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*

Prentice-Hall, Inc.
Upper Saddle River, New Jersey 07458

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 *W*ords to *H*elp *Y*ou *P*rogram. 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 in 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