

LOW PRICE EDITION

PEARSON

Education

Documents
Document Sharing



An Embedded Software Primer

David E. Simon



CD contains the real-time
operating kernel,
µC/OS V1.11



This edition is manufactured in India and is authorized
for sale only in India, Bangladesh, Bhutan, Pakistan, Nepal,
Sri Lanka and the Maldives. Circulation of this edition
outside of these territories is UNAUTHORIZED.

Full document of An Embedded Software Primer

Other related Pearson Education titles available in LOW PRICE EDITION

Fundamentals of Embedded Software: Where C & Assembly Meet, 1/e (with CD)

Daniel W. Lewis

The 8051 Microcontrollers & Embedded Systems, 1/e (with CD)

Muhammad Ali Mazidi, Janice Gillispie Mazidi

Embedded Microcontrollers

Todd D. Morton

Design with PIC Microcontrollers

John B. Peatman

For more details log on to **www.pearsoned.co.in**

Leonardo
Steinfeld

An Embedded Software Primer



Documents
Document Sharing Hub

An Embedded Software Primer

David E. Simon



Documents
Document Sharing Hub

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Pearson Education was aware of a trademark claim, the designations have been printed in initial caps or in all caps.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Copyright © 1999 by Pearson Education, Inc.

This edition is published by arrangement with Pearson Education, Inc.

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of both the copyright owner and the above-mentioned publisher of this book.

ISBN 81-7808-045-1

First Indian Reprint, 2000

Second Indian Reprint, 2001

Third Indian Reprint, 2001

Fourth Indian Reprint, 2001

Fifth Indian Reprint, 2002

Sixth Indian Reprint, 2002

Seventh Indian Reprint, 2002

Eighth Indian Reprint, 2003

Ninth Indian Reprint, 2003

Tenth Indian Reprint, 2004

Eleventh Indian Reprint, 2004

Twelfth Indian Reprint, 2005

This edition is manufactured in India and is authorized for sale only in India, Bangladesh, Pakistan, Bhutan, Nepal, Sri Lanka and the Maldives.

Published by Pearson Education (Singapore) Pte. Ltd., Indian Branch, 482 F.I.E. Patparganj,
Delhi 110 092, India

Printed in India by Tan Prints (I) Pvt. Ltd.

To A. J. Nichols

Contents

Preface xi

Acknowledgments xiii

About This Book and the Accompanying CD-ROM xv

1 A First Look at Embedded Systems 1

1.1 Examples of Embedded Systems 1

1.2 Typical Hardware 8

Chapter Summary 10

2 Hardware Fundamentals for the Software Engineer 13

2.1 Terminology 13

2.2 Gates 16

2.3 A Few Other Basic Considerations 20

2.4 Timing Diagrams 28

2.5 Memory 33

Chapter Summary 40

Problems 41

3 Advanced Hardware Fundamentals 45

3.1 Microprocessors 45

3.2 Buses 47

3.3 Direct Memory Access 57

3.4 Interrupts 61

3.5 Other Common Parts 62

- 3.6 Built-Ins on the Microprocessor 72
- 3.7 Conventions Used on Schematics 75
- 3.8 A Sample Schematic 75
- 3.9 A Last Word about Hardware 77
- Chapter Summary 78
- Problems 79

4 Interrupts 81

- 4.1 Microprocessor Architecture 81
- 4.2 Interrupt Basics 85
- 4.3 The Shared-Data Problem 92
- 4.4 Interrupt Latency 103
- Chapter Summary 111
- Problems 112

5 Survey of Software Architectures 115

- 5.1 Round-Robin 115
- 5.2 Round-Robin with Interrupts 119
- 5.3 Function-Queue-Scheduling Architecture 127
- 5.4 Real-Time Operating System Architecture 129
- 5.5 Selecting an Architecture 132
- Chapter Summary 133
- Problems 134

6 Introduction to Real-Time Operating Systems 137

- 6.1 Tasks and Task States 139
- 6.2 Tasks and Data 144
- 6.3 Semaphores and Shared Data 153
- Chapter Summary 168
- Problems 169

7 More Operating System Services 173

- 7.1 Message Queues, Mailboxes, and Pipes 173
- 7.2 Timer Functions 184
- 7.3 Events 191
- 7.4 Memory Management 195
- 7.5 Interrupt Routines in an RTOS Environment 199
- Chapter Summary 206
- Problems 207

8 Basic Design Using a Real-Time Operating System 215

- 8.1 Overview 215
- 8.2 Principles 217
- 8.3 An Example 233
- 8.4 Encapsulating Semaphores and Queues 244
- 8.5 Hard Real-Time Scheduling Considerations 253
- 8.6 Saving Memory Space 254
- 8.7 Saving Power 257
- Chapter Summary 259
- Problems 260

9 Embedded Software Development Tools 261

- 9.1 Host and Target Machines 261
- 9.2 Linker/Locators for Embedded Software 263
- 9.3 Getting Embedded Software into the Target System 276
- Chapter Summary 280

10 Debugging Techniques 283

- 10.1 Testing on Your Host Machine 284
- 10.2 Instruction Set Simulators 302
- 10.3 The assert Macro 304

10.4 Using Laboratory Tools 307

Chapter Summary 326

Problems 327

11 An Example System 329

11.1 What the Program Does 330

11.2 Environment in Which the Program Operates 333

11.3 A Guide to the Source Code 336

11.4 Source Code 339

Summary 402

Problems 403

Afterword 405

Further Reading 407

Index 409

Preface

This book is to help you learn the basic principles of writing software for embedded systems. It surveys the issues and discusses the various techniques for dealing with them. In particular, it discusses approaches to the appropriate use of the real-time operating systems upon which much embedded software is based. In addition to explaining what these systems do, this book points out how you can use them most effectively.

You need know nothing about embedded-systems software and its problems to read this book; we'll discuss everything from the very beginning. You should be familiar with basic computer programming concepts: you might be a software engineer with a year or more of experience, or perhaps a student with a few programming courses under your belt. You should understand the problems involved in writing application programs. This book requires a reading knowledge of the C programming language; since C is the lingua franca of embedded systems, you will have to learn it sooner or later if you hope to get into the field. A little knowledge of assembly language will also be helpful.

You have no doubt seen many books about software that are 800 or 900 or even 1000 pages long. Presumably you have noticed by now that this book is much smaller than that. This is intentional—the idea is that you might actually want to read all the way through it. This book is not entitled *Everything There Is to Know about Embedded Systems Software*. Nobody could write that book, and if someone could and did, you wouldn't want to read it anyway. This book is more like *What You Need to Know to Get Started in Embedded Systems Software*, telling you enough that you'll understand the issues you will face and getting you started on finding the information about your particular system so that you can resolve those issues.

This book is not specific to any microprocessor or real-time operating system nor is it oriented towards any particular software design methodology. The principles are the same, regardless of which microprocessor and which real-time operating system and which software design methodology you use. We will concentrate on the principles—principles that you can apply to almost

any embedded system project. When you need to know the specifics of your microprocessor and your real-time operating system, look in the voluminous manuals that hardware and software vendors provide with their products. This book will help you know what information to look for.

This book is not academic or theoretical; it offers engineering information and engineering advice.

In short, this book is the cornerstone of the knowledge that you'll need for writing embedded-systems software.

David E. Simon

Acknowledgments

No one has enough good ideas for a book such as this or the perseverance to see it through without help from other people. So here—more or less in chronological order—is the story of this book and the people who helped me turn it into reality.

First, thanks are due to the people at Probitas Corporation: to A. J. Nichols, who has made the company the thoughtful, high-quality software environment that it is; to Michael Grischy for the ongoing debates on embedded-system design and coding style; to Richard Steinberg, who checked the code examples in this book; and to Ric Vilbig, who reviewed the two chapters on hardware and corrected a number of my misconceptions.

My wife, Lynn Gordón, encouraged me to write this book, predicting—correctly, as it turned out—that I would enjoy doing it. Thank you for getting me started, and thanks for the editing help . . . even if you are always right about the fine points of English usage.

Thank you to a half-dozen classes full of students: to those of you who asked the advanced questions and forced me to clarify my thinking, to those of you who asked the elementary questions and forced me to clarify my explanations, and to all of you who suffered in silence with early versions of the manuscript.

Thanks to the smart people at Dragon Systems, Inc., who wrote *NaturallySpeaking*, a voice recognition program good enough to allow me to prepare a manuscript while injuries prevented me from typing much.

A huge thanks to Jean Labrosse for giving permission to include his real-time operating system, $\mu C/OS$, as part of this book. You have done the world a real favor in writing this system and in allowing it to be used freely for educational purposes.

A thank you to John Keenan, who taught me a lot of what I know about hardware and who straightened out a few blunders that made it into the manuscript.

The following people reviewed the first version of the manuscript and provided reams of good suggestions, many of which I incorporated into the

book: Antonio Bigazzi, Fred Clegg, David Cuka, Michael Eager, John Kwan, Tom Lyons, and Steve Vinoski. Thank you all.

Thanks are due to Mike Hulme at Zilog, who gave permission to use the schematic example at the end of Chapter 3 and who ran down a legible copy of it.

Finally, thanks to Debbie Lafferty and Jacquelyn Doucette, who shepherded this book through its various stages; to Ann Knight and Regina Knox, who pointed out all of those things that had somehow made sense when I wrote them but didn't later when someone else tried to read them; and to Laurel Muller, who turned my scruffy sketches into legible figures.

About This Book and the Accompanying CD-ROM

The Perversities of Embedded Systems

One very unfortunate aspect of embedded systems is that the terminology surrounding them is not very consistent. For every concept, there are two or three different words. For every word, there are four or five subtly different meanings. You will just have to live with this problem. In this book we will point out the variations in meaning; then we will assign a specific meaning for each word so that we don't get confused. When you are reading other books or talking to people, however, you'll have to be aware that their words may mean something slightly different from what they mean in this book.

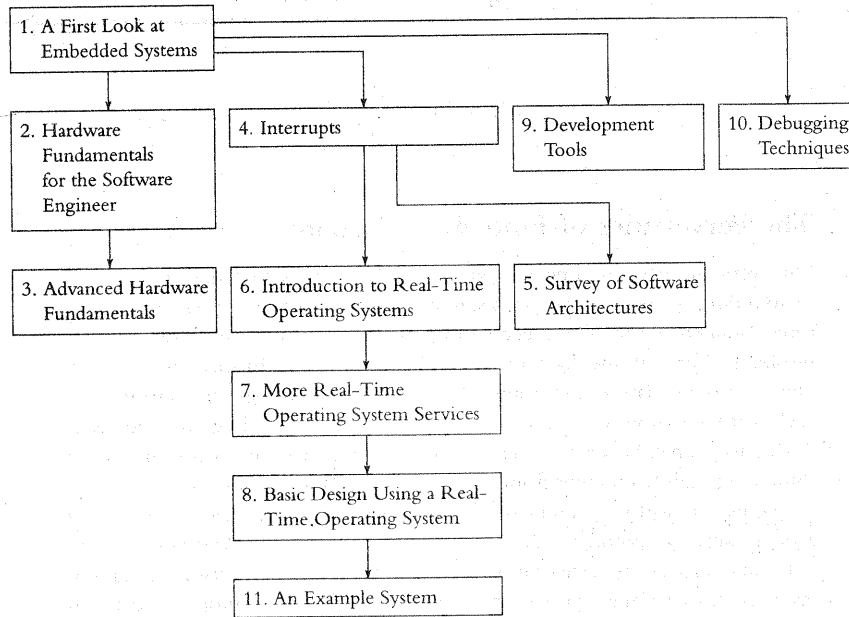
Another unfortunate problem is that the term *embedded systems* covers such a broad range of products that generalizations are difficult. Systems are built with microprocessors as diverse as the Z8, an 8-bit microprocessor that cannot even use external memory, to the PowerPC, a 32-bit microprocessor that can access gigabytes. The code size of the applications varies from under 500 bytes to millions of bytes.

Because of this and because of the wide variety of applications, embedded software is a field in which no wisdom is universal. Any rule followed by 85 percent of engineers as part of the accepted gospel of best practice has to be broken by the other 15 percent just to get their systems to work. This book will focus on the rules of the 85 percent, emphasizing the concepts and the reasoning behind the rules and helping you decide whether you should follow the common practice or if your project is part of the 15 percent.

Chapter Dependencies in This Book

Although this book is intended to be read straight through, and although every chapter depends at least a little upon the chapters that precede it, you can skip around if you like. Since this book starts every subject at the very beginning, you may be able to skip some sections if you already know some of the material.

The most important dependencies among the chapters are shown in the diagram here.



If you already know about hardware, for example, or if your work doesn't require that you know anything about it, you can skip Chapters 2 and 3. However, don't try to read Chapter 6 without reading Chapter 4 or knowing about the material in it.

C++

Although C++ is an increasingly popular language in the embedded-systems world, you will not see it in this book. This is not intended to discourage you from using C++, which is popular in the embedded-systems world for the same good reasons it is popular in the applications world. However, one of the acknowledged disadvantages of C++ is that it is a complicated language, in many ways much more difficult and subtle than C. The programming principles discussed in this book apply equally to C and to C++ (and to Ada, Java, BASIC,

and any other language in which you might choose to program your embedded system, for that matter). Therefore, for the purposes of illustration in this book, it makes sense to steer clear of the complications of C++.

C!!

One of the problems of providing understandable examples in a book such as this (and in fact one of the problems of software in general) is that important points tend to get lost in a morass of detail. To prevent that from happening, some of the examples in this book will not be written entirely in C. Instead, they will be written in C!!.

C!! is identical to C, except that wherever you put two exclamation points, the computer does whatever is described after those exclamation points. For example:

```

if (x != 0)
{
    !! Read timer value from the hardware
    !! Do all the necessary ugly arithmetic
    y = !! Result of the ugly arithmetic
}
if (y > 197)
{
    !! Turn on the warning light
}
:
:
:

```

If x is not zero, then this program does whatever is necessary to read the value from the hardware timer. Then it does various necessary calculations and stores the result in y. If y is greater than 197, this program turns on the warning light.

Rest assured that we only use the special feature of C!! for such things as hardware-dependent code or specific calculations needed for some arcane application. The parts of the examples that pertain to the point being made are written in plain old vanilla C.¹

1. It would be nice if we could all write all our problems in C!!, but unfortunately, the compiler for C!! is still under development.

Hungarian Variable Naming

Many of the code examples in this book use a convention for naming variables called **Hungarian**. Here is a brief introduction to that convention. In Hungarian, the name of a variable contains information about its type. For example, names for `int` variables begin with the letter “i,” names for `byte` (unsigned `char`) variables begin with “by,” and so on. In addition, certain prefixes on variable names indicate that a variable is a pointer (“p_”), an array (“a_”), and so on. Here are some typical variable names and what you can infer about the variables from their names:

`byError`—a `byte` variable (that contains an error code, probably).

`iTank`—an integer (that contains the number of a tank, perhaps).

`p iTank`—a pointer to an integer.

`a_chPrint`—an array of characters (to be printed, most likely).

`fDone`—a flag (indicating whether a process is done).

Hungarian is popular because, even though the variable names are at first somewhat cryptic, many believe that the little bit of information contained in the name makes coding a little easier.²

μ C/OS

When you get to the discussion of real-time operating systems in Chapter 6 and beyond, you might want to try your hand at using one of these systems. To help you do that, the CD that accompanies this book has one, named μ C/OS (pronounced “micro-see-oh-ess”). It is on the CD with the permission of Jean Labrosse, who wrote the system. You are free to use μ C/OS as a study tool, but you may not use it for a commercial product without permission from Mr. Labrosse. **μ C/OS IS NOT “SHAREWARE.”** The licensing provisions for μ C/OS are shown on page xix.

The information in Chapters 6 and 7 and in Table 11.2 should get you started using this system. If you want more information about the system, see Mr. Labrosse’s book, listed in the section on Further Reading, or go to the μ C/OS

2. The Hungarian used in this book is actually a dialect, not quite the original convention. The major differences between the dialect and the original are (1) the original does not use an underscore to separate a prefix from the rest of the variable name, and (2) the dialect uses the convention to name functions as well as variables.

μ C/OS Licensing Information

μ C/OS source and object code can be freely distributed (to students) by accredited colleges and universities without requiring a license, as long as there is no commercial application involved. In other words, no licensing is required if μ C/OS is used for educational use.

You must obtain an Object Code Distribution License to embed μ C/OS in a commercial product. This is a license to put μ C/OS in a product that is sold with the intent to make a profit. There will be a license fee for such situations, and you need to contact Mr. Labrosse for pricing.

You must obtain a Source Code Distribution License to distribute μ C/OS source code. Again, there is a fee for such a license, and you need to contact Mr. Labrosse for pricing.

You can contact Jean Labrosse at:

Jean.Labrosse@uCOS-II.com

or

Jean J. Labrosse
949 Crestview Circle
Weston, FL 33327
USA

1-954-217-2036 (phone)
1-954-217-2037 (fax)

Web site at www.uCOS-II.com. You can contact Mr. Labrosse for support of μ C/OS, but please do not do this until you have checked the μ C/OS Web site for the latest versions and fixes.

The example programs on the CD and μ C/OS are intended for use with the Borland C/C++ compiler for DOS. You can get the “scholar edition” of this compiler (again, for study use, not for use in developing a commercial product) for \$49.95 as of this writing. Various retailers carry it, or you can contact Borland at www.borland.com.

A First Look at Embedded Systems

As microprocessors have become smaller and cheaper, more and more products have microprocessors “embedded” in them to make them “smart.” Such products as VCRs, digital watches, elevators, automobile engines, thermostats, industrial control equipment, and scientific and medical instruments are driven by these microprocessors and their software. People use the term **embedded system** to mean any computer system hidden in any of these products.

