy Controller	364
14.2.1 Fuzzification of Inputs: <i>get_inputs()</i>	365
The 68HC12 <i>MEM</i> Instruction	366
WHYP Words for Defining Membership Functions	368



14.2.2	Fuzzy Inference	371
14.2.3	Processing the Rules: <code>fire_rules()</code>	373
	The 68HC12 <i>REV</i> Instruction	373
	WHYP Words for Defining Fuzzy Rules	375
14.2.4	Centroid Defuzzification	376
14.2.5	Output Defuzzification: <code>find_output()</code>	379
	The 68HC12 <i>WAV</i> Instruction	379
14.2.6	A Fuzzy Control Example—Floating Ping-Pong Ball	381
14.3	Summary	385
	Exercises	386
<b>Chapter 15</b>	<b>Special Topics</b>	<b>389</b>
15.1	Computer Operating Properly ( <i>COP</i> )	389
15.2	Key Wakeup (68HC812A4 only)	391
15.3	Flash EEPROM (68HC912B32 only)	394
	15.3.1 Erasing and Programming the Flash EEPROM	394
15.4	Pulse-Width Modulator (68HC912B32 only)	396
15.5	Summary	400
	Exercises	401
<b>Chapter 16</b>	<b>WHYP12 C++ Classes</b>	<b>402</b>
16.1	A Character Queue Class	402
16.2	A UART Class	405
	16.2.1 The 8250 UART	405
16.3	An S-Record Class	413
16.4	A Link List Class	416
16.5	A Dictionary Class	420
16.6	Summary	430
<b>Chapter 17</b>	<b>WHYP12 C++ Main Program</b>	<b>430</b>
17.1	Compiling and Running the WHYP Host Program	430
	17.1.1 The WHYP Configuration File	431
17.2	The Immediate Dictionary	434
17.3	The WHYP Main Program	439
	17.3.1 Checking the COM Port	441
	17.3.2 Getting Input from the Keyboard	442
	17.3.3 Processing an Input Word	445
17.4	Communicating with the Target Board	448
	17.4.1 The Terminal Host Function	450
17.5	Compiler Words	451
	17.5.1 Branching Words	451
	17.5.2 Compiling Colon Definitions	453
17.6	Processing Characters Received from the Target	454
17.7	Summary	458