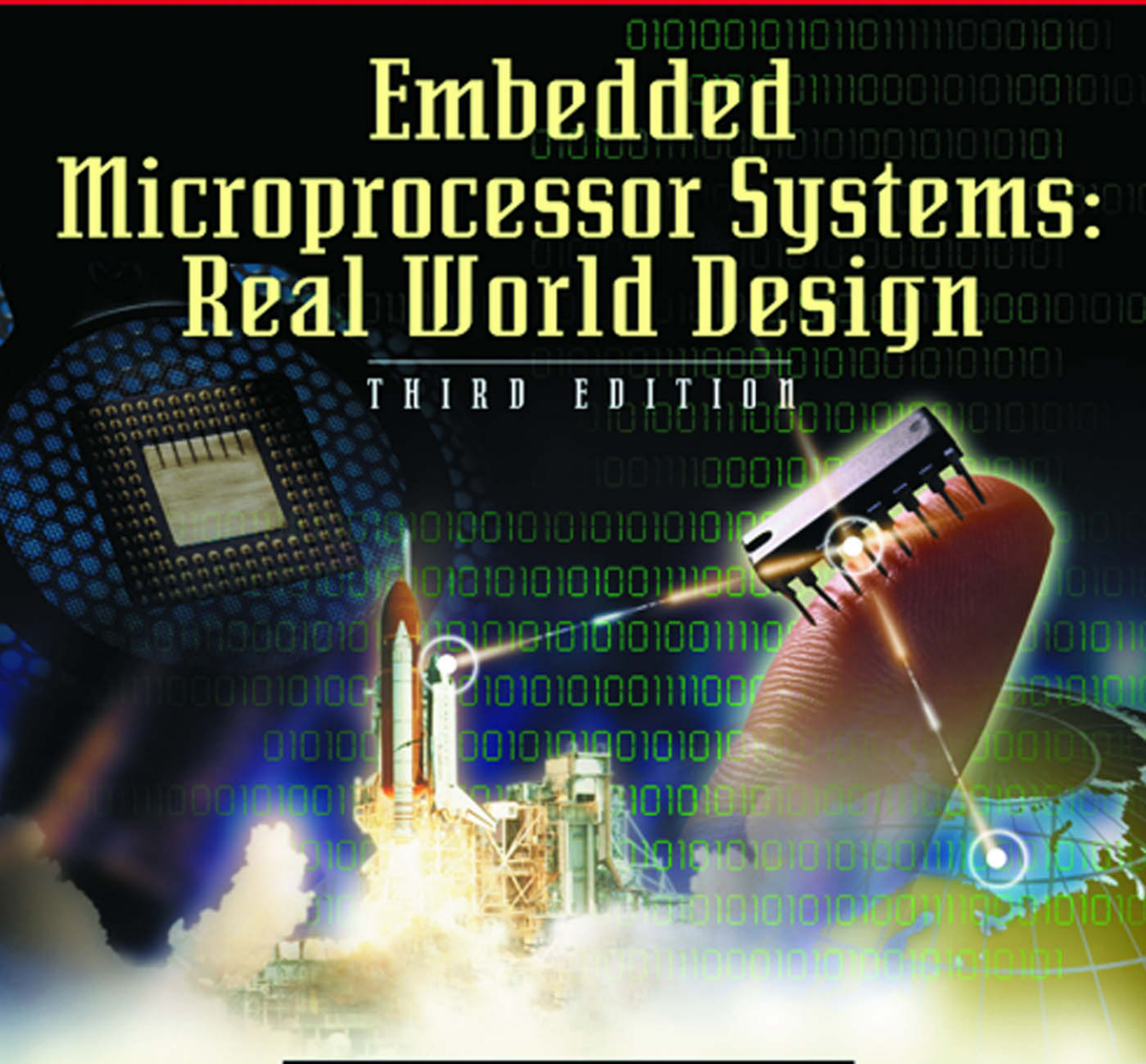
EMBEDDED TECHNOLOGY™

SERIES

Embedded Microprocessor Systems: Real World Design

THIRD EDITION



Stuart R. Ball, P.E.

Newnes

Embedded Microprocessor Systems
Real World Design

This Page Intentionally Left Blank

Embedded Microprocessor Systems
Real World Design

Third Edition

Stuart R. Ball

An imprint of Elsevier Science
Amsterdam Boston London New York Oxford Paris San Diego
San Francisco Singapore Sydney Tokyo

Newnes is an imprint of Elsevier Science.