Software for embedded systems must handle many problems beyond those found in application software for desktop or mainframe computers. Embedded systems often have several things to do at once. They must respond to external events (e.g., someone pushes an elevator button). They must cope with all unusual conditions without human intervention. Their work is subject to deadlines.

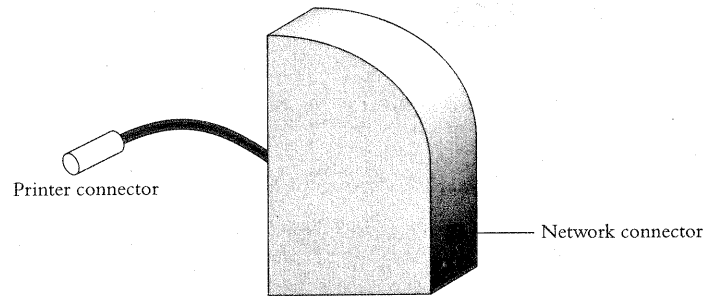
1.1 Examples of Embedded Systems

To understand the issues of embedded-systems software and to make the problems a little more concrete, let's start by examining a few sample systems. We'll look back at these examples from time to time as we discuss specific issues and specific solutions.

Telegraph

The first system that we will study is one that was code-named “Telegraph” during its development. Telegraph allows you to connect a printer that has only a high-speed serial port to a network. From the outside, Telegraph is a little

Figure 1.1 Telegraph



plastic box, 2 to 3 inches on a side and about half an inch thick. A pigtail cable on one side of the box plugs into the serial port on the printer. A connector on the other side of the box plugs into the network. A sketch of Telegraph is shown in Figure 1.1.¹

Obviously, Telegraph must receive data from the network and copy it onto the serial port. However, Telegraph is rather more complicated than that. Here are just a few things that Telegraph must do:

- On the network, data sometimes arrive out of order, data sometimes get lost along the way, and some of the data sometimes arrive twice. Telegraph must sort out the chaos on the network and provide a clean data stream to the printer.
- There might be lots of computers on the network, all of which might want to print at once. The printer expects to be plugged into a single computer. Telegraph must feed the printer one print job at a time and somehow hold off all the other computers.
- Network printers must provide status information to any computer on the network that requests it, even if they are busy printing a job for some other computer. The original, serial-port printer can't do that. Telegraph has to.
- Telegraph has to work with a number of different types of printers without customer configuration; Telegraph has to be able to figure out the kind of printer to which it is attached.

1. Telegraph was built to work with Apple inkjet printers, which typically had a serial port that you could connect directly to a Macintosh computer. Its shape allows it to snap directly onto the back of one of these printers. Various versions of it worked with different networks.

- Telegraph must respond quite rapidly to certain events. There are, for example, various kinds of network frames to which Telegraph must send a response within 200 microseconds.
- Telegraph must keep track of time. For example, if a computer that has been sending print data to Telegraph crashes, Telegraph must eventually give up on that print job—perhaps after 2 minutes—and print from another computer on the network. Otherwise, one computer crash would make the printer unavailable to everybody.

Telegraph Development Challenges

To satisfy the list of requirements given above, Telegraph has a microprocessor embedded in it. Its software is more extensive and sophisticated than its external appearance might lead you to believe. What problems arise in developing such software? Before reading on, you might consider writing down what you think these problems might be.

To begin with, of course, software for Telegraph must be logically correct. It can't lose track of which computer is printing or drop data or report incorrect status. This is the same requirement placed on every piece of software in both the embedded and the applications arenas.

However, writing the software for Telegraph—like writing software for many other embedded systems—offers up a few additional challenges, which we shall now discuss.

Throughput

The printer can print only as fast as Telegraph can provide data to it. Telegraph must not become a bottleneck between the computers on the network and the printer. For the most part, the problem of getting more data through an embedded system is quite similar to that of getting an application to run faster. You solve it by clever programming: better searching and sorting, better numerical algorithms, data structures that are faster to parse, and so on. Although these techniques are beyond the scope of this book, we will discuss mistakes possible with real-time operating systems that will spoil your throughput.

Response

When a critical network frame arrives, Telegraph must respond within 200 microseconds, even if it is doing something else when the frame arrives. The

software must be written to make this happen. We will discuss response extensively, because it is a common problem in embedded systems and because all of the solutions represent compromises of various kinds.

People often use the relatively fuzzy word “speed.” However, embedded-system designers must deal with two separate problems—throughput and response—and the techniques for dealing with the two are not at all the same. In fact, dealing with one of these problems often tends to make the other one worse. Therefore, in this book we will stick to the terms **throughput** and **response**, and we will avoid speed.

Testability

It is not at all easy to determine whether Telegraph really works. The problem is that a lot of the software deals with uncommon events. Telegraph is typical of embedded systems in this regard, because these systems must be able to deal with *anything* without human intervention. For example, lots of the Telegraph code is dedicated to the problem that data might get lost on the network. However, data doesn’t get lost very often, especially in a testing laboratory, where the network is probably set up perfectly, is made entirely of brand-new parts, and is all of 15 feet long. This makes it hard to test all those lines of code.

Similarly, Telegraph must deal with events that are almost simultaneous. If two computers request to start their print jobs at exactly the same time, for example, does the software cope properly? Telegraph contains code to handle this situation, but how do you make it happen in order to test that code?

We will discuss testing problems.

Debugability

What do you think typically happens when testing uncovers a bug in the Telegraph software? Telegraph has no screen; no keyboard; no speaker; not even any little lights. When a bug crops up, you don’t get any cute icons or message boxes anywhere. Instead, Telegraph typically just stops working. A bug in the network software? A bug in the software that keeps track of which computer is printing? A bug in the software that reports printer status? Telegraph just stops working.

Unfortunately, having Telegraph stop working doesn’t give you much information about a bug. Further, with no keyboard and screen you can’t run a debugger on Telegraph. You must find other ways to figure out what has happened. We will discuss techniques for debugging embedded-systems software,

and we’ll discuss a few techniques for keeping some of the more difficult bugs from creeping into your software in the first place.

Reliability

Like most other embedded systems, Telegraph is not allowed to crash. Although customers seem to have some tolerance for desktop systems that must be rebooted once in a while, nobody has any patience with little plastic boxes that crash. In particularly awkward situations, application software can put a message on the screen and ask the user what to do. Embedded systems do not have that option; whatever happens, the software must function without human intervention.

Memory Space

Telegraph has only a very finite amount of memory—specifically, 32 KB of memory for its program and 32 KB of memory for its data. This was as much memory as Telegraph could have if its price were to be reasonable. Memory gets nothing but cheaper, but it still isn’t free. Making software fit into the available space is a necessary skill for many embedded-system software engineers, and we’ll discuss it.

Program Installation

The software in Telegraph didn’t get there because somebody clicked a mouse on an icon. We will discuss the special tools that are needed to install the software into embedded systems.

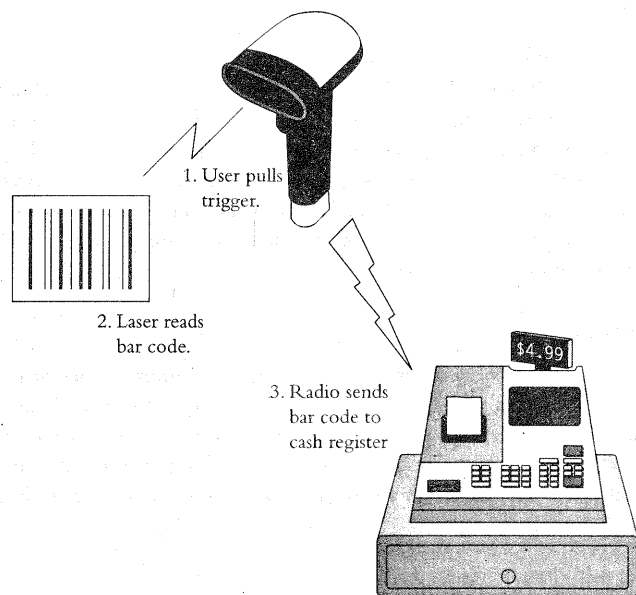
Cordless Bar-Code Scanner

Let’s turn to another embedded-systems example, a cordless bar-code scanner. Whenever its user pulls the trigger, the cordless bar-code scanner activates its laser to read the bar code and then sends the bar code across a radio link to the cash register. (See Figure 1.2.)

How do the problems of developing the software for the cordless bar-code scanner compare to those of developing the software in Telegraph?

Well, they’re mostly the same. One problem that the cordless bar-code scanner does *not* have is the problem of throughput. There just isn’t very much data in a bar code, and the user can’t pull the trigger that fast. On the other hand, the cordless bar-code scanner has one problem that Telegraph does not.

Figure 1.2 Cordless Bar-Code Scanner



Power Consumption

Since the scanner is cordless, its battery is its only source of power, and since the scanner is intended to be handheld, the weight of the battery is limited by what an average user can comfortably hold up. How long does the customer want the battery to last? The obvious answer—forever—isn't feasible. What is the next best answer?

The next best answer is that the battery should last for an 8-hour shift. After that, the scanner can go back into a holster on the side of the cash register for the night and recharge its battery. It turns out, however, that it also isn't feasible to run a laser, a microprocessor, a memory, and a radio for 8 hours on battery power. Therefore, one of the major headaches of this software is to figure out what parts of the hardware are not needed at any given time and turn those parts off. That includes the processor. We'll discuss this, too.

Laser Printer

Another embedded system is the laser printer. Most laser printers have fairly substantial microprocessors embedded in them to control all aspects of the printing. In particular, that microprocessor is responsible for getting data from the various communication ports on the printer, for sensing when the user presses a button on the control panel, for presenting messages to the user on the control panel display, for sensing paper jams and recovering appropriately, for noticing when the printer has run out of paper, and so on.

But the largest responsibility of the microprocessor is to deal with the **laser engine**, which is that part of the printer responsible for putting black marks on the paper. The only thing that a laser engine knows how to do without microprocessor assistance is to put a black dot or not to put a black dot at each location on a piece of paper. It knows nothing about the shapes of letters, fonts, font sizes, italic, underlining, bold, or any of those other things that printer users take for granted. The microprocessor must read the input data and figure out where each black dot should go. This brings us to another problem found in some embedded systems.

Processor Hogs

Figuring out where the black dots go when a printer has been asked to print some text on a slanted line with an unusual font in a screwball size takes a lot of time, even for powerful microprocessors. Users expect a quick response when they push buttons, however; it is no concern of theirs that the microprocessor is busy figuring out values for trigonometric functions to discover where on the page the serifs of a rotated letter should go. Work that ties up the processor for long periods of time makes the response problem that much harder.

Underground Tank Monitor

The underground tank monitoring system watches the levels of gasoline in the underground tanks at a gas station. Its principal purpose is to detect leaks before the gas station turns into a toxic waste dump by mistake and to set off a loud alarm if it discovers one. The system also has a panel of 16 buttons, a 20-character liquid crystal display, and a thermal printer. With the buttons, the user can tell the system to display or print various information such as the gasoline levels in the tanks or the time of day or the overall system status.

To figure out how much gasoline is in one of the tanks, the system first reads the level of two floats in the tank, one of which indicates the level of the gasoline and one of which indicates the level of the water that always accumulates in the bottom of such tanks. It also reads the temperature at various levels in the tank; gasoline expands and contracts considerably with changes in temperature, and this must be accounted for. The system should not set off the alarm just because the gasoline cooled off and contracted, thereby lowering the float.

None of this would be particularly difficult, except for the problem of cost that often arises in the context of embedded systems.

Cost

A gas station owner buys one of these systems only because some government agency tells him he has to. Therefore, he wants it to be as inexpensive as possible. Therefore, the system will be built with an extremely inexpensive microcontroller, probably one that barely knows how to add 8-bit numbers much less how to use the coefficient of expansion of gasoline in any efficient way. Therefore, the microprocessor in this system will find itself extraordinarily busy just calculating how much gasoline there really is down there; that calculation will turn into a processor hog.

A sketch of the underground tank monitor is in Figure 8.7. Figure 8.6 contains a more detailed description of what the underground tank monitor does.

Nuclear Reactor Monitor

Last, one very simple example from which we can learn a surprising amount is a hypothetical system that controls a nuclear reactor. Our hypothetical system must do many things, but the only aspect that will interest us is the part of the code that monitors two temperatures, which are always supposed to be equal. If they differ, it indicates that a malfunction in the reactor is sending it toward China. We'll revisit this system several times.

1.2 Typical Hardware

If you know generally what kinds of hardware parts are typically found in embedded systems, you can skip this section. Otherwise, read on for a summary of what usually inhabits one of these systems.

First, all of the systems need a **microprocessor**. The kinds of microprocessors used in embedded systems are quite varied, as you can see from Table 1.1, a list of some of the common microprocessor families and their characteristics.

Table 1.1 Microprocessors Used in Embedded Systems

Processor	Bus Width	Largest External Memory	Internal Peripherals	Speed (MIPS)
Zilog Z8 family	8	None on some models; 64 KB on others	2 timers	1
Intel 8051 family	8	64 KB program + 64 KB data	3 timers + 1 serial port	1
Zilog Z80 family	8	64 KB; 1 MB, sort of	Various	2
Intel 80188	8	1 MB	3 timers + 2 DMA channels	2
Intel 80386 family	16	64 MB	3 timers + 2 DMA channels + various others	5
Motorola 68000 family	32	4 GB	Varying	10
Motorola PowerPC family	32	64 MB	Many	75

(Note that the semiconductor companies all sell a variety of models of each microprocessor. The data in Table 1.1 are *typical* of these microprocessor families; individual microprocessors may differ considerably from what is shown in Table 1.1.)

An embedded system needs memory for two purposes: to store its program and to store its data. Unlike a desktop system, in which programs and data are stored in the same memory, embedded systems use different memories for each of the two different purposes. Because the typical embedded system does not have a hard disk drive from which to load its program, the program must be stored in the memory, even when the power is turned off. As you are no doubt aware, the memory in a desktop system forgets everything when the power is turned off. The embedded system needs special kinds of memory that will

remember the program, even with no power. Unfortunately, as we will discuss in Chapter 2, these special memories are not very suitable for data; therefore, embedded systems need some regular memory for that.

After a processor and memory, embedded systems are more noted for what they do not have than for what they do. Most embedded systems do not have the following:

- *A keyboard.* Some systems may have a few push buttons for user input; some—Telegraph, for example—do not have even that.
- *A screen.* Many systems, especially in consumer products, will have a liquid crystal display with two or three dozen characters. A laser printer, for example, commonly has a two-line status display with 10 or 12 characters on each line. Other systems do not even have this much output capability. Some may just have a few **light-emitting diodes** (those tiny lights you sometimes see on systems) to indicate certain basic system functions.
- *A disk drive.* The program is stored in the memory, and most embedded systems do not need to store much data on a permanent basis. Those that do typically use various kinds of specialized memory devices rather than disk drives.
- *Compact discs, speakers, microphones, diskettes, modems.* Most embedded systems have no need for any of these items.

What embedded systems very often do have are a standard serial port, a network interface, and hardware to interact with sensors and activators on equipment that the system is controlling.

Chapter Summary

- An embedded system is any computer system hidden inside a product other than a computer.
- You will encounter a number of difficulties when you write embedded-system software in addition to those you encounter when you write applications:
 - *Throughput*—Your system may need to handle a lot of data in a short period of time.
 - *Response*—Your system may need to react to events quickly.
 - *Testability*—Setting up equipment to test embedded software can be difficult.
 - *Debugability*—Without a screen or a keyboard, finding out what the software is doing wrong (other than not working) is a troublesome problem.

- *Reliability*—Embedded systems must be able to handle any situation without human intervention.
 - *Memory Space*—Memory is limited on embedded systems, and you must make the software and the data fit into whatever memory exists.
 - *Program Installation*—You will need special tools to get your software into embedded systems.
 - *Power Consumption*—Portable systems must run on battery power, and the software in these systems must conserve power.
 - *Processor Hogs*—Computing that requires large amounts of CPU time can complicate the response problem.
 - *Cost*—Reducing the cost of the hardware is a concern in many embedded system projects; software often operates on hardware that is barely adequate for the job.
- Embedded systems have a microprocessor and a memory. Some have a serial port or a network connection. They usually do not have keyboards, screens, or disk drives.

Hardware Fundamentals for the Software Engineer

If you're already familiar with digital hardware schematics, you can skip this chapter and the next. (If you are just impatient and want to get on with the software, you also can skip these two chapters, but if you know nothing about hardware, you may end up having to peek back at them to understand some of the material in the later chapters.)

Although a software engineer who writes only applications may spend an entire career and learn nothing about hardware, an embedded-systems software engineer usually runs up against hardware early on. The embedded-systems software engineer must often understand the hardware in order to write correct software; must install the software on the hardware; must sometimes figure out whether a problem is caused by a software bug or by something wrong in the hardware; may even be responsible for reading the hardware schematic diagram and suggesting corrections.

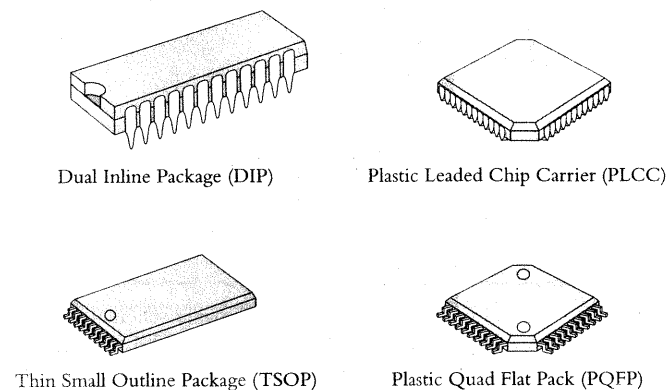
In this chapter and the next we will discuss the basics of digital hardware. These chapters will provide you with enough information to read the schematic diagrams for a typical embedded system well enough to be able to write the software and talk intelligently to the hardware engineers. There is not nearly enough information here for you to start designing your own hardware.

