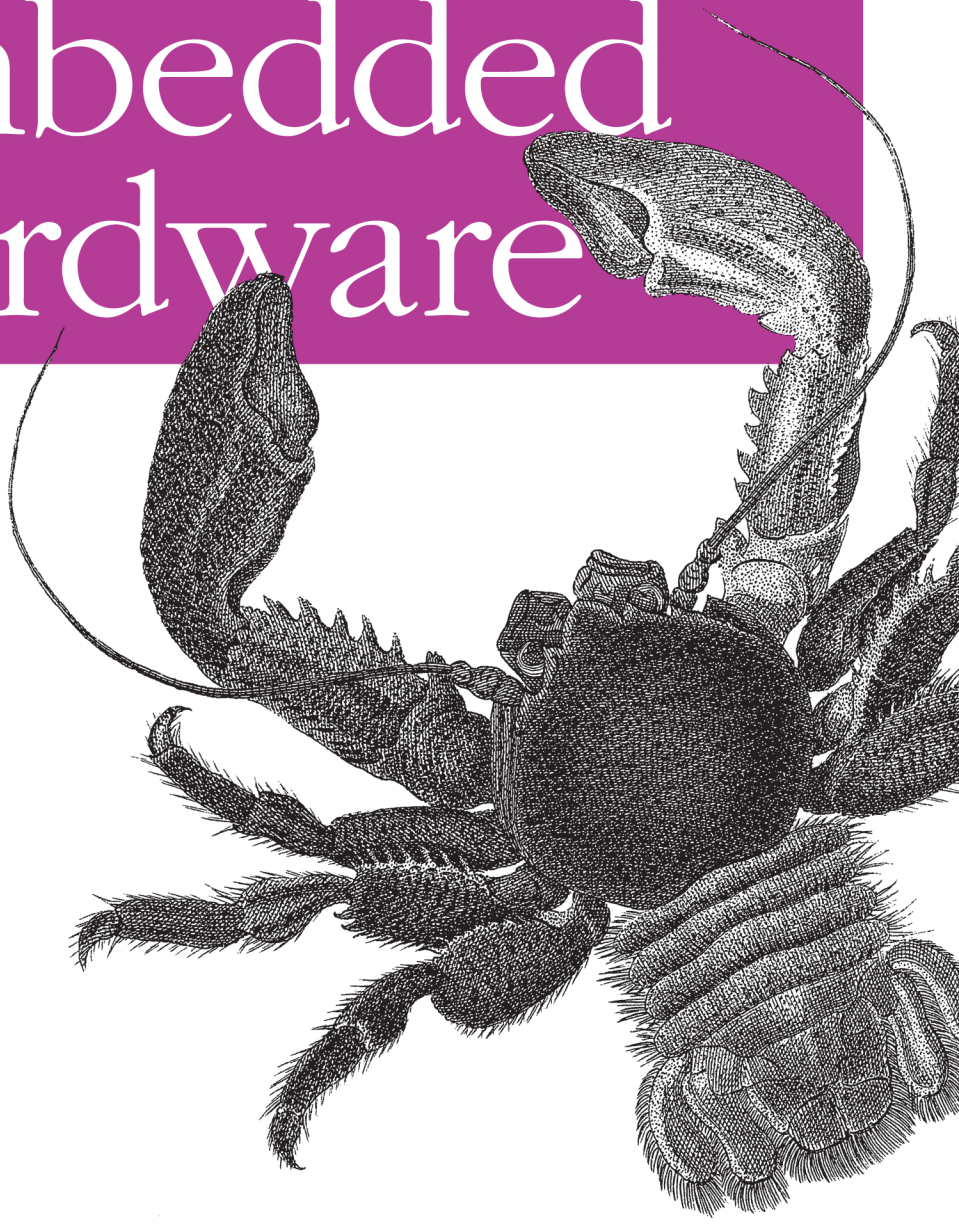
Create New Computers and Devices

2nd Edition

Designing

Embedded Hardware



O'REILLY®

John Catsoulis

INTEL EXHIBIT 1008 Page 1
IPR2023-00787

O'REILLY®

Designing Embedded Hardware



Embedded computer systems literally surround us: they're in our cell phones, PDAs, cars, TVs, refrigerators, heating systems, and more. In fact, embedded systems are one of the most rapidly growing segments of the computer industry today.

Along with the growing list of devices for which embedded computer systems are appropriate, interest is growing among programmers, hobbyists, and engineers of all types in how to design and build devices of their own. Furthermore, the knowledge offered by this book of the fundamentals of these computer systems can benefit anyone who has to evaluate and apply the systems.