Library of Congress Cataloging-in-Publication Data
Ball, Stuart R., 1956–
Embedded microprocessor systems : real world design / Stuart R. Ball.—3rd ed.
p. cm.
ISBN 0-7506-7534-9 (pbk. : alk. paper)
1. Embedded computer systems—Design and construction. 2. Microprocessors.
I. Title.
TK7895.E42 B35 2002
621.39′16—dc21 2002071917

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

The publisher offers special discounts on bulk orders of this book.
For information, please contact:

Manager of Special Sales
Elsevier Science
200 Wheeler Road
Burlington, MA 01803
Tel: 781-313-4700
Fax: 781-313-4880

For information on all Newnes publications available, contact our World Wide Web home page at: http://www.newnespress.com

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Contents

This Page Intentionally Left Blank

Introduction

Imagine this scene: You get into your car and turn the key on. You take a 3.5″ floppy disk from the glove compartment, insert it into a slot in the dashboard, and drum your fingers on the steering wheel until the operating system prompt appears on the dashboard liquid crystal display (LCD). Using the cursor keys on the center console, you select the program for the electronic ignition, then turn the key and start the engine. On the way to work you want to listen to some music, so you insert the program compact disc (CD) into the player, wait for the green light to flash indicating that the digital signal processor (DSP) in the player is ready, then put in your music CD.

You get to work and go to the cafeteria for a pastry. Someone has borrowed the mouse from the microwave but has not unplugged the microwave itself, so the operating system is still up. You can heat your breakfast before starting work.

What is the point of this inconvenient scenario? This is how the world would work if we used microprocessor technology without having *embedded microprocessors*. Every microprocessor-based appliance would need a disk drive, some kind of input device, and some kind of display.

Embedded microprocessors are all around us. Since the original Intel 8080 was pioneered in the 1970s, engineers have been embedding microprocessors in their designs. They even are embedded in general-purpose computers; if you own a variation of the IBM PC/AT, there is an embedded microprocessor in the keyboard. Virtually all printers have at least one microprocessor in them, and no car on the market is without at least one under the hood. Embedded microprocessors may control the automatic processing equipment that cans your soup or the controls of your microwave oven. Basically, we can define an embedded microprocessor as having the following characteristics:

- Dedicated to controlling a specific real-time device or function.
- Self-starting, not requiring human intervention to begin. The user cannot tell if the system is controlled by a microprocessor or by dedicated hardware.
- Self-contained, with the operating program in some kind of nonvolatile memory.

Of course, there are exceptions to this general description, which we will get to eventually, but this definition will serve us for now.

An embedded microprocessor system usually contains the following components:

- A microprocessor
- RAM (random access memory)
- Nonvolatile storage: erasable programmable read-only memory (EPROM), read-only memory (ROM), flash memory, battery-backed RAM, and so on
- I/O (some means to monitor or control the real world)

If you have seen textbooks describing general computer systems, this description fits those as well. The difference is in the details. A general-purpose computer, such as the one this book was written on, may have many megabytes of RAM, whereas an embedded system may have less than 256 bytes (that is bytes, not megabytes) of RAM. Your PC at home or at the office may have a 10GB IDE hard drive with DOS, Windows, and several other applications.

An embedded system usually contains its entire program in a few thousand bytes of EPROM. The most important difference between the two is the application. Your home personal computer (PC) runs a word processor, then you switch over to the money management program to balance your checkbook, then to the spreadsheet to work on the family budget, then back to the word processor. The embedded system does just a limited number of tasks, such as making sure your toast does not burn or timing the cook cycle in your microwave.

Why would anyone want to use a microprocessor? The main reasons are:

- **Cost.** The cost of developing firmware for an embedded system can be very high, but it is a *nonrecurring expense*, only spent once to develop the product. The actual cost of the finished product can be very low. On the other hand, the product cost of a system such as a microwave oven controller, if implemented in discrete hardware, can be very high by comparison.
- **Flexibility.** Say a typical microwave oven manufacturer gets a contract from a very large discount store for microwave ovens, but the contract specifies certain changes in the way the user controls the device. In a hardware-based system, the control electronics would need to be redesigned. In a microprocessor-based system, the only change may be a few lines of code.
- **Programmability.** You may want to program a robotic arm to paint car doors one day and trunk lids the next. An embedded microcontroller permits you to have the same hardware perform different tasks. Of course, this also could be implemented in discrete hardware but at much higher cost.
- **Adaptability.** A system may need to adapt to its environment or to a user's needs. A typical example of this is an automobile's "smart" automatic transmission, which remembers your driving patterns and adjusts its shift points for optimum

comfort, economy, or even reliability. You *could* implement this sort of feature with dedicated hardware, but a microprocessor makes the job much easier.