2.1 Terminology

Some Very Basic Terms

Most digital electronic circuits today are built with semiconductor parts called chips that are purchased from manufacturers specializing in building such parts.

Figure 2.1 Various Types of Chip Packages



The semiconductors themselves are encased in small, thin, square or rectangular black **packages** made of plastic or ceramics. To attach the semiconductors to the outside world, each package has a collection of **pins**, stiff metallic legs that protrude from the sides of the package. Depending upon the type of the part, there may be from 8 to 500 pins. (See Figure 2.1.) The chip manufacturers provide information about each of their products in documents called **data sheets**.

The most common mechanism to connect the chips to one another is the **printed circuit board** or **board**, a thin board typically made out of fiberglass with the required connections printed on it in copper. The chips are soldered to the appropriate locations on the board after it has been manufactured. Companies building embedded-system products typically must design their own boards, although many of them will subcontract their manufacture and assembly to companies that specialize in this.

Hardware engineers record their designs by drawing **schematic diagrams**, drawings that show each part needed in the circuit and the interconnections needed among them. An example of one is shown in Figure 3.20. In this chapter and in Chapter 3 we will discuss some of the symbols and conventions that are used on schematics. Note that schematic diagrams are not layouts showing where the parts are located on the board (although many schematics will contain notations to indicate where the parts are).

Some More Basic Terms

Most digital circuits use just two voltages to do their work:

- 0 volts, sometimes called **ground** or **low**.
- Either 3 volts or 5 volts, sometimes called **VCC** (which stands for **Voltage Connected to Collector**) or **high**.¹

At any given time, every signal in the circuit is at one of these two voltages (although there is some very short transition time from one voltage to another). For most circuit components, voltages within a volt or so of high or low are good enough. For example, “low” might be from 0 volts to 1 volt; “high,” from 3 to 5 volts; and from 1 volt to 3 volts might not be allowed. The entire activity of the circuit consists of the changing of the signals from high to low and from low to high as the various parts in the circuit signal one another.

With a few exceptions, whenever signals represent data or addresses, the low voltage represents a 0, and the high voltage represents a 1. In addition to data and addresses, however, every circuit contains many signals whose purposes are to indicate various conditions, such as “reset the microprocessor” or “get data from this memory chip.” These signals are said to be **asserted** when they are signaling whatever it is that they signal. For example, when the microprocessor wants to get data from a particular memory chip, the engineer must design the circuit to **assert** the “get data from this memory chip” signal. Some signals are asserted when they are high, and some are asserted when they are low.² You must read the schematic and the information in the data sheets about the parts in the circuit to determine which are which.

Most hardware engineers will assign a name to each signal in the circuit. For example, the data signals might be named D0, D1, D2, and so on. The address signals might be A0, A1, A2, and so on. The signal that indicates “read from memory now” might be named MEMREAD. Many careful engineers will give a special name to each signal that is asserted when it is low by starting or ending the name with an asterisk (*), ending the name with a slash (/), or

1. If the parts in your system have been built using **metal oxide semiconductor (MOS)** technology, you’ll sometimes hear the term **VDD** instead of VCC and **VSS** instead of ground.

2. The electronic properties of the semiconductor materials from which chips are made makes it “natural,” from the perspective of the chip designer, for certain signals to be asserted high and others low. Also, as we’ll see later when we examine open collector devices in Section 2.3, high and low have somewhat different properties when you connect chips to one another.

by putting a bar over the name. For example, a signal named MEMREAD/ or *MEMREAD would most likely be a signal that is set low to read from memory.

Chips have connections through which they expect to control the voltage level on the attached signal—**outputs**—and other connections through which they expect to sense the voltage level on the attached signal—**inputs**. Most signals are connected to the output of just one part in the circuit; each may be connected to the inputs of several parts in the circuit, however. The part whose output controls the voltage on a given signal is said to **drive** the signal. If no part on the circuit is driving a signal, then that signal is said to be **floating**. Its voltage will be indeterminate and may change as time passes. The results of a floating signal vary between harmless and disastrous, depending upon how the parts with inputs connected to the floating signal cope with the problem.

If two parts drive the same signal at the same time, things work pretty well as long as the two parts both drive high or both drive low. If one tries to drive one way and the other tries to drive the other, then the usual result is to destroy one (or both) of the parts. Usually the parts get very hot—hot enough to raise a blister on your thumb if you touch one of them—then they stop working for good. This is sometimes called a **bus fight**. Bus fights that last only a short time—say several nanoseconds—but that occur periodically, may not destroy the parts, but may cause the circuit to run unreliably and to become less reliable as time goes by. Bus fights invariably indicate an error in the hardware design.

2.2 Gates

A very simple part built from just a handful of semiconductor transistors is called a **gate**, or sometimes a **discrete**. In this section we cover some of the very basic gates used in typical digital hardware circuits. Although you can buy parts that contain just one gate each, chips that contain three, four, or even five or six of these very simple circuit elements are the norm.

Inverters, AND Gates, and OR Gates

Figure 2.2 shows the symbol that hardware engineers place on their schematic to indicate an **AND gate**. An AND gate is one whose output (shown at the right of the figure) is driven high if both of the inputs are high and whose output is driven low if either input is low or if both inputs are low. The table in Figure 2.2 shows this.

Figure 2.2 AND Gate

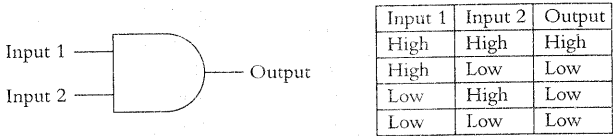


Figure 2.3 Multiple-Input AND Gates

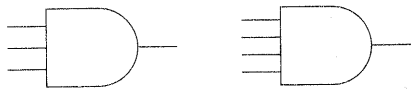
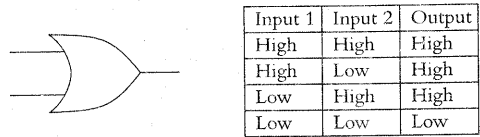


Figure 2.4 OR Gate



You can also get AND gates with three or even more inputs, as shown in Figure 2.3. The outputs of these gates are high if all of the inputs are high.

Figure 2.4 shows the symbol for an **OR gate**. An OR gate is one whose output (again, shown at the right of the figure) is driven high if either or both of the inputs are high and whose output is driven low only if both inputs are low. As with AND gates, you will find circuits with multiple-input OR gates.

Figure 2.5 shows the symbol for an **XOR gate** or **exclusive OR gate**. An XOR gate is one whose output (again, shown at the right of the figure) is driven high if one but *not* both of the inputs are high and whose output is driven low if both inputs are high or both are low.

Figure 2.6 shows the symbol for an **inverter**. Inverters are very simple: the output is driven low if the input is high and vice versa.

Figure 2.5 XOR Gate

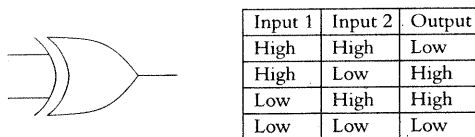


Figure 2.6 Inverter

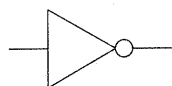
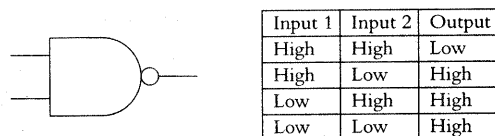


Figure 2.7 NAND Gate



The Bubble

The little loop, or **bubble**, on the inverter symbol is used in other schematic symbols to indicate that an input or an output is inverted (low when it would otherwise be high and vice versa). For example, the symbol in Figure 2.7 is the one for a gate whose output is the opposite of an AND gate: its output is low if both inputs are high and is high otherwise. This gate is called a **not-AND gate** or, more often, a **NAND gate**.

The bubble can be used for the inputs on a gate as well (see Figure 2.8). The operation of this gate is to invert each input, and then to feed the inverted inputs to a regular OR gate.

Occasionally, you'll even see the symbol shown in Figure 2.9. It's just the same as the inverter we saw before. The triangular part of this symbol represents

Figure 2.8 OR Gate with Negated Inputs

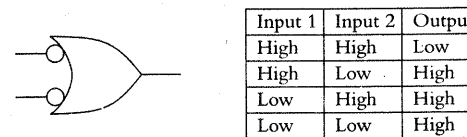
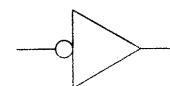


Figure 2.9 Another Inverter



a **driver**, a device that drives its output signal to match its input signal. The bubble indicates that the gate also inverts; for an inverter, it makes no difference whether the bubble is shown on the input or the output.

When the circuit is actually built, of course, the manufacturer will use the same sort of inverter, regardless of which symbol is on the schematic. Why would an engineer favor one symbol over another? Some engineers follow the convention that a signal that goes into or comes out of a bubble is one that is asserted low. These engineers might use the symbol in Figure 2.9 if the input signal (on the left) is one that asserts low and would use the one in Figure 2.6 if the input signal is one that asserts high. (Note, however, that this is not the only convention and that many engineers will use the symbol in Figure 2.6 consistently, regardless of which signals assert high or low. Note also that some engineers are careless about this in any case.)

In a similar vein, a NAND gate and an OR gate with inverted inputs also are identical. You can convince yourself of this by reviewing the truth tables in Figure 2.7 and Figure 2.8: they're the same. As with the inverter, the same part will go into the circuit, no matter which symbol is on the schematic. Many engineers will use the symbol for the OR gate with inverted inputs if the underlying operation is more 'or'-like ("I want the output to assert if input 1 is asserted *or* input 2 is asserted, ignoring the issues of low-asserting signals") and will use the NAND symbol if the underlying operation is more 'and'-like. Again, however, this is not the only convention.

Figure 2.10 Another Circuit



You can invent schematic symbols with bubbles on some inputs but not others, but the problem is that no manufacturer makes parts that correspond to these symbols. See Figure 2.10.

2.3 A Few Other Basic Considerations

Power and Decoupling

For the most part, the problems of providing power to a circuit are beyond the scope of this book. However, there are several useful things to know.

The first thing to know is that, with very few exceptions, each chip in any circuit has a **power pin** (sometimes called a **VCC pin**), which must be connected to a signal that is always high (at VCC), and a **ground pin**, which must be connected to a signal that is always low. These two are *in addition* to the pins for the various input and output signals, and they provide power to run the part itself. For example, the standard “7400” part has four NAND gates in it. Each NAND gate has two inputs and one output, for a total of 12 connections. The 7400 package has 14 pins: the 12 signal connections plus a power pin and a ground pin. The connections to the power pin and the ground pin usually do *not* appear on circuit schematics, but they must be made for the circuit to work. In fact, one common test when a circuit isn’t running is to use a voltage meter to ensure that power and ground are connected as required to each part.

When it is necessary to show VCC and ground connections on a schematic, engineers use the symbols shown in Figure 2.11.

One problem that hardware engineers must solve is that most chips use much more power at some times than at others. Typically, if a chip must change many of its output signals from high to low or from low to high at the same time, that chip will need a lot of power to change these signals. In fact, they need more power than the skinny conductors on the average board can provide quickly. Unless you take steps to combat this problem, you end up with what amounts to a localized brownout for a few microseconds. Most types of chips

Figure 2.11 Power and Ground Symbols

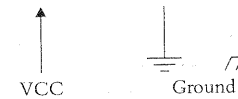


Figure 2.12 Capacitor



stop working temporarily if the voltage drops by 10 percent or so, even for just a few microseconds, so circuits subject to brownouts fail often. To deal with this, engineers add **capacitors** to the circuit, with one end of the capacitor connected to the signal providing power and the other to the signal providing ground. A capacitor is a device that stores a small amount of electricity, like a minuscule rechargeable battery.

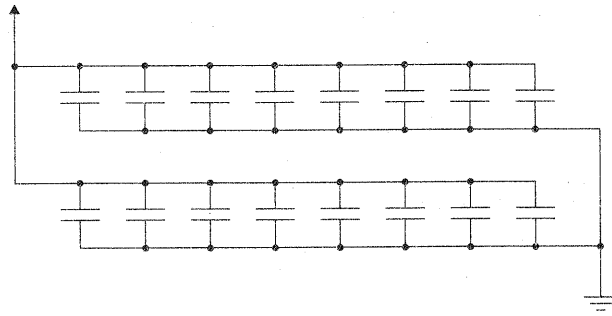
If some part in the vicinity of a capacitor suddenly needs a lot of power, and the voltage begins to fall because of that, the capacitor will give up its stored electricity to maintain the voltage level. At other times, the capacitor quietly recharges itself. A capacitor can smooth over brownouts that last up to a few microseconds, enough to take care of the voltage drops caused when other parts of the circuitry suddenly demand a lot of power.

A capacitor used this way is called a **decoupling capacitor**. Decoupling capacitors are usually scattered around the circuit, since they need close proximity to the parts needing power to do their work effectively. They are often shown on the schematic. The symbol used for a capacitor is shown in Figure 2.12. On many schematics, you’ll see something like the diagram shown in Figure 2.13, which indicates that a collection of decoupling capacitors needs to be placed in the circuit.

Open Collector and Tri-Stating Outputs

One special class of outputs, the **open collector outputs**, allows you to attach the outputs of several devices together to drive a single signal. Unlike the usual outputs, which drive signals high or drive them low, the open collector outputs

Figure 2.13 Decoupling Capacitors



can drive their outputs low or let them float. With open collector outputs, there is no such thing as a bus fight. If several open collector outputs are attached to the same signal, then the signal goes low if *any* of the outputs is driving low.

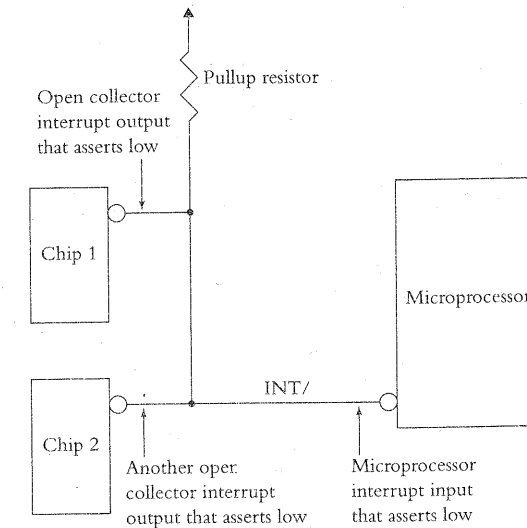
Figure 2.14 shows how you might use devices with open collector outputs. If your microprocessor has only one input for an interrupt signal but you have two devices that need to signal interrupts, and if the following two conditions hold, then the circuit in Figure 2.14 will work.

- The interrupt input on the microprocessor is asserted when it is low.
- The interrupt outputs on the two devices are asserted when they are low and they are both open collector outputs.

(We'll discuss interrupts further in Section 3.4 and in Chapter 4; here we'll just explain how the circuit works.) If one of the devices wants to signal the interrupt, then it drives its interrupt output low, the signal INT/ will go low, and the microprocessor will respond to the interrupt signal. (Then it's a small matter of software to figure out which device signaled the interrupt.) If neither device wants to signal the interrupt, each will let its output float, the **pullup resistor** will cause INT/ to go high, and the microprocessor will sense its interrupt input as not asserted.

Note that the pullup resistor is necessary for this circuit to work; otherwise, INT/ would float when neither device wanted to interrupt. The pullup resistor ensures that INT/ goes high in this situation. Note also that you cannot omit the resistor and connect the INT/ signal directly to VCC. If you did this, then you would have a bus fight on your hands as soon as one of the devices tried

Figure 2.14 Open Collector Outputs

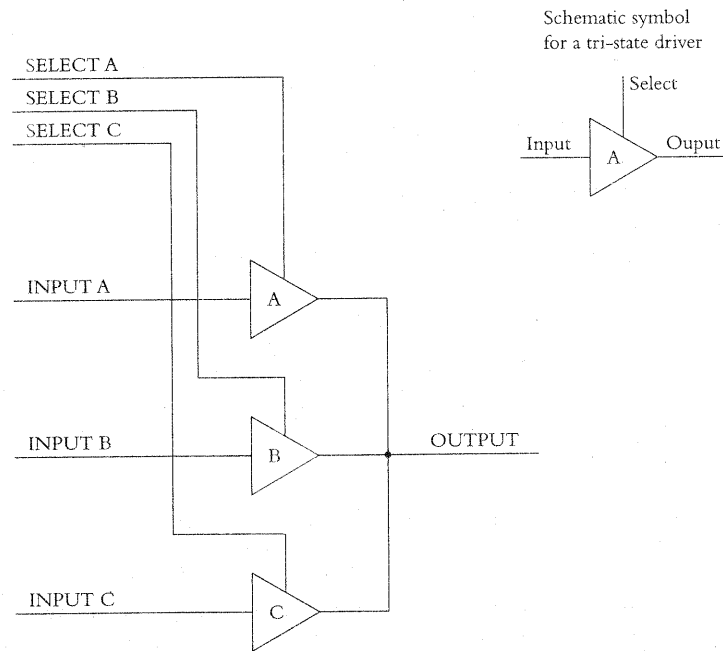


to drive INT/ low, since the parts that provide electrical power to your circuit would then try to keep INT/ high. The resistor provides a necessary buffer to prevent this bus fight. See the following section on Floating Signals (or Multiply Driven Signals) for more discussion of pullup resistors.

Standard parts either drive their output signals high or drive them low. Open collector outputs drive their signals low or let them float. Another class of outputs can drive signals high, drive them low, or let them float. Since letting the output signals float is a third possible state (after driving them high or driving them low), these outputs are usually called the **tri-state outputs**, and letting the signals float is called **tri-stating** or going into the **high impedance state**. Tri-state devices are useful when you want to allow more than one device to drive the same signal.

The circuit shown in Figure 2.15 shows a simple use for tri-state devices. The triangular symbol in that schematic is a **tri-state driver**. A tri-state driver works like this: when the select signal is asserted, then the tri-state driver output will be the same as the input; when the select signal is not asserted, the output on the tri-state driver floats. In the circuit in Figure 2.15, if SELECT A is asserted

Figure 2.15 A Circuit Using Tri-State Drivers



and SELECT B and SELECT C are not, then the tri-state driver A can drive the OUTPUT signal high or low; tri-state drivers B and C do not drive it. The OUTPUT signal will reflect the input of the driver for which the select signal is asserted. Note that you can get tri-state drivers whose select signals assert high and others whose select signals assert low.

Consider this extremely common situation: in your circuit a microprocessor, a memory chip, and some I/O device must send bytes to one another. You could have multiple sets of data signals: eight signals for the memory to send bytes to the microprocessor, eight more for the microprocessor to send bytes to the I/O, and so on. However, if all the devices can tri-state their data output signals, you can use just one collection of data signals to interconnect all of them. When the microprocessor wants to send data to the memory, the I/O device and the

memory tri-state their outputs, and the microprocessor can drive its data onto the data signals. Whenever one of the devices wants to send data, the other two tri-state their outputs while the sending device drives the data signals.

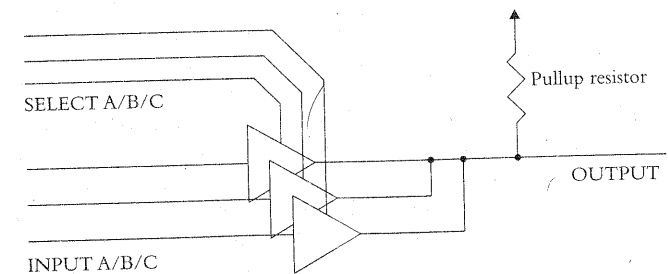
Figure 2.15, by the way, illustrates a common convention used on schematic diagrams: the dot. Where the three tri-state driver outputs intersect, the solid black dot indicates that these signals are to be connected to one another. The usual convention on schematics is that two crossing lines on a schematic are *not* connected without a dot. So, for example, in Figure 2.15 the INPUT A signal is not attached to SELECT B or SELECT C, nor is INPUT B connected to SELECT C, even though the lines representing these signals cross one another on the left hand side of the Figure 2.15 schematic.

Floating Signals (or Multiply Driven Signals)

The circuit in Figure 2.15 has two potential problems. First: what happens if none of the select signals is asserted? In this case, none of the drivers drives the OUTPUT signal, and that signal floats. Whether it is high or low or somewhere in between is indeterminate, depending upon transient conditions in the drivers and in the parts that are sensing the signal. If the circuit's function depends upon other parts sensing this signal as high or low, then the behavior of the entire circuit may become random.

The usual solution to this sort of problem is to put a part on the circuit that drives it high or low by default. Figure 2.16 shows the same circuit as shown in Figure 2.15, but with an added **pullup resistor**, a resistor with one end connected to VCC and one end connected to a signal. When none of the select

Figure 2.16 A Circuit With a Pullup



lines is asserted and none of the drivers drives the OUTPUT signal, enough electrical current will flow through the resistor to drive the voltage high. When one of the drivers drives the OUTPUT signal low, current still flows through the resistor, but not enough to raise the voltage enough to matter. As is apparent, you could just as well attach the resistor to ground, and the OUTPUT signal would go low if none of the drivers drive it. In this case the resistor would be called a **pull-down resistor**.

The second problem arises if more than one of the select signals is asserted and therefore more than one of the drivers drive the output signal. Unlike open-collector devices, tri-state devices *can* and *will* have bus fights if one of them tries to drive a signal low and another tries simultaneously to drive that signal high. Tri-state devices can overheat and burn up in bus fights just like regular parts. If the software controls the select signals in Figure 2.16, you must ensure that no two of the select lines are ever asserted simultaneously. If hardware controls them, then the hardware engineer must ensure this.

Signal Loading

Examine the circuit in Figure 2.17, particularly the OVERLOADED signal, and look for a potential problem.

The problem is that the output signal from the inverter in the lower left corner of the figure is connected to the input signals of an awful lot of other parts. Any kind of part—the inverter that drives OVERLOADED as well as any other—can drive only a limited amount of electrical current on its various output signals. Each of the inputs attached to OVERLOADED absorbs a certain amount of current in the process of detecting whether the signal is high or low. If the inputs attached to OVERLOADED absorb more current than the inverter can drive onto OVERLOADED, the circuit won't work. This is the **loading problem**.

Manufacturers provide data about each part that indicates how much current it can drive out of its outputs and how much current is required for its inputs. Hardware engineers must ensure that the outputs driving each signal in the circuit can generate enough current to make all of the inputs happy. As a software engineer, you should not have to worry about this problem, but you'll occasionally see peculiar things on the schematics of your systems that will turn out to be solutions to this problem. It is useful to be familiar with the common solutions so as not to be puzzled by them when you encounter them.

One common solution to the loading problem is shown in Figure 2.18. The added part in that figure is called a **driver**. Its output is the same as its input.

Figure 2.17 An Overloaded Circuit

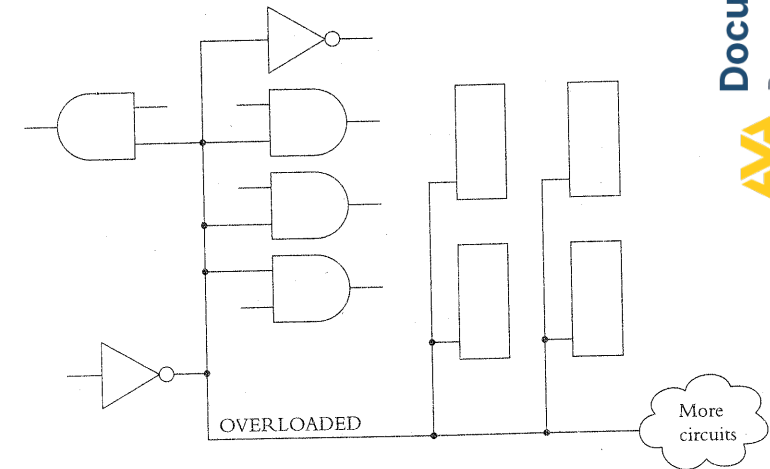


Figure 2.18 A Circuit Not Overloaded Anymore

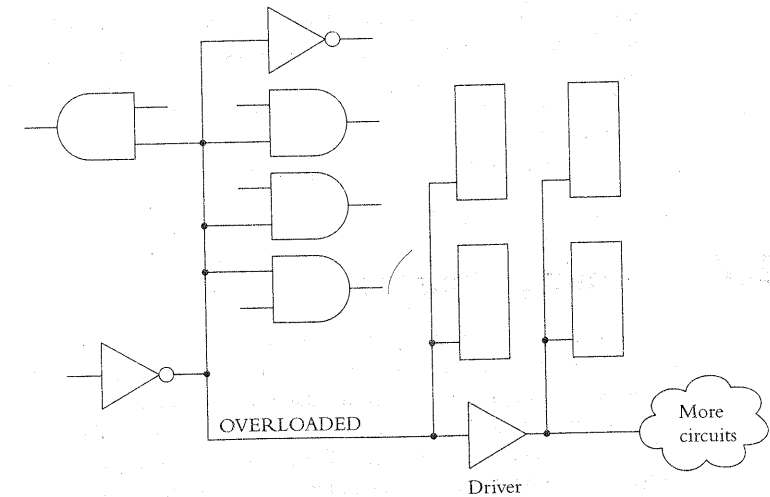
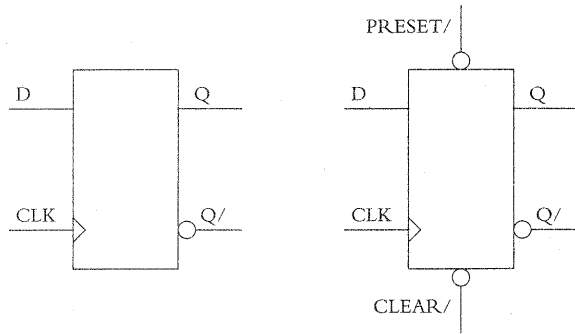


Figure 2.21 D Flip-Flop



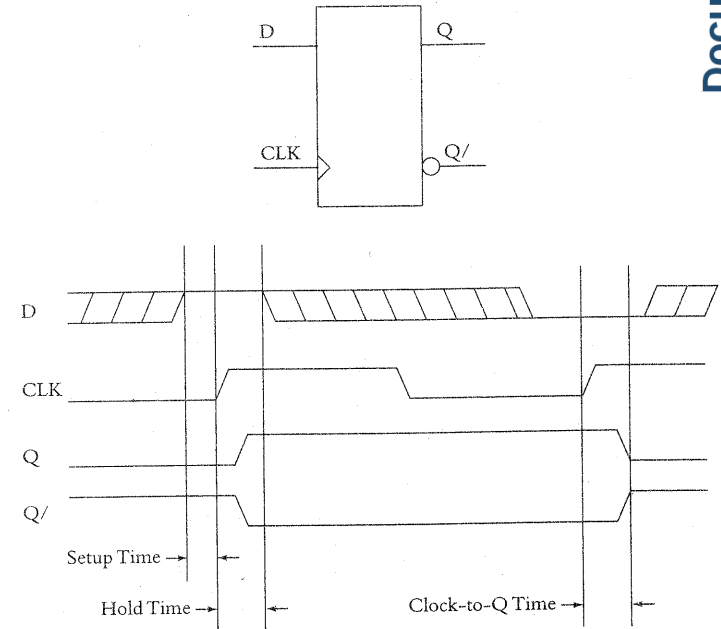
is driven low again and then high again. The $Q/$ signal is, as its name implies, the inverse of the Q signal. Some D flip-flops also have a $CLEAR/$ signal and a $PRESET/$ signal. On those parts, asserting the $CLEAR/$ signal forces the Q signal low, no matter what the CLK and D signals are doing; asserting the $PRESET/$ signal forces the Q signal high.

A D flip-flop is essentially a 1-bit memory. Its Q output remembers the state of the D input at the time that the CLK input rises. A similar part, called a *latch*, also can be used as a memory. A latch is the same as a D flip-flop in that it captures the state of the D input on the rising edge of the CLK input. However, the Q output in a latch is driven to be the same as the D input whenever the CLK input is low, whereas the Q output in a D flip-flop does not change until the rising edge of the CLK input.

Hold Time and Setup Time

D flip-flops have more interesting timing diagrams, because the timing relationship between the two inputs is critical. (See Figure 2.22.) At the rising edge of the CLK signal, the Q output signal will take on the value of the D input signal. However, there is a minimum amount of time, both before and after the rising edge of the CLK signal, during which the D input must remain constant for the D flip-flop to run reliably. The time before the rising edge of CLK during which the D input must remain constant is called the **setup time**. The time after the rising edge of CLK during which the D input must remain con-

Figure 2.22 Timing Diagram for a D Flip-Flop



stant is called the **hold time**. The timing diagram for a D flip-flop indicates the minimum required for these two times (probably just a few nanoseconds). The timing diagram also indicates the maximum amount of time, called the **clock-to-Q time**, after the rising edge of CLK before the Q output is guaranteed to be valid. Sometimes this amount of time is different, depending upon whether Q is going high or going low. Note that the terms setup time, hold time, and clock-to-Q time are used for all kinds of parts, even for parts with no signal called Q .

In the timing diagram in Figure 2.22, the shaded area of the D signal indicates a time period during which it does not matter what the input does. Timing diagrams often use this convention. Note also that Figure 2.22 shows two complete timing cycles, each with a rising edge on CLK : the one on the left in which D is high and Q changes to high, and the one on the right in which D is

Figure 2.23 A Clock Signal



low and Q changes to low. Each cycle will have a setup time, a hold time, and a clock-to-Q time, but for clarity some of the times are shown on one cycle and some on the other. This is also a common timing diagram convention.

Clocks

Obviously, for a circuit to do anything interesting, the levels on the signals have to change. Some embedded-system products do things only when external events cause a change on one of the inputs, but many circuits need to do things just because time is going by. For example, a microprocessor-based circuit must go on executing instructions even if nothing changes in the outside world. To accomplish this, most circuits have a signal called the clock. The timing diagram for the clock is very simple and is shown in Figure 2.23.

The purpose of the clock signal is to provide rising and falling edges to make other parts of the circuit do their jobs.

The two types of parts used to generate clock signals are **oscillators** and **crystals**. An oscillator is a part that generates a clock signal all by itself. Oscillators typically come in metallic packages with four pins: one for VCC, one for ground, one that outputs the clock signal, and one that is there just to make it easier to solder the oscillator securely onto the printed circuit board. A crystal has just two signal connections, and you must build a little circuit around it to get a clock signal out. Many microprocessors have two pins on them for attachment to a circuit containing a crystal.

You can buy oscillators and crystals in a wide range of frequencies. In picking a frequency consider first that since other parts in the circuit must react to the clock signal, the clock signal must be slow enough that the other parts' timing requirements are met. For example, when you buy a microprocessor that is the 16-megahertz model, this means that that microprocessor will work with a clock signal that is 16 megahertz (16 million cycles per second), but not with one that is faster. (Note, however, that microprocessors frequently need a crystal that oscillates at some multiple of the actual clock speed that the microprocessor uses.)

The second consideration in picking a frequency for an oscillator or crystal is that it is often desirable to have the clock signal frequency be an integer multiple of the data rate on your network or serial port or other communication medium. It's a lot easier to divide the clock signal by an integer to create another signal at the correct data rate than it is to divide by some fraction. (It's even easier to divide by some power of two, if you can get away with that.)

2.5 Memory

In this section, we'll discuss the memory parts typically found in an embedded-system circuit. Memories of all kinds are sold in a variety of widths, sizes, and speeds. For example, a "8 x 512 KB 70 nanosecond memory" is one that has 512 KB³ storage locations of 8 bits each that can respond to requests for data within 70 nanoseconds. After you decide what kind of memory is useful for your system, you buy the size and speed that you need.

Read-Only Memory

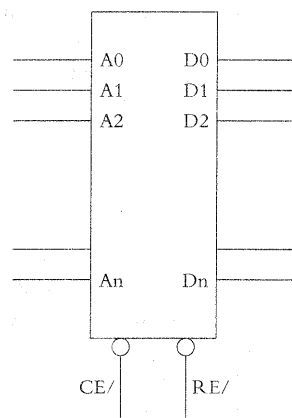
Almost every computer system needs a memory area in which to store the instructions of its program. This must be a **nonvolatile memory**, that is, one that does not forget its data when the power is turned off. In most embedded systems, which do not have a disk drive or other storage medium, the entire program must be in that memory. In a desktop system, enough program must be in memory to start up the processor and read the rest of the program from a disk or a network.

Most computer systems use some variant of **Read-Only Memory**, or **ROM** (pronounced "rahm," just like you would expect) for this purpose. The characteristics of ROM are the following:

- The microprocessor can read the program instructions from the ROM quickly, typically as fast as the microprocessor can run the program.
- The microprocessor cannot write new data to the ROM; the data is unchangeable.
- The ROM remembers the data, even if the power is turned off.

3. For memory sizes, KB invariably means 1,024. Therefore, for example, 512 KB means 512 x 1,024, or 524,288.

Figure 2.24 Typical ROM Chip Schematic Symbol



When the power is first turned on, the microprocessor will start fetching the (still-remembered) program from the ROM.

Figure 2.24 shows the typical pins you find on a ROM part. The signals from A0 to An are the **address signals**, which indicate the address from which the processor wants to read. The number of these signals depends upon the size of the ROM. (You need more address lines to select a particular address in a larger ROM.) The signals from D0 to Dn are the **data signals** driven by the ROM. There are typically eight or sixteen of these. The CE/ signal is the **chip enable signal**, which tells the ROM that the microprocessor wants to activate the ROM. It is sometimes called the **chip select signal**. The ROM ignores the address signals unless the chip enable signal is asserted. The RE/ signal is the **read enable signal**, which indicates that the ROM should drive its data on the D0 to Dn signals. The read enable signal is often called **output enable**, or OE/, instead. Unless both CE/ and RE/ are asserted, the ROM tri-states its output data signals.

Although it may seem redundant, it is normal for ROM parts to have both a chip enable signal and a read enable signal. The purpose for this will become apparent in the next chapter, when we discuss bus architectures. Note that it is very common for these enable signals to be asserted low.

Figure 2.25 Timing Diagram for a Typical ROM

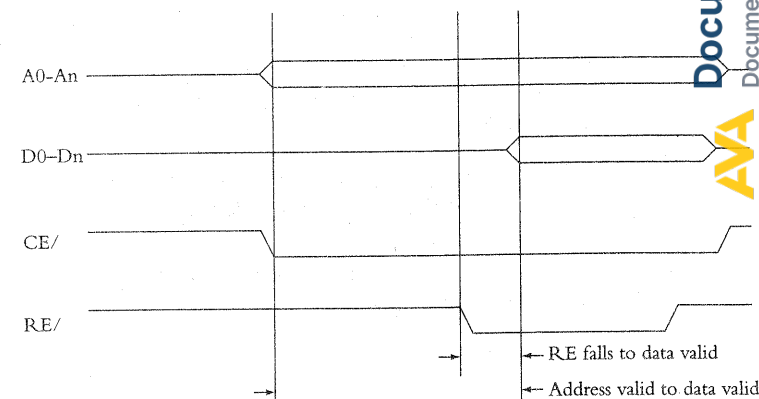


Figure 2.25 shows the timing diagram for a typical ROM. This timing diagram illustrates several other conventions often used in timing diagrams. With parts such as memory chips, which have multiple address or data signals, it is common to show a group of such related signals on a single row of the timing diagram. With such a group of signals, a single line that is neither high nor low indicates that the signals are floating or changing. When the signals take on a particular value, that is shown in the timing diagram with two lines, one high and one low, to indicate that each of the signals has been driven either high or low.