The second edition of *Designing Embedded Hardware* has been updated to include information on the latest generation of processors and microcontrollers, including the new MAXQ processor. The book spells out the basics of embedded design for beginners while providing material useful for advanced systems designers.

Designing Embedded Hardware steers a course between books dedicated to writing code for particular microprocessors and those that stress the philosophy of embedded-system design without providing any practical information. Author John Catsoulis brings a wealth of real-world experience to show readers how to design and create entirely new embedded devices and computerized gadgets, as well as how to customize and extend off-the-shelf systems.

Loaded with real-world examples, this book also provides a roadmap to the pitfalls and traps to avoid. *Designing Embedded Hardware* includes:

- Theory and practice of embedded systems
- Understanding schematics and datasheets
- Powering an embedded system
- Producing and debugging an embedded system
- Processors such as the PIC, Atmel AVR, and Motorola 68000 series
- Digital Signal Processing (DSP) architectures
- Protocols (SPI and I2C) used to add peripherals
- RS-232C, RS-422, infrared communication, and USB
- CAN and Ethernet networking
- Pulse Width Monitoring and motor control

If you want to build your own embedded system, or tweak an existing one, this invaluable book gives you the knowledge and practical skills you need.

www.oreilly.com

US $49.99 **CAN $57.99**

ISBN: 978-0-596-00755-3

9 780596 007553 54999

Safari® BOOKS ONLINE ENABLED Includes FREE 45-Day Online Edition

INTEL EXHIBIT 1008 Page 2

IPR2023-00787

Designing Embedded Hardware

INTEL EXHIBIT 1008 Page 3
IPR2023-00787

Other resources from O'Reilly

Related titles

- Building Embedded Linux Systems
- Home Hacking Projects for Geeks
- Hardware Hacking Projects for Geeks
- PC Hardware Buyer's Guide
- Programming Embedded Systems in C and C++
- Car PC Hacks
- Smart Home Hacks

oreilly.com

oreilly.com is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.



oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.



Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Search across thousands of electronic books simultaneously and zero in on the information you need in seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.

SECOND EDITION

Designing Embedded Hardware

John Catsoulis

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

INTEL EXHIBIT 1008 Page 5
IPR2023-00787

Designing Embedded Hardware, Second Edition
by John Catsoulis

Copyright © 2005, 2002 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram

Production Editor: Sanders Kleinfeld

Cover Designer: Emma Colby

Interior Designer: David Futato

Printing History:

November 2002: First Edition.

May 2005: Second Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Designing Embedded Hardware*, the image of a porcelain crab, and related trade dress are trademarks of O'Reilly Media, Inc.

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 O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-00755-3
[LSI] [2011-04-08]

INTEL EXHIBIT 1008 Page 6
IPR2023-00787

This book is dedicated to my uncle,
Vince Catsoulis

INTEL EXHIBIT 1008 Page 7
IPR2023-00787

Table of Contents

Preface . . . . . **xi**

1. An Introduction to Computer Architecture **1**
Concepts 2
Memory 16
Input/Output 20
DMA 20
Embedded Computer Architecture 26

2. Assembly Language **30**
Registers 32
Machine Code 32
Signed Numbers 34
Addressing Modes 35
Coding in Assembly 37
Disassembly 40
Position-Independent Code 41
Loops 41
Masking 42
Indexed Addressing 43
Stacks 44
Timing of Instructions 45

3. Forth/Open Firmware **48**
Introducing Forth 48
String Words 51
Stack Manipulation 52
Creating New Words 54

INTEL EXHIBIT 1008 Page 9
IPR2023-00787

Comments 56
if ... else 57
Loops 58
Data Structures 61
Interacting with Hardware and Memory 62
Forth Programming Guidelines 64

4. Electronics 101 65
Voltage and Current 65
Analog Signals 67
Power 68
Reading Schematics 68
Resistors 73
Capacitors 80
RC Circuits 83
Inductors 86
Transformers 89
Diodes 90
Crystals 93
Digital Signals 98
Electrical Characteristics 99
Logic Gates 108
The Importance of Reading the Datasheet 109

5. Power Sources 110
The Stuff Out of the Wall 110
Batteries 111
Low Power Design 111
Regulators 112
LM78xx Regulators 114
MAX603/MAX604 Regulators 116
MAX1615 Regulator 117
MAX724 Regulator 118
Electrical Noise and Interference 119

6. Building Hardware 124
Tools 124
Soldering 129
Quick