This book will take you step by step through the procedures involved in designing an embedded control system. Many of the tricks I have learned in my 20 years in the field will be passed on, as well as some pitfalls to avoid. Along the way, we will use as an example of a simple embedded control system, a swimming pool pump timer, to illustrate these concepts.

The book is aimed primarily at students, new graduates who will be moving into the embedded processor field, and engineers working in another field who want to switch to embedded microprocessors. It assumes that the reader has a basic knowledge of software concepts, binary and hexadecimal number systems, and a basic understanding of digital logic. A review of this material is included in the appendixes at the end.

Special Introduction to the Third Edition

Since the first edition of this book was published, the embedded microprocessor world has changed. Entire families of microprocessors have become obsolete, along with their associated peripheral devices. This march of technology has the disadvantage of making examples using those devices obsolete as well. In some cases, I have kept examples that used some of these older parts because they provide a clearer means of communicating a concept than examples using newer, more complex devices. In general I have tried to use parts that are still in production for the examples, although some of these parts may be nearing their end of life and not as common as newer parts.

In addition to using some older devices in examples, the text still refers to older logic devices as well. These latches, gates, and registers provide a well-understood means of illustrating an interface mechanism that tends to become overly complex if all the component parts must be explained in detail before the desired concept can be covered. In most modern circuits, these functions have been taken over by programmable logic or custom ICs. The concepts, however, are still valid even if the implementation technology has changed.

Owing to these advances in technology, I have added some new examples, using updated parts, to the book. Readers of the first and second editions of the book will note that some original examples have been replaced with examples that use these newer parts. Of course, there is no guarantee that any current production part will still be in production by the time you read this, but that is the nature of the electronics industry!

System Design 1

It has been said that if you do not know where you are going, you will not know when you get there. Success experts tell us that the first step in achieving anything is to establish a goal—to be debt free in one year or to pay off the car in six months.

Like most things in life, the process of designing an embedded microprocessor system begins with a goal—the definition of the product. The product definition describes what the product is to be and do. The product definition is the first element in a process that is key to any successful electronics system design: documentation. The documentation describes what you are going to build and how you are going to build it. It tells marketing people what product they will have to sell, and it tells the engineering team how to implement that product. Since this book is about embedded systems, it will focus on documenting embedded systems. The development documents that I have found useful in designing embedded systems are as follows:

- **Product Requirements**: Describe what the product is.
- **Functional Requirements**: Describe what the product must do.
- **Engineering Specification**: Describes how the design will be implemented and how the requirements will be met.
- **Hardware Specifications**: Describe how specific hardware is designed.
- **Firmware Specifications**: Describe how the firmware for specific processors will be designed.
- **Test Specifications**: Describe what must be tested and how to verify that the system operates correctly.

Figure 1.1 shows how each of the documents relates to the overall design. The embedded design process generally follows these steps:

Product requirements definition

Functional requirements definition

Processor selection

Hardware/software specifications

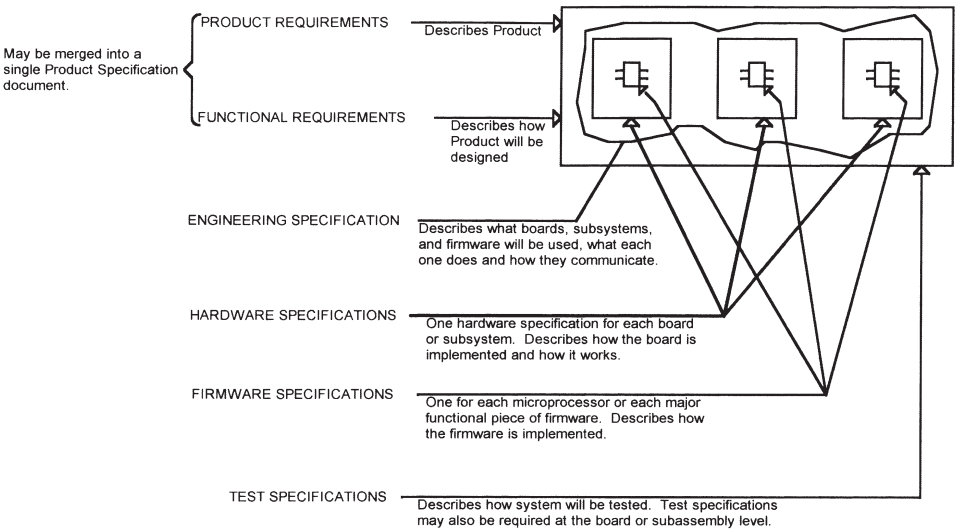


Figure 1.1
Design Documentation.