The expected sequence of events when a microprocessor reads from a ROM is as follows:

- The microprocessor drives the address lines with the address of the location it wants to fetch from the ROM.
- At about the same time, the chip enable signal is asserted.
- A little while later the microprocessor asserts the read line.
- After a propagation delay, the ROM drives the data onto the data lines for the microprocessor to read.
- When the microprocessor has seen the data on the data lines (an event not shown on this timing diagram; that event would appear on the microprocessor's timing

diagram), it releases the chip enable and read enable lines, and the ROM stops driving the data onto the data lines.

Most ROM chips also can handle a cycle in which the read enable line is asserted first and the chip enable line is asserted second, but they often respond much more slowly in this situation. The typical critical times for a ROM chip are the following:

- How long is it between the time when the address is valid and the chip enable signal is asserted and the time when the data signals driven by the ROM are valid?
- How long is it between the time when read enable is asserted and the time when the data signals driven by the ROM are valid?

ROM Variants

All sorts of ROMs are available. The data in a first kind of ROM is written into it at the semiconductor factory when that ROM is built; it can never be changed. Some people use the term **masked ROM** for this sort of ROM; others just call it ROM.

The next kind is **Programmable Read-Only Memory**, or **PROM**. PROMs are shipped blank from the factory, and you can write a program into them in your office with a **PROM programmer** or **PROM burner**, a tool made for that purpose. It takes only a matter of seconds to write a program into a PROM, but you can only write into a PROM once. If a program in a PROM has a mistake, you throw the PROM away, fix the program, and write the new program into a new PROM. PROM programmers are relatively inexpensive, selling for as little as \$100.

The next variant is **Erasable Programmable Read-Only Memory**, or **EPROM** ("ee-prahm"). EPROMs are like PROMs, except that you can erase them and reuse them. The usual way to erase an EPROM is to shine a strong ultraviolet light into a window on the top of the chip. **EPROM erasers**, boxes with ultraviolet lights in them, are also widely available and inexpensive. The only sophisticated thing about an EPROM eraser is that it must be designed to keep you from looking into the ultraviolet light by mistake and damaging your eyes. It usually takes an EPROM eraser 10 to 20 minutes to erase an EPROM.

The next variant on ROM is **flash memory**, sometimes called **flash**. Flash memories are similar to PROMs, except that they can be erased and rewritten by presenting certain signals to their input pins. Therefore, the microprocessor

itself can change the program in the flash. However, there are a few limitations of flash memory that you should know about:

- You can write new data into flash memory only a certain number of times before it wears out, typically on the order of 10,000 times.⁴
- In most flash memories you have to write a whole block of data, say 256 bytes or maybe even 4K bytes, at one time. There is no way to write just 1 byte or 4 bytes.
- The writing process is very slow (unlike the reading process, which is fast), taking on the order of several milliseconds to write a new block of data into the flash.
- The microprocessor usually can't fetch instructions from the flash during the several milliseconds that it takes to write new data into the flash, even if the part of the flash that is changing does not include the program. Therefore, the flash-programming program itself has to be stored somewhere else, at least when it is actually running.

For these reasons, the most typical use of flash memory is to store a program or rarely changed configuration data such as an IP address or the date on which the product should next be serviced and the diagnostic programs run.

The next variant is **Electrically Erasable Read-Only Memory**, or **EEROM** ("ee-ee-rahm" or "double-ee rahm"), sometimes called **EEPROM** (the P in the middle standing for "programmable," as you might guess). EEROM is very similar to flash memory, except that

- Both the writing process and the reading process are very slow in an EEROM. In fact, some EEROMs require that you write a little software routine to get data into and out of them one *bit* at a time.
- EEROMs often store only a very little data, often less than 1 K or so.
- You can write new data into an EEROM only a certain number of times before it wears out, but that number is often on the order of millions of times, so in many applications the limit doesn't matter.

Because of these characteristics, EEROM is useless for storing a program. It is used instead to store configuration information that might change relatively

4. All of the quantitative characteristics mentioned in this book about memory parts were current when the book was written. However, as this is an area of rapid development and evolution, you should assume that they may have changed by now.

frequently but that the system should recover on power-up; for example, as a network address, data rates, user names, number of pages printed, miles driven, etc.

See Table 2.1 for a comparison of the various kinds of memory.

Table 2.1 Types of Memory

Technology	Read Speed	Write Speed	Write Times	Comments
ROM (masked ROM)	Fast	N/A	0	ROM is useful for programs. It is programmed at the semiconductor factory. After an initial setup charge, ROMs are the least expensive type of permanent memory, and they are thus the best choice for a product with large volumes. In general, although they are not quite as fast as RAMs, ROMs are still fast enough to allow most microprocessors to execute programs directly from them.
PROM	Fast	N/A	1	PROM also is useful for programs. It is shipped from the factory blank, and you use a PROM programmer to program it. PROM is useful for products with lower volumes, since there is no setup charge, but it is more expensive than ROM.
EPROM	Fast	N/A	Many	EPROM is also shipped from the factory blank and is programmed with a PROM programmer. It can be erased by shining a strong ultraviolet light on it for 10 or 20 minutes and then reused; it is therefore useful when you are debugging a program.

Table 2.1 Types of Memory (*Continued*)

Technology	Read Speed	Write Speed	Write Times	Comments
Flash	Fast	Slow	10,000	Flash is useful for storing programs. The principal advantage of flash over the various other kinds of program memory is that it can be written to even after the product is shipped; for example, to upgrade to a new software version. Since it cannot be written to quickly, however, it is unsuitable for rapidly changing data. You <i>can</i> store data in flash, but you cannot change that data very often.
EEROM	Slow	Slow	1,000,000	EEROM is useful for storing data that must be remembered when the power goes off. Since both reading from and writing to EEROMs are slow processes, EEROMs are not suitable for programs or for working data.
RAM	Very fast	Very fast	Infinite	RAM is useful for data. Also, some very fast microprocessors would be slowed down if they executed the program directly from any flavor of ROM; in these cases, it is sometimes useful to copy the program from ROM to RAM at power-up time.

Random Access Memory

Every computer system needs a memory area in which to store the data on which it is working. This memory area is almost invariably made up of **Random Access Memory**, or RAM ("ram"). The general characteristics of RAM are listed below:

- The microprocessor can read the data from the RAM quickly, faster even than from ROM.
- The microprocessor can write new data to the RAM quickly, erasing the old data in the RAM as it does so.
- The RAM forgets its data if the power is turned off.

Obviously, the RAM is not a good place for a bootstrap program, because it would be forgotten on power failure. However, RAM is the only possible place to store data that needs to be read and written quickly.

Computer systems use two types of RAM: **static RAM** and **dynamic RAM**. Static RAM remembers its data without any assistance from other parts of the circuit. Dynamic RAM, on the other hand, depends on being read once in a while; otherwise, it forgets its data. To solve this problem, systems employing dynamic RAM use a circuit—often built into the microprocessor—called **dynamic RAM refresh**, whose sole purpose is to read data from the dynamic RAM periodically to make sure that the data stays valid. This may seem like a lot of complication that you can avoid by using static RAM instead of dynamic RAM, but the payoff is that dynamic RAM is comparatively cheap.

Static RAM parts look much like ROM parts, except that they have a **write enable signal** in addition to the other signals, which tells the RAM when it should store new data. Dynamic RAM is more complex and is quite different; a discussion of how circuits containing dynamic RAM must be built is beyond the scope of this book.

Chapter Summary

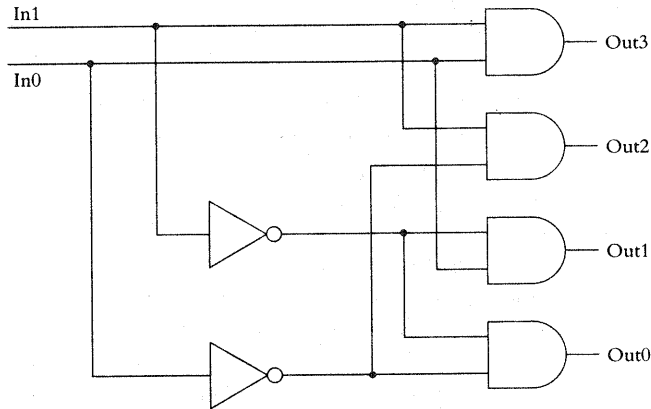
- Most semiconductor parts, **chips**, are sold in plastic or ceramic packages. They are connected to one another by being soldered to **printed circuit boards**.
- Electrical engineers draw **schematic diagrams** to indicate what parts are needed in each circuit and how they are to be connected to one another. Names are often assigned to signals on schematics.
- Digital signals are always in one of two states: **high** and **low**. A signal is said to be **asserted** when the condition that it signals is true. Some signals are asserted when they are high; others, when they are low.
- Each chip has a collection of pins that are inputs, and a collection that are outputs. In most cases, each signal must be driven by exactly one output, although it can be connected to multiple inputs.

- The standard semiconductor **gates** perform Boolean NOT, AND, OR, and XOR functions on their inputs.
- In addition to their input and output pins, most chips have a pin to be connected to VCC and a pin to be connected to ground. These pins provide power to the chip.
- **Decoupling capacitors** prevent local brownouts in a circuit.
- A signal that no output is driving is a **floating signal**.
- **Open collector devices** can drive their outputs low or let them float but they cannot drive them high. You can connect multiple open collector outputs to the same signal; that signal will be low if any output is driving low.
- **Tri-state devices** can drive their outputs high or low or let them float. You can connect multiple tri-state outputs to the same signal, but you must ensure that only one of the outputs is driving the signal at any one time and that the rest are letting the signal float.
- A dot on a schematic indicates that crossing lines represent signals that are to be connected to one another.
- A single output can drive only a limited number of inputs. Too many inputs leads to an overloaded signal.
- **Timing diagrams** show the timing relationship among events in a circuit.
- The various important timings for most chips are the **hold time**, the **setup time**, and the **clock-to-Q time**.
- **D flip-flops** are 1-bit memory devices.
- The most common types of memory are **RAM**, **ROM**, **PROM**, **EPROM**, **EEROM**, and **flash**. Since they each have unique characteristics, you will use them for different things.

Problems

1. In what kind of memory would you store each of the following?
 - The program for an intelligent VCR of which your company hopes to sell 10 million units.
 - A user-configurable name for a printer attached to a network that the printer should remember even if the power fails.
 - The program for a beta version of an x-ray machine that your company is about to ship to several hospitals on an experimental basis.
 - The data that your program just received from the network.

Figure 2.26 Circuit for Question 3



2. Write out the truth table for a three-input AND gate.
3. What does the circuit in Figure 2.26 do?
4. You can buy a three-input NAND gate, but nobody makes a three-input NAND gate such as the one shown in Figure 2.10, in which some of the inputs are negated. How would you expect to see that circuit really appear on a schematic?
5. What does the circuit in Figure 2.27 do?
6. What does the circuit in Figure 2.28 do? Why would anyone do this?
7. Examine the circuit in Figure 2.29. The idea is that the circuitry on the left-hand side is always running, but the circuitry on the right-hand side gets turned on and off from time to time to save power. The capacitor shown in the middle of the diagram is intended to cushion the voltage when the switch is closed. What is wrong with this design? What will the symptoms most likely be? How should it be fixed?
8. Why does the circuit in Figure 2.19 solve the loading problem? How does the circuit in Figure 2.19 compare to the circuit in Figure 2.18?
9. What does the timing diagram for a static RAM look like? Remember to include both a read cycle and a write cycle.

Figure 2.27 Circuit for Question 5

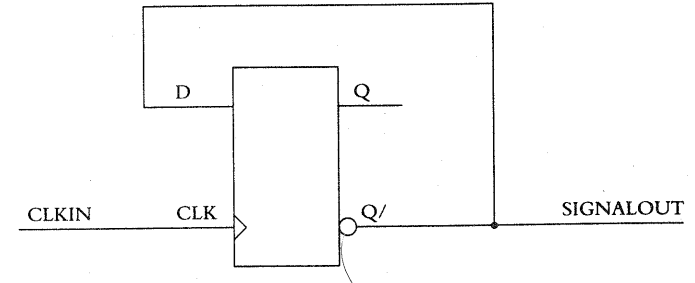


Figure 2.28 Circuit for Question 6

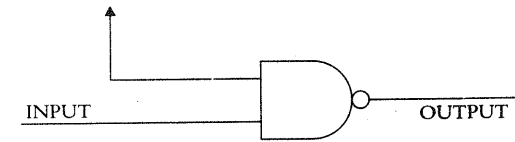
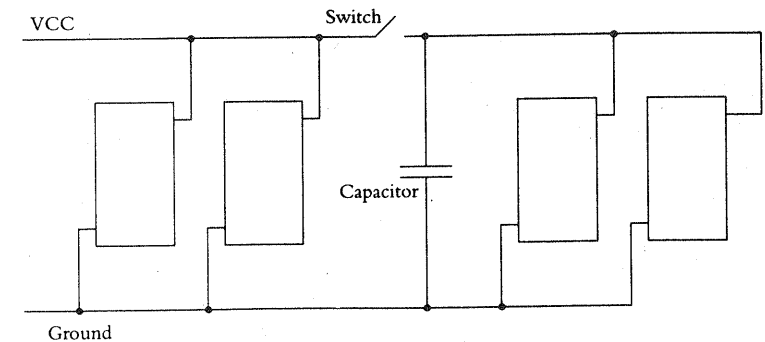


Figure 2.29 Circuit for Question 7



Advanced Hardware Fundamentals

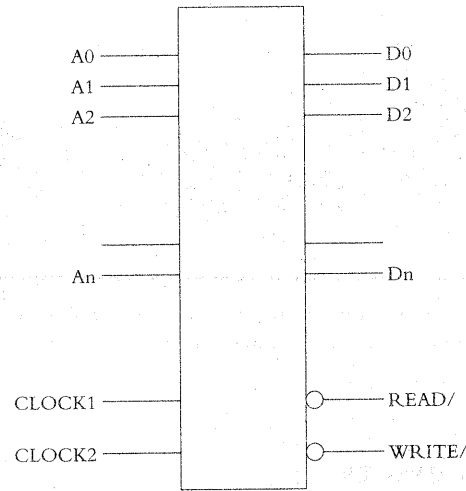
This chapter is a continuation of the previous one. We'll discuss the various parts you will commonly find in an embedded-system circuit.

3.1 Microprocessors

Microprocessors come in all varieties from the very simple to the very complex, but in the fundamental operations that they perform they are very similar to one another. In this section, we will discuss a very basic microprocessor, so basic that no one makes one quite this simple. However, it shares characteristics with every other microprocessor. It has the following signals, as shown in Figure 3.1:

- A collection of address signals it uses to tell the various other parts of the circuit—memory, for example—the addresses it wants to read from or write to.
- A collection of data signals it uses to get data from and send data to other parts in the circuit.
- A READ/ line, which it pulses or **strokes** low when it wants to get data, and a WRITE/ line, which it pulses low when it wants to write data out.
- A clock signal input, which paces all of the work that the microprocessor does and, as a consequence, paces the work in the rest of the system. Some microprocessors have two clock inputs to allow the designer to attach the crystal circuits discussed in Chapter 2 to them.

Figure 3.1 A Very Basic Microprocessor



These signals should look familiar from our discussions of memory chips in Chapter 2.

Most microprocessors have many more signals than this, and we'll discuss some of them later in the chapter. However, the above collection is all that the microprocessor needs in order to fetch and execute instructions and to save and retrieve data.

Some people use the term **microcontroller** for the very small end of the range of available microprocessors. Although there is no generally accepted definition for microcontroller, most people use it to mean a small, slow microprocessor with some RAM and some ROM built in, limited or no capability for using RAM and ROM other than what is built in, and a collection of pins that can be set high or low or sensed directly by the software. Since the principles of programming a microcontroller are the same as those for programming a microprocessor, and since manufacturers build these parts with every combination of capabilities imaginable, in this book we will use the term **microprocessor** to mean both.

3.2 Buses

Let's construct a very simple system from just a microprocessor, a ROM, and a RAM. Let's suppose that

- All three parts have eight data signals, D0 through D7.
- The microprocessor can address 64 K of memory and thus has 16 address lines A0 through A15.
- The ROM and the RAM each have 32 K and thus have 15 address lines each, A0 through A14.

Examine Figure 3.2 for a schematic of such a system.

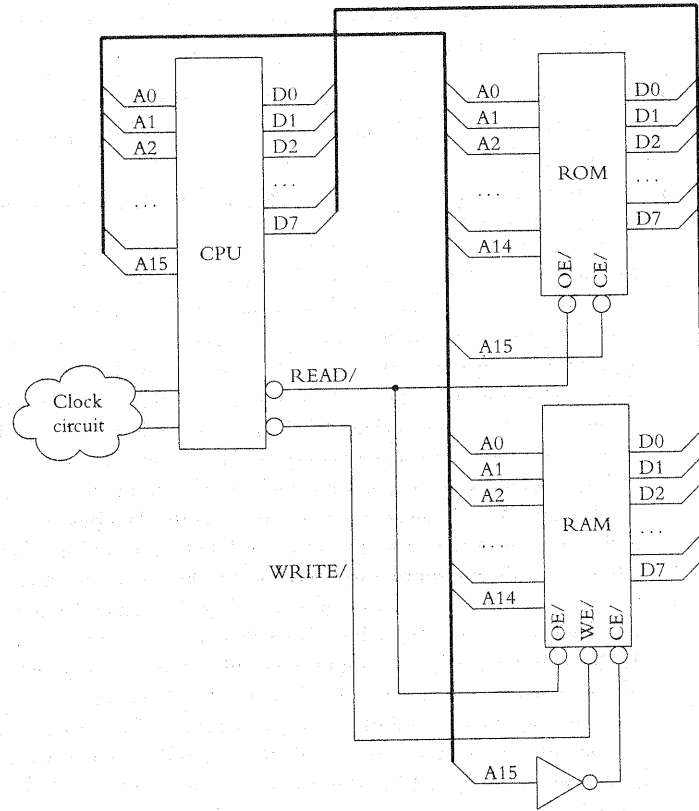
As you might expect, the address and data signals on the microprocessor are connected to the address and data signals on the ROM and the RAM. The READ/ signal from the microprocessor is connected to the output enable (OE/) signals on the memory chips. The write signal for the microprocessor is connected to the write enable (WE/) signal on the RAM. Some kind of clock circuit is attached to the clock signals on the microprocessor.

The address signals as a group are very often referred to as the **address bus**. Similarly, the data signals are often referred to as the **data bus**. The combination of the two, plus perhaps the READ and WRITE signals from the processor, are referred to as the **microprocessor bus**, or as the **bus**. The schematic in Figure 3.2 follows a common convention, in which all of the signals that are part of a bus are drawn as a single, heavy line rather than as a collection of the 8 or 16 (or 32) lines. Individual signals from the bus branch off of the heavy line and are labeled wherever they connect to some part in the circuit.

How does this circuit deal with the fact that the microprocessor might want to read either from RAM or from ROM? From the microprocessor's point of view, there are no ROM and RAM chips. It just has a 64 K address space, and when it drives address signals on the address bus to represent one of the addresses in this address space, it expects circuitry out there somewhere to provide data signals on the data bus. To make sure that the microprocessor can read from either, you must divide up the address space, assigning some of it to the ROM and some to the RAM. Then you must build circuitry to implement your division. Since the ROM and the RAM each have 32 K, one possible division is shown in Table 3.1.

You can do the arithmetic and see that both of the ranges in Table 3.1 are 32 K. To use these address ranges, you must build a circuit that activates the

Figure 3.2 A Very Basic Microprocessor System



ROM chip when an address in its range appears on the bus and that activates the RAM chip when an address in its range appears on the bus. In this particular case this is simple. Notice that in all of the addresses that correspond to ROM, the highest-order address signal (A15) is 0, whereas in all of the addresses that correspond to RAM, A15 is 1. Therefore, you can use the A15 signal to decide which of the two chips—ROM or RAM—should be activated. In Figure 3.2 you can see that A15 is attached to the chip enable (CE/) signal on the ROM,

Table 3.1 A Possible Division of the Address Space

	Low Address	High Address
ROM	0x0000 binary: 0000000000000000	0x7fff binary: 0111111111111111
RAM	0x8000 binary: 1000000000000000	0xffff binary: 1111111111111111

enabling it whenever A15 is 0. The A15 signal is inverted and then attached to the chip enable signal on the RAM, enabling the RAM whenever A15 is 1.

As an example, consider what happens if the microprocessor tries to read from address 0x9123. The A15 signal will be a 1 (because 0x9123 is 1001000100100011 in binary), which means that the chip enable signal on the ROM will be high, and the ROM will therefore be disabled. But because the A15 signal is high, the output of the inverter at the bottom of Figure 3.2 will be low, enabling the RAM. The RAM will place the data from its cell number 0x1123 on the bus, 0x1123 (not 0x9123) because the A15 signal is not part of the address bus that goes to the RAM; the RAM sees only 001000100100011. See Figure 3.3.

Figure 3.3 Another Look at the Address Space

Microprocessor addresses	0xffff	0x7fff	RAM addresses
	0x8000	0x0000	ROM addresses
	0x7fff	0x7fff	
	0x0000	0x0000	

Additional Devices on the Bus

In addition to the microprocessor, the ROM, and the RAM, most embedded systems have other hardware. For example, the underground tank monitoring system must have hardware to capture the float levels; the cordless bar-code scanner must have some device to send bytes out on the radio. Some of these devices must be connected to the microprocessor, because the microprocessor needs to be able to read data from them or write data to them. Almost invariably, the microprocessor and these devices are connected using the same bus that is used to connect the microprocessor and the memory; the address and data signals that make up the bus connect to the additional devices as well.

One common way to make this work is to assign each of these devices an address range within the address space that is not used by any of the memory parts. For example, if the microprocessor can address a megabyte (addresses from 0x00000 to 0xffff) and the ROM and RAM between them take up half a megabyte (addresses from 0x00000 to 0x7fff, perhaps), then a network chip might be assigned addresses from 0x80000 to 0x800ff. The size of the address range that the network chip needs depends upon that chip. The hardware engineer builds a circuit that asserts the chip enable signal on the network chip when one of the addresses in the range appears on the address bus.

This scheme is known as **memory mapping**, since the additional devices will look like more memory to the microprocessor. The following C code fragment is a sample of code to use a memory-mapped device.

```
#define NETWORK_CHIP_STATUS ((BYTE *) 0x80000)
:
:
void vFunction ()
{
    BYTE byStatus;
    BYTE *p_byHardware;
    :
    :
    /* Set up a pointer to the network chip. */
    p_byHardware = NETWORK_CHIP_STATUS;

    /* Read the status from the network chip. */
    byStatus = *p_byHardware;
    :
    :
}
```

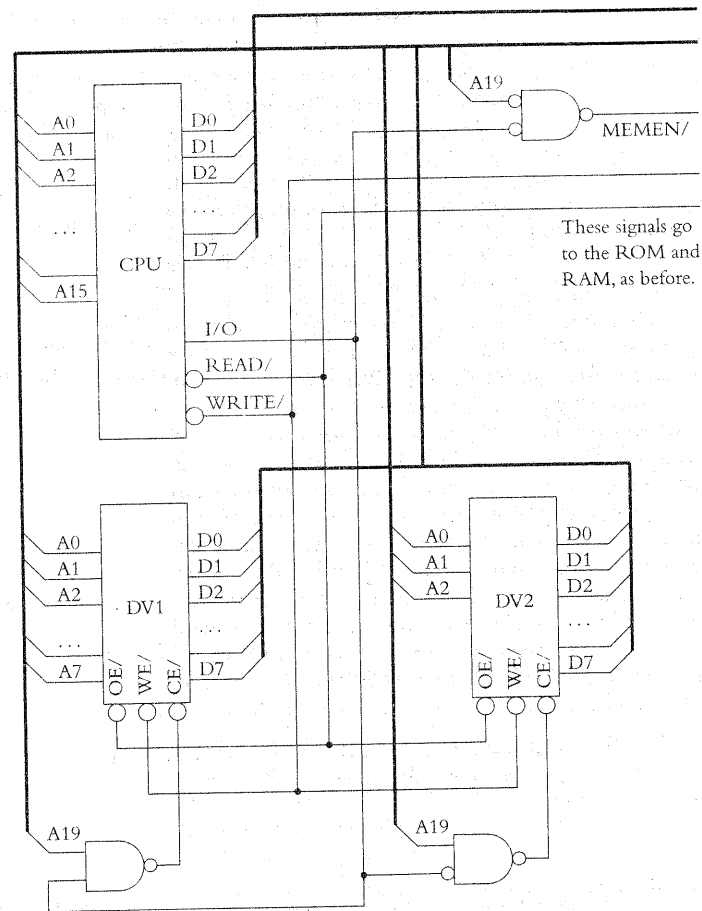
Some microprocessors allow an alternative mechanism because they support two address spaces: the **memory address space**, which we have already discussed, and an **I/O address space**. A microprocessor that supports an I/O address space has one or two additional pins with which it signals when it is reading or writing in the memory address space or in the I/O address space. Different microprocessors signal this in different ways; perhaps the most common is a single pin that the microprocessor drives low for the memory address space and high for the I/O address space.

Microprocessors that support an I/O address space have extra assembly language instructions for doing that. The MOVE instruction reads from or writes to memory; instructions such as “IN” and “OUT” access devices in the I/O address space. The libraries of the C compilers for these microprocessors typically contain functions to read to and write from devices in the I/O address space, with names such as `inport`, `outport`, `inp`, `outp`, `inbyte`, `inword`, `inpw`, and so on. The code fragment shown here illustrates typical use of these functions.

```
#define NETWORK_CHIP_STATUS (0x80000)
#define NETWORK_CHIP_CONTROL (0x80001)
:
:
void vFunction ()
{
    BYTE byStatus;
    :
    :
    /* Read the status from the network chip. */
    byStatus = inp (NETWORK_CHIP_STATUS);
    :
    :
    /* Write a control byte to the network chip. */
    outp (NETWORK_CHIP_CONTROL, 0x23);
    :
    :
}
```

Figure 3.4 is an example of a system with one device in the I/O address space (DV1) and another device in the memory address space (DV2). The hypothetical microprocessor in the system sets the I/O signal high to read or write in the I/O address space and low to read or write in the memory address space. The gate in the upper right-hand corner of the schematic that drives the memory enable signal (MEMEN/) asserts that signal low when the I/O signal and A19 are both low. This enables the memory chips in the memory address space in the range

Figure 3.4 Memory Mapping and the I/O Address Space



from 0x00000 to 0x7fff. The gate below DV1 asserts the chip enable signal to DV1 when A19 and I/O are both high. Since DV1 has eight address signals, it appears in the I/O address space in the range from 0x80000 to 0x800ff. The gate below DV2 asserts the chip enable signal to DV2 when A19 is high and I/O

is low. Since DV2 has three address signals, it appears in the memory address space in the range from 0x80000 to 0x80007.

(Note that since this circuit asserts DV1's chip enable signal whenever A19 and I/O are high and does not check A18 through A8, the circuit can read from or write to DV1 no matter what the values of those address signals. Effectively, whatever the circuit reads from 0x80000, it can also read from 0x80100, 0x80200, 0x8fe00, and so on. Similarly, the same data appears at multiple addresses in device DV2. This sort of thing is fairly common in embedded systems.)

Bus Handshaking

In addition to the logic problems of hooking up the address and data busses correctly in Figure 3.2, another issue that must be resolved is the problem of **timing**. As we discussed in the last chapter, the ROM and the RAM will have various timing requirements: the address lines must stay stable for a certain period of time, and the read enable and chip enable lines must be asserted for some period of time; only then will the data be valid on the bus. The microprocessor is in control of all of these signals, and it decides when to look for data on the bus. This entire process is called a **bus cycle**. For the circuit to work, the signals that the microprocessor produces must conform to the requirements of the other parts in the circuit. The various mechanisms by which this can be accomplished are referred to collectively as **bus handshaking**. Several of these mechanisms are discussed below. One of them requires the active cooperation of the software.

No Handshake

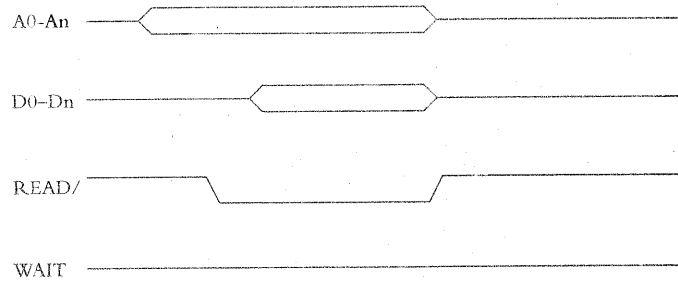
If there is no bus handshaking, then the microprocessor just drives the signals at whatever speed suits it, and it is up to the other parts of the circuit to keep up. In this scenario, the hardware engineer must select parts for the circuit that can keep up with the microprocessor (or, conversely, buy a microprocessor that is slow enough that it won't get ahead of the other parts). As we discussed in Chapter 2, you can buy ROMs and RAMs that run at various speeds. For example, you can purchase ROMs that respond in 120, 90, or 70 nanoseconds, depending on how fast they must be to keep up with your microprocessor (and on how much you're willing to pay).

Wait Signals

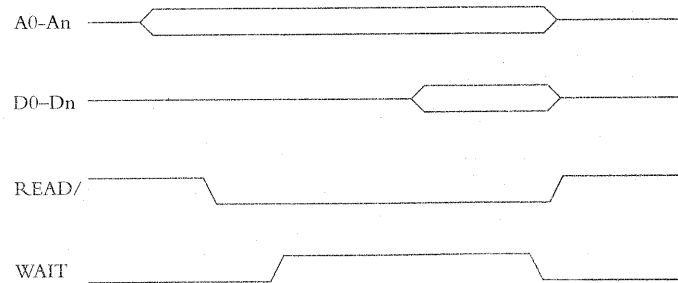
Some microprocessors offer a second alternative; they have a WAIT input signal that the memory can use to extend the bus cycle as needed. This is illustrated in Figure 3.5. In the top half of the figure is the microprocessor's "normal" bus cycle. If a device cannot respond as quickly as that diagram requires, however,

Figure 3.5 Bus Handshaking with a Wait Signal

Normal bus cycle



Bus cycle extended by asserting the WAIT signal



The device can assert the WAIT signal as long as it needs to, and the microprocessor will wait.

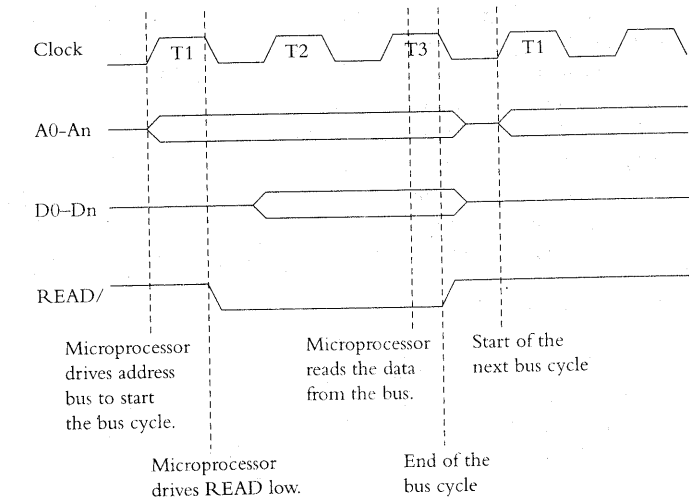
it can assert the WAIT signal to make the microprocessor extend the bus cycle. As long as the WAIT signal is asserted, the microprocessor will wait indefinitely for the device to put the data on the bus. This is illustrated in the lower half of the figure.

The only disadvantage of using a WAIT signal is that ROMs and RAMs don't come from the manufacturer with a wait signal, so someone has to build the circuitry to drive the wait signal correctly, and this can take up engineering time to design and cost money to build.

Wait States (and Performance)

Some microprocessors offer a third alternative for dealing with slower memory devices—wait states. To understand wait states, you need first to understand how the microprocessor times the signals on the bus in the first place. The microprocessor has a clock input, as we've mentioned, and it uses this clock to time all of its activities, in particular its interaction with the bus. Examine Figure 3.6.

Figure 3.6 The Microprocessor Clock Times the Bus



Each of the signal changes during the bus cycle happens at a certain time in relation to the microprocessor's input clock signal. The clock cycles in a single bus cycle are typically labeled T1, T2, T3, etc. The microprocessor shown in this figure behaves as follows (This is essentially the timing of a Zilog Z80.):

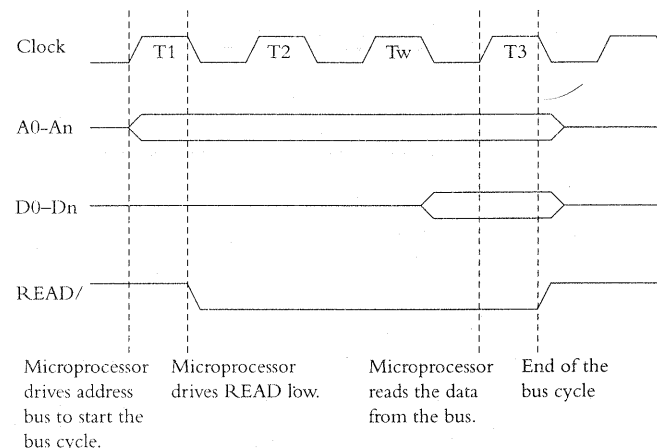
- It outputs the address on the rising edge of T1, that is, when the clock signal transitions from low to high in the first clock cycle of the bus cycle.
- It asserts the READ/ line at the falling edge of T1.
- It expects the data to be valid and actually takes the data in just a little after the rising edge of T3 (shown by the third vertical line in the figure).
- It de-asserts the READ/ line at the falling edge of T3 and shortly thereafter stops driving the address signals, thereby completing the transaction. The next clock cycle would be T1 of the following bus cycle, and if the microprocessor is ready, it will drive another address onto the address bus to start another bus cycle.

If this microprocessor is capable of using wait states, then it will be able to insert extra clock cycles, typically between cycles T2 and T3. See Figure 3.7. The beginning of the bus cycle is the same as before, with the microprocessor driving the address signals and the READ/ signal at the start of the cycle. However, the microprocessor then waits one extra bus cycle before reading the data and completing the cycle. A piece of circuitry inside the microprocessor called a **wait state generator** is responsible for this behavior.

Most wait state generators allow software to tell them how many wait states to insert into each bus cycle, up to some maximum, perhaps three or perhaps fifteen. Most microprocessors also allow you to use different numbers of wait states for different parts of the address space. This latter is useful because some devices are much faster than others: RAM, for example, is typically faster than ROM; I/O devices tend to be slow.