System evaluation

Hardware design

Firmware design

Integration

Verification (test)

These steps are not necessarily serial. For example, if there are separate hardware and software teams, the hardware and firmware design can proceed in parallel. The process is not always linear—system evaluation may reveal a problem with the selected processor, which means that step must be repeated. Last, the process is not always this well divided. The requirements definition and functionality description, for example, may be merged into a product specification or other customer-required documents.

Many companies require such product specifications early in the design process. I will not dwell on that here, as the requirements for this type of document are specific to the company or the customer for whom the product is intended. Commercial customers, to pick one example, have considerably different requirements than the Department of Defense. The design and documentation process begins with the next level of documentation below the product specification: the requirements definition.

Requirements Definition

The requirements definition (which, again, may actually be part of the product specifications), describes what the product is to do. In a very large company, the marketing department or a major customer may define the requirements. In a smaller company, the hardware and software engineers may sketch out the requirements definition. For a small, one-engineer project, the requirements may be the result of a momentary inspiration.

The requirements definition can take the form of a book—defining every interaction, interface, and error condition in the system—or a single-page list of what the finished product must do. In either case, the requirements definition must describe:

- What the system is to do
- What the real world I/O consists of
- What the operator interface is (if any)

In a small embedded control system, defining the requirements is crucial, as it prevents problems later when you find out that there is insufficient RAM or that the microprocessor you have chosen is too slow for the job. A simple example of this is the following system definition for a swimming pool pump timer. (Appendix A contains the complete requirements definition and specifications.)

> System description: A swimming pool timer that cycles the alternating current (AC) pump motor on a swimming pool.
>
> Power input: 9 to 12V DC from a wall-mount transformer.
>
> Pump is a 1/2-hp, single-phase, AC motor, controlled by mechanical relay.
>
> Provision is to be made for a switch closure input that inhibits pump operation if the water level is low.
>
> User can set the length of time the pump is on and off. An override is available to permit turning off the pump when it is on for maintenance and turning on the pump when it is off so that chemicals can be added.
>
> On/off/override time is to be adjustable in 30-minute increments from 1/2 hour to 23 hours.
>
> A display will indicate the on/off condition of the pump, the time remaining, and whether the pump is in override mode. The display also will indicate the condition of the water-low monitor.
>
> Minimum switches and knobs.

In addition to a list of requirements and functions like this, a system that is intended to be a commercial product might also include requirements for EMI/

EMC (electromagnetic interference/electromagnetic compatibility) certification, safety agency approval (UL/IEC), and environmental specifications (temperature, humidity, salt spray, and so on).

Although we'll discuss this further in Chapter 7, one problem with specifying requirements is verifying them. It is easy to determine whether the product meets the EMI/EMC requirements—you can run tests to prove it. But how do you prove you've met the requirement for "minimum switches and knobs"? Thus, keep in mind the problem of verification when specifying requirements.

A complex system may have another level of documentation, which I usually refer to as the *Engineering Specification*. This document describes the approach that will be used to implement the design, including which boards will be included and how the functions are partitioned onto those boards. I will return to this information later, in Chapter 8. For now, assume that we have a simple product, which makes this intermediate document unnecessary.

After the requirements are defined, the next step is to determine whether a microprocessor is the best choice. For the pool timer, it is fairly obvious that a microprocessor is the easiest way to do the job. Some other systems are not so obvious. The following questions can help determine whether a microprocessor is justified:

- At what speed must the inputs and outputs be processed or updated? Although the clock rates are ever increasing, there is a practical upper limit to the speed at which a microprocessor can read an input or update an output and still do any real work. At the time of this writing, an update rate of a few hundred kHz is a practical upper limit for a simple microprocessor system with few processing demands and running on a fast processor or digital signal processor (DSP). If the system must do significant processing, buffer manipulation, or other computing, the potential update rate will decrease.
- Is there a single integrated circuit (IC) or a programmable logic device (PLD) that will do the job? If so, a microprocessor is probably not justified.
- Does the system have a lot of user I/O, such as switches or displays? If so, a microprocessor usually makes the job much easier.
- What are the interfaces to other external systems? If your system must talk to something else using Synchronous Data Link Control (SDLC) or some other complex communication protocol, a microprocessor may be the only practical choice.
- How complex is the computational burden on the system? Modern electronic ignition systems, for example, have so many inputs (air sensors, engine rpm, and so on) with complex relationships that few choices other than a microprocessor are suitable.
- Will the design need to be changed once it is finished, or will the requirements be changing as the design progresses? Is there a need for customization of the