The typical microprocessor inserts the maximum number of wait states into every bus cycle when it is first powered up or is reset. This means that the hardware engineer can use a slow ROM if he or she wants to save some money. It also means that the processor will start off very slowly, even if the hardware engineer decides to pay for a fast ROM. It is obvious that the fewer wait states that your system is using the faster it will run. It is up to software engineers to find out from the hardware engineers how few wait states they can get away with, and then write code to set up the wait state generator accordingly.

Figure 3.7 The Microprocessor Adds a Wait State



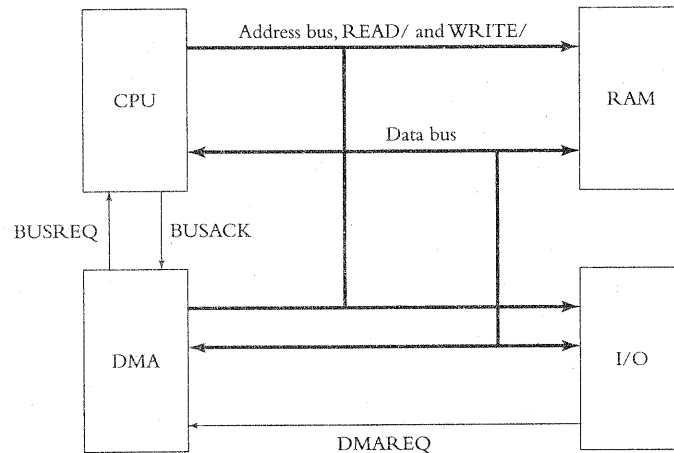
3.3 Direct Memory Access

One way to get data into and out of systems quickly is to use **direct memory access** or **DMA**. DMA is circuitry that can read data from an I/O device, such as a serial port or a network, and then write it into memory or read from memory and write to an I/O device, all without software assistance and the associated overhead. However, DMA creates some new problems for hardware designers to resolve. The first difficulty is that the memory only has one set of address and data signals, and DMA must make sure that it is not trying to drive those signals at the same time as the microprocessor is trying to drive them.

This is usually solved in a manner similar to that shown in Figure 3.8. In all of the discussion that follows, we will discuss transferring data from the I/O device to the RAM; the process for moving data in the other direction is similar.

When the I/O device has data to be moved into the RAM, it asserts the DMAREQ signal to the DMA circuit. The DMA circuit in turn asserts the BUSREQ signal to the microprocessor. When the microprocessor is ready to give up the bus—which may mean not executing instructions for the short

Figure 3.10 Alternative DMA Architecture



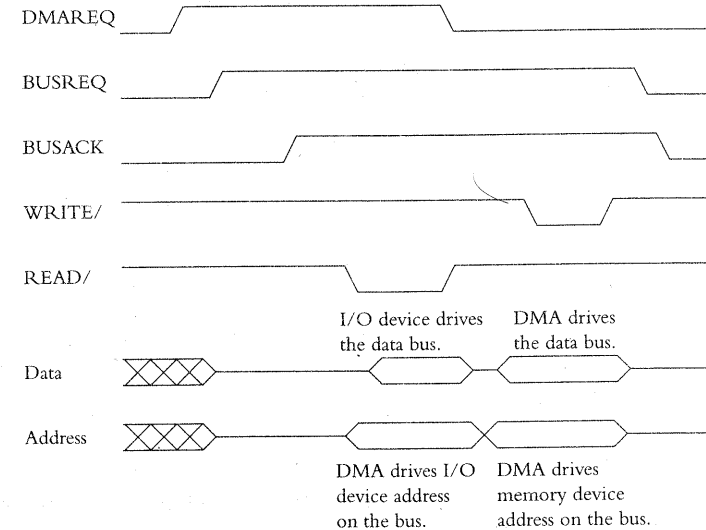
A timing diagram for this is shown in Figure 3.11.

The advantage of this second architecture over the one shown in Figure 3.8 is that it puts less burden on the I/O device circuitry. The I/O device needs only to be able to assert DMAREQ at appropriate times, because the fulfillment of the request looks like a regular read from the perspective of the I/O device. On the other hand

- The DMA circuit is quite a bit more complicated in that it has to be able to store the data.
- It takes about twice as much bus time to transfer the data, since it has to be transferred first to the DMA and then on to the memory.

If several I/O devices need to use DMA simultaneously to move data, your system will need a copy of the DMA circuitry, called a **DMA channel**, for each one. Some I/O devices come with DMA channels built into them. I/O devices that can move large amounts of data quickly, such as network controllers, are particularly likely to have a DMA channel built in.

Figure 3.11 Alternate DMA Timing

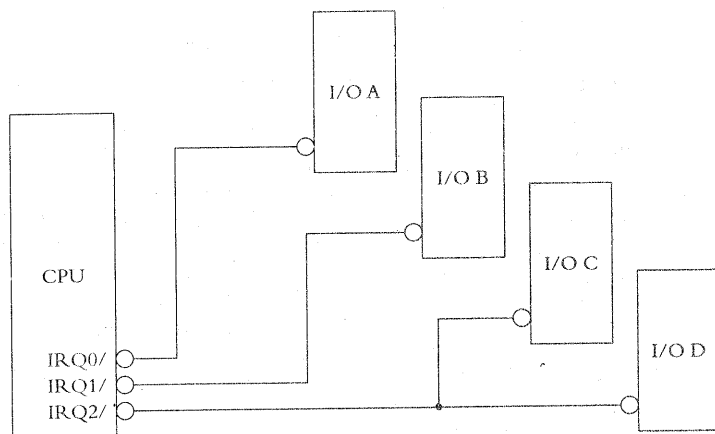


3.4 Interrupts

As you probably know, the microprocessor can be **interrupted**, that is, told to stop doing what it is doing and execute some other piece of software, the **interrupt routine**. The signal that tells the microprocessor that it is time to run the interrupt routine is the **interrupt request** or **IRQ**. Most microprocessors have several external interrupt request input pins on them. The hardware designer can connect them to the interrupt request output pins typically provided on I/O parts to allow those parts to interrupt the processor.

It is typical for the interrupt request signals to be asserted low, and it is typical for the interrupt request pins on I/O devices to be open collectors, so that several of them can share an interrupt request pin on the microprocessor. See the schematic in Figure 3.12. I/O Device A can interrupt the processor by asserting the signal attached to IRQ0/; I/O Device B can interrupt the processor by asserting IRQ1/; I/O Devices C and D can interrupt the processor by asserting IRQ2/.

Figure 3.12 Interrupt Connections



Like DMA channels responding to a DMAREQ signal, the microprocessor's response to the interrupt inputs can be edge triggered or level triggered.

3.5 Other Common Parts

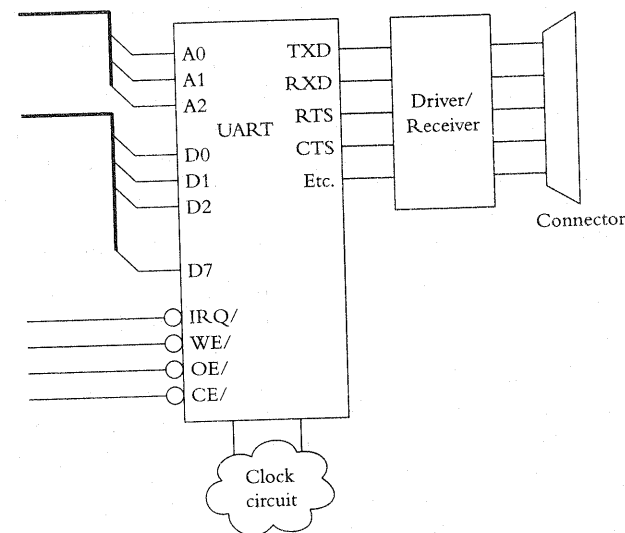
In this section we'll discuss other parts found on many systems.

Universal Asynchronous Receiver/Transmitter and RS-232

A Universal Asynchronous Receiver/Transmitter or UART is a common device on many systems. Its purpose is to convert data to and from a **serial interface**, that is, an interface on which the bits that make up the data are sent one after another. A very common standard for serial interfaces is the RS-232 interface, used between computers and modems and nowadays often between computers and mice.

A typical UART and its connections are shown in Figure 3.13. On the left-hand side of the UART are those signals that attach to the bus structures we discussed back in Section 3.1: address lines, data lines, read and write lines, and an interrupt line. From the perspective of the microprocessor, the UART looks

Figure 3.13 A System with a UART



very much like some more memory in that when the microprocessor wishes to send data to or receive data from the UART, it puts out the same sequences of signals on the bus (unless the UART is in the I/O address space). As with a ROM or a RAM, external circuitry must figure out when to drive the chip enable signal on the UART.

At the bottom of the UART is a connection into a clock circuit. The clock circuit for the UART is often separate from the microprocessor's clock circuit, because it must run at a frequency that is a multiple of the common bit rates. UART clock circuits typically run at odd rates such as 14.7456 megahertz, simply because 14,745,600 is an even multiple of 28,800, and 28,800 bits per second is a common speed for communications. There is no similar restriction on the clock that drives the microprocessor.

The signals on the right are the ones that go to the serial port: a line for transmitting bits one after another (TXD), a line for receiving bits (RXD), and some standard control lines used in the RS-232 serial protocol (request-to-send, RTS; clear-to-send, CTS; etc.). The lines are connected to an RS-232 driver/receiver part. The UART usually runs at the standard 3 or 5 volts of the

rest of the circuit, but the RS-232 standard specifies that a 0 be represented by +12 volts and a 1 by -12 volts. The driver/receiver part is responsible for taking the UART output signals and converting them from 0 volts and 5 volts to +12 and -12 volts; and for converting the input signals from the connector from +12 and -12 volts to 0 volts and 5 volts.

A typical UART, in common with many other I/O devices, has a handful of internal locations for data, usually called **registers**, to which the microprocessor can write to control behavior of the UART and to send it data to be transmitted and from which the microprocessor can read to retrieve data that the UART has received. Each register is at a different address within the UART. The typical registers you might find in a UART include the following:

- A register into which the microprocessor writes bytes to be transmitted. (The microprocessor writes the data a byte at a time, and the UART will transmit them a bit at a time.)
- A register from which the microprocessor reads received bytes. Note that this might be at the same address within the UART as the previous register, since the manufacturer of the UART can reasonably assume that you will only read from this register and only write to the other. Note that it is often the case that you cannot read back data that you have written into registers in UARTs and other devices, whether or not the manufacturer has used the same address for another register.
- A register with a collection of bits that indicate any error conditions on received characters (bad parity, bad framing, etc.)
- A register the microprocessor writes to tell the UART when to interrupt. Individual bits in that register might indicate that the UART should interrupt when it has received a data byte, when it has sent a byte, when the **clear-to-send** signal has changed on the port, etc.
- A register the microprocessor can read to find out why the UART interrupted. Note that reading or writing this or other registers in the UART often has side effects, such as clearing the interrupt request and causing the UART to stop asserting its interrupt signal.
- A register the microprocessor can write to control the values of request-to-send and other outgoing signals.
- A register the microprocessor can read to find out the values of the incoming signals.
- One or more registers the microprocessor writes to indicate the data rate. Typically, UARTs can divide their clocks by whatever number you specify. You specify the number by writing it into some registers in the UART.

Your program controls the UART by reading from and writing to these registers at appropriate moments.

UARTs come with all sorts of bells and whistles, of which the following are just examples:

- On very simple ones, you must write one byte and wait for that byte to be transmitted before writing the next; more complex UARTs contain a **First-In-First-Out** buffer, or FIFO, that allows your software to get several bytes ahead. The UART will store the bytes and eventually catch up.
- Similarly, more complex UARTs contain FIFOs for data that is being received, relieving your software of the requirement to read one byte before the next one arrives.
- Some UARTs will automatically stop sending data if the clear-to-send signal is not asserted.
- Some UARTs have built-in DMA or at least the logic to cooperate with a DMA channel.

Programmable Array Logic

Most systems require a certain amount of **glue** circuitry in addition to the microprocessor, the ROM, the RAM, and the other major parts. Glue circuitry connects outputs that assert high to inputs that assert low, drives chip-enable signals appropriately based on the address signals, and so on. In the past, this glue was often constructed out of individual AND, NAND, and OR gates and inverters. However, circuits with fewer parts are generally cheaper to build and more reliable, so engineers nowadays try to avoid large collections of these simple parts and use instead fewer, more complex parts.

Each system needs its own combination of glue circuitry to work, however, so each one must be designed afresh. No single chip will do the job for any arbitrary system. This problem has led to a class of parts called **Programmable Logic Devices** or PLDs. These devices allow you to build more or less any small glue circuit you want, even if what you want includes three-input NAND gates in which two of the inputs are inverted.

The smallest of the PLDs have 10 to 20 pins and an array of gates that you can hook up after you buy them; these parts are called **Programmable Array Logic** or PALs. In essence, a PAL has a rather large collection of discrete parts in it and a method by which you can rearrange the connections among these parts and between the parts and the pins. The method usually requires a piece of equipment, a **PAL programmer**, much as programming PROMs requires a

PROM programmer. (In fact, there are a number of PROM programmers that also will program some kinds of PALs.)

Let's suppose that the glue we need for a certain system is as follows:

- The ROM is at addresses 0 to 0x3fff; therefore, the glue must assert its chip enable signal when address lines 14 and 15 are both low.
- The UART is at addresses starting at 0x4000; therefore, the glue must assert its chip enable signal when address line 15 is low and address line 14 is high.
- The RAM is at addresses 0x8000 to 0xffff; therefore, the glue must assert its chip enable signal when address line 15 is high.
- The ROM and the UART are slow devices, and the processor can be made to extend its cycle with a WAIT signal. The WAIT signal needs to be asserted for two processor clock cycles whenever the ROM or the UART is used.

If we build this system with a PAL, the schematic might look something like the one in Figure 3.14. The data and address busses and the READ/ and WRITE/ lines are hooked up as we discussed earlier, but the PAL takes in A14,

Figure 3.14 A Circuit with a PAL In It

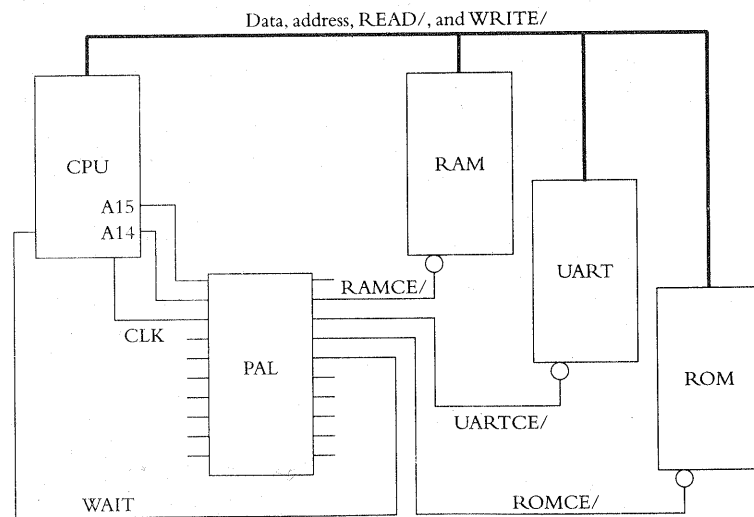


Figure 3.15 PAL Code

```

Declarations
  AddrDecode DEVICE 'P22V10'

  "INPUTS"
    A15 PIN 1
    A14 PIN 2
    iClk PIN 3

  "OUTPUTS"
    !RamCe PIN 19
    !UartCe PIN 18
    !RomCe PIN 17
    Wait PIN 16
    Wait2 PIN 15

Equations
  RamCe = A15
  RomCe = !A15 * !A14
  UartCe = !A15 * A14

  Wait.CLK = iClk
  Wait2.CLK = iClk

  Wait := (RomCe + UartCe) * !Wait2
  Wait2 := Wait * !Wait2

end AddrDecode

```

A15, and the processor clock and generates the various chip enables and the WAIT signal back to the processor.

Obviously, it's a little difficult to determine how the circuit in Figure 3.14 works without knowing something about how the PAL works. To know how the PAL works, you need to know the PAL equations, which describe what the PAL does. The PAL equations can be written in any of several languages that have been created for the purpose. An example is in Figure 3.15.

This PAL code starts by declaring that we will build a device named AddrDecode, which will be created in a P22V10 (one of the standard PALs you can buy).

The sections on “INPUTS” and “OUTPUTS” assign names to each of the pins that we will use. An exclamation point on the pin declaration indicates that the signal on that pin is asserted low. Subsequently, whenever the name that corresponds to the pin is set to 1, the pin itself will go low.

The Equations section tells how the outputs depend upon the various inputs. The first three equations determine how the PAL will drive the chip enable signals. For example:

$$\text{RamCe} = \text{A15}$$

will assert the RamCe signal whenever A15 is high. Since A15 is high for addresses 0x8000 to 0xffff, the RAM chip enable will be asserted whenever the microprocessor puts out an address in the RAM's range. Note that because the pin declaration for the RamCe signal indicates that it is asserted when it is low, pin 19 of the PAL will go low and select the RAM whenever the microprocessor selects an address in the range from 0x8000 to 0xffff.

Similarly, the equation for RomCe asserts that signal (low) whenever A14 and A15 are both low, that is, when the address is between 0x0000 and 0x3fff, and the equation for UartCe asserts that signal in the address range from 0x4000 to 0x7fff. In this language, the asterisk represents a logical AND and the plus sign represents OR.

The equations for Wait and Wait2 are a little different from those for the chip enable lines. The equations for the chip enable lines are **combinatorial**. They depend only upon the levels of the signals on the right-hand sides of the equation, and the output signals named on the left-hand sides of the equations change as soon as the input signals on the right-hand side change. The equations for Wait and Wait2 are **clocked**. The equations are only evaluated by the PAL—and the Wait and Wait2 outputs are only changed—on the edge of the given clock signal. This is behavior similar to that of the D flip-flop discussed in Chapter 2. The difference between the two types of equations is shown in this PAL equation language by the use of the colon (:) in front of the equals sign.

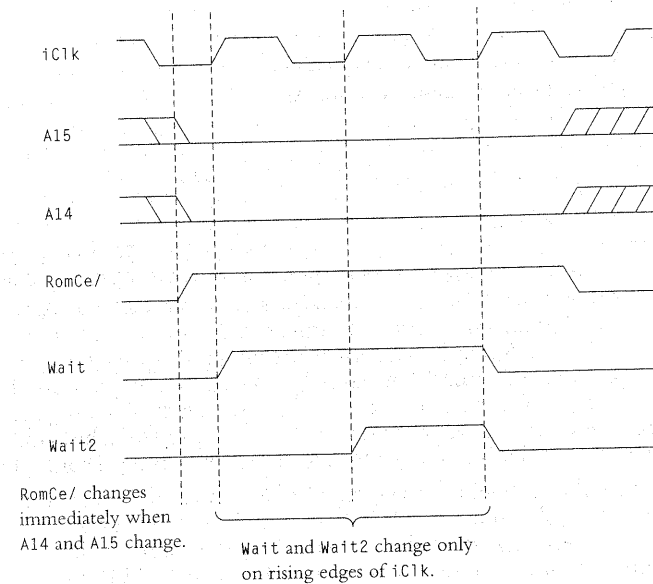
Because of these two lines among the equations

$$\begin{aligned}\text{Wait.CLK} &= \text{iClk} \\ \text{Wait2.CLK} &= \text{iClk}\end{aligned}$$

the clock that causes the Wait and Wait2 equations to be evaluated is the rising edge of the iClk signal. Wait and Wait2 will be low at first. See how these equations work:

$$\begin{aligned}\text{Wait} &:= (\text{RomCe} + \text{UartCe}) * \text{!Wait2} \\ \text{Wait2} &:= \text{Wait} * \text{!Wait2}\end{aligned}$$

Figure 3.16 PAL Timing



On the first rising edge of iClk after RomCe or UartCe is asserted, Wait will be asserted, but the equation for Wait2 will evaluate to FALSE, because it will use the old value of the Wait signal in its calculation. On the second rising edge of iClk, Wait will remain asserted (because none of the signals on the right-hand side of the equation will have changed), but now Wait2 will go high. On the third rising edge of iClk, Wait and Wait2 will both go low.

Figure 3.16 shows the timing diagram for this PAL. Note how RomCe/ reacts immediately to A14 and A15. Note how Wait and Wait2 react only when iClk rises.

Most PAL languages have a few other features:

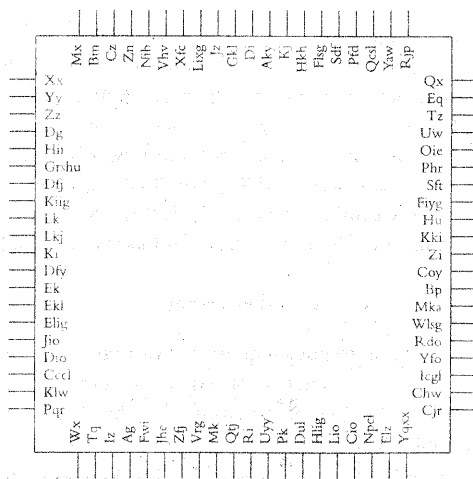
- They allow the programmer to put a sequence of **test vectors** into the program, a sequence of inputs and a sequence of expected outputs. The device that programs the PAL uses these vectors to ensure that the PAL operates correctly after it is programmed.
- They have mechanisms to allow the engineer to build state machines easily.

Application-Specific Integrated Circuits and Field-Programmable Gate Arrays

Two other kinds of parts you are likely to find on modern circuits are **Application-Specific Integrated Circuits** (or ASICs, pronounced “ay’-sicks”) and **Field-Programmable Gate Arrays** (or FPGAs). These parts are increasingly popular because they are an economical way to create custom, complex hardware on a circuit without adding a lot of parts.

An ASIC is an integrated circuit built specially to go into the circuit for which it is designed. In theory, an ASIC can contain whatever the hardware engineer wants, but in practice many ASICs consist of a core of some kind, typically a microprocessor, plus perhaps some modest peripherals and all of the glue necessary to hold the circuit together. On the schematic, an ASIC is very often shown as a symbol such as that in Figure 3.17, which tells you nothing about what the ASIC does. Therefore, if the circuit you are working with contains one or more ASICs, you must get some description of what the ASICs do. Fortunately, since it is extremely expensive to get an ASIC into production and extremely expensive to change it if there’s a bug in it, most hardware engineers document their ASICs carefully before they build them.

Figure 3.17 An ASIC



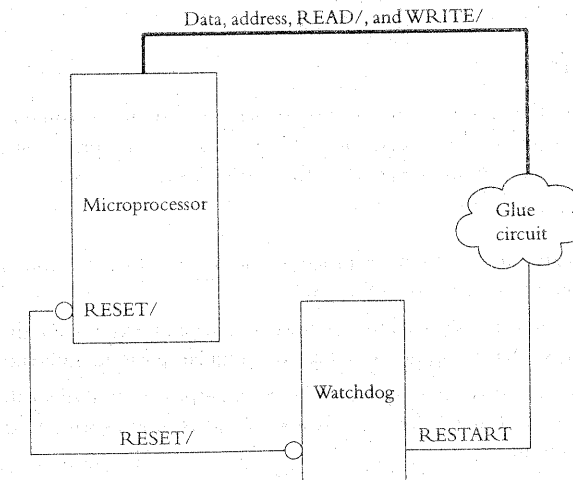
An FPGA is like a large PAL, in that it has a large number of gates and the connections among them can be programmed after the part has been manufactured. Some of these parts are programmed in a special programming device; others can be programmed by the microprocessor even after the product has been shipped into the field. In some systems, the software must program the FPGAs every time the system starts up.

Watchdog Timers

One very common part on many embedded-system circuits is a **watchdog timer**. A watchdog timer contains a timer that expires after a certain interval unless it is restarted. The watchdog timer has an output that pulses should the timer ever expire, but the idea is that the timer will never expire. Some mechanism allows software to restart the timer whenever it wishes, forcing the timer to start timing its interval over again. If the timer is ever allowed to expire, the presumption is that the software failed to restart it often enough because the software has crashed.

The way that watchdog timers are connected into circuits is shown in Figure 3.18. The output of the watchdog timer is attached to the RESET/

Figure 3.18 Typical Use of a Watchdog Timer



signal on the microprocessor; if the timer expires, the pulse on its output signal resets the microprocessor and starts the software over from the beginning. Different watchdog circuits require different patterns of signals on their inputs to restart them; typical is to require any edge on the RESTART signal. Some glue circuitry may be necessary to allow the microprocessor to change the RESTART signal appropriately.

3.6 Built-Ins on the Microprocessor

Microprocessors, particularly those marketed for embedded systems, very often come with a number of auxiliary circuits built into them. In this section we'll discuss some of them. These auxiliary circuits are usually logically separate from the microprocessor—they're just built on the same piece of silicon and then wired directly to the microprocessor. The advantage of these built-ins is that you get the auxiliary circuits in your system without having to add extra parts.

Each auxiliary circuit, or **peripheral**, is controlled by writing values to a small collection of registers that typically appear at some fixed locations in the microprocessor's address space. The peripherals usually can interrupt the microprocessor, just as if they were completely separate from it; there is some mechanism that coordinates the interrupts from the on-board circuitry and the interrupts coming from outside the microprocessor.

Timers

It is common for microprocessors to have one or more **timers**. A timer is essentially just a counter that counts the number of microprocessor clock cycles and then causes an interrupt when the count expires. Here are a few features of the usual microprocessor timers:

- A **pre-scaler** divides the microprocessor clock signal by some constant, perhaps 20, before the signal gets to the timer.
- The counter can reset itself to its initial value when it expires and then continue to count, so that it can be the source of a regular, periodic interrupt.
- The timer can drive an output pin on the microprocessor, either causing a pulse whenever the timer expires or creating a square wave with an edge at every timer expiration.

- The timer has an input pin that enables or disables counting. The timer circuit also may be able to function as a counter that counts pulses on that input pin.

Most timers are set up by writing values into a small collection of registers, typically registers to hold the count and a register with a collection of bits to enable the counter, to reset the interrupt, to control what the timer does to its output pin, if any, and so on.

DMA

It is not unusual to find a few DMA channels built into a microprocessor chip. Since a DMA channel and the microprocessor contend for the bus, certain processes are simplified if the DMA channel and the microprocessor are on the same chip.

(If your microprocessor supports some kind of memory mapping, note that the DMA circuitry will most likely bypass it. DMA circuits operate exclusively on the physical memory addresses seen outside of the microprocessor chip.)

I/O pins

It is common for microprocessors intended for embedded systems to contain anywhere from a few to a few dozen I/O pins. These pins can be configured as outputs that software can set high or low directly, usually by writing to a register, or they can be configured as inputs that software can read, again usually by reading from a register. These pins can be used for any number of purposes, including the following:

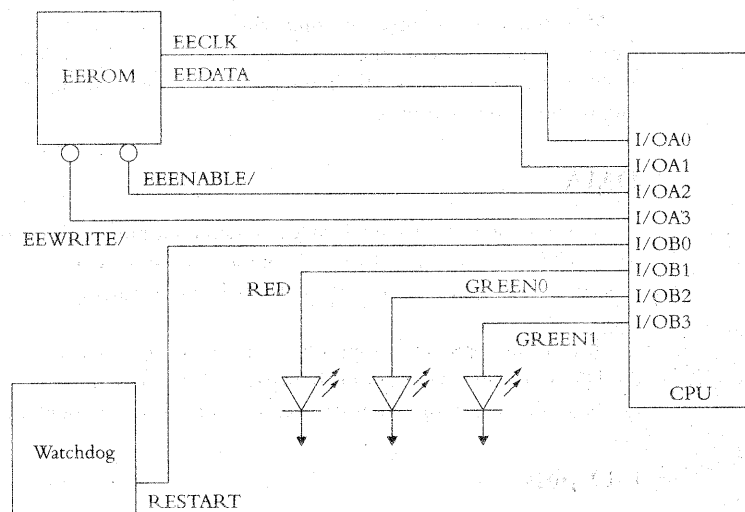
- Turning LEDs on or off
- Resetting a watchdog timer
- Reading from a one-pin or two-pin EEPROM
- Switching from one bank of RAM to another if there is more RAM than the processor can address

Figure 3.19 shows some of these common uses.

Address Decoding

As we have seen in some of our earlier discussions, using an address to generate chip enables for the RAM, ROM, and various peripheral chips can be a nuisance. Some microprocessors offer to do some of that **address decoding**

Figure 3.19 Uses for I/O Pins



for you by having a handful of chip enable output pins that can be connected directly to the other chips. Typically, the software has to tell the microprocessor the address ranges that should assert the various chip enable outputs. Often, you can program the microprocessor to use different numbers of wait states, depending upon which chip enable pin is asserted.

Memory Caches and Instruction Pipelines

A number of microprocessors, particularly faster RISC (Reduced Instruction Set Computer) systems, contain a **memory cache** or cache on the same chip with the microprocessor. These are small, but extremely fast memories that the microprocessor uses to speed up its work. The microprocessor endeavors to keep in its cache the data and instructions it is about to need; the microprocessor can fetch items that happen to be in the cache when they are needed much more quickly than it can fetch items from separate memory chips. For the most part, you can ignore the memory cache when you are designing program logic. It affects you only when you must determine how quickly the program will

execute (because that depends on what is in the cache when) and when you are trying to debug your software (because the cache conceals much about what the microprocessor is doing; see Chapter 10).

An **instruction pipeline** or **pipeline** is similar to a memory cache in that the microprocessor endeavors to load into the pipeline instructions that it will need later, so that they will be ready for execution more rapidly than if they must be fetched from separate memory chips. The differences between pipelines and caches are that pipelines are typically much smaller than caches, that the logic behind them is often much simpler, and that the microprocessor uses them only for instructions, not for data.

3.7 Conventions Used on Schematics

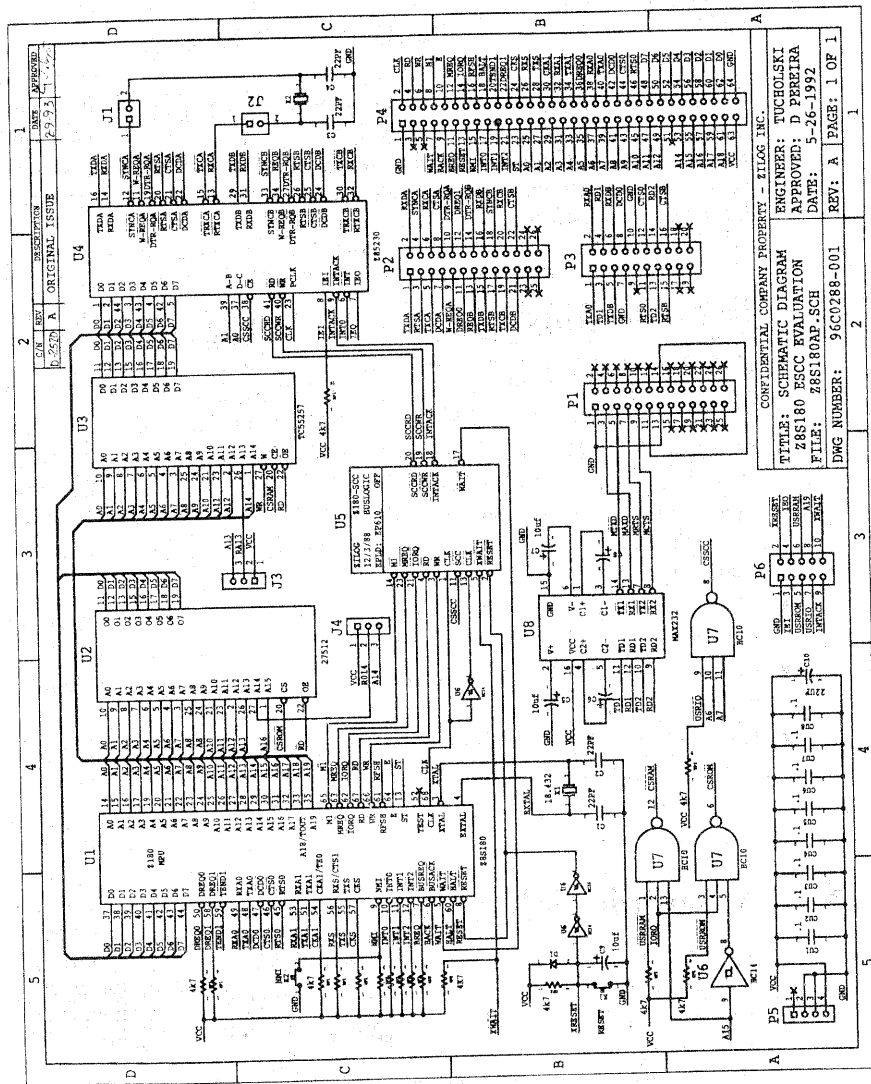
Several common conventions used on schematic diagrams have not appeared in the simple diagrams in this chapter:

- Signals are not always shown as continuous lines. Each signal is given a name; if two lines on the schematic have the same name, they are connected, even though it isn't explicitly shown. For example, if one of the address lines coming out of the microprocessor is labeled "A15," then every other line labeled A15 is that same address signal.
- The actual pin numbers on the parts that will be used in the finished circuit are shown next to each signal coming out of each part.
- Parts numbered P1, P2, P3, etc. are **connectors**, places where we can connect this circuit to external devices.
- Parts numbered J1, J2, etc. are **jumpers**, places on the circuit where a customer is expected to connect signals together or not, depending upon how he wants to use the circuit.

3.8 A Sample Schematic

Figure 3.20 is the schematic diagram for a board distributed by Zilog, Inc. to demonstrate its Z80180 microprocessor and a communication chip called the SCC, which is almost too fancy to be called a UART. A few comments about the schematic are listed below; more guidance about how this circuit works is included in the problems at the end of this chapter. With this and with the

Figure 3.20 A Sample Schematic



material we have discussed, you should be able to figure out much about how this circuit works.

Here are a few facts about the schematic in Figure 3.20 that are not obvious from the discussion we have had:

- The parts labeled P1 through P4 are indeed connectors, as you might expect. Since this is a demonstration board, practically every significant signal on it goes to a connector, just to make it easy to connect test equipment. On most circuit boards you would not see so many signals going to connectors.
- The part labeled P5 is a connector to which the user can connect a power supply.
- The parts labeled J1 through J4 are jumpers. J1 and J2 control clock options on the SCC. J3 and J4 control how certain address lines connect to memory parts. To start with you should assume that pin 2 and pin 3 on J3 have been connected to one another and that pin 2 and pin 3 on J4 have been connected to one another.
- The part labeled P6 is also a connector, but its purpose is to allow the user to configure the board further by connecting some of its pins to one another. Assume that the user has connected none of these pins to one another.
- Because of the extensive configurability of this board, many signals have pullup resistors attached to them. This forces them to be high if the user does not force them low. For example, the signal USRRAM, found on connector P6 and attached to one of the inputs on one of the NAND gates in the lower left-hand corner of the schematic, will always be high because of the pullup resistor, unless the user connects it directly to ground by connecting pin 1 to pin 6 of connector P6.
- The part labeled U5 is a programmable logic device that deals with the timing requirements of the SCC.
- The part labeled U8 is an RS-232 driver.

3.9 A Last Word about Hardware

One thing that you will notice whenever you talk to hardware engineers is that they operate with a different set of concerns than do software engineers. Here are some of those concerns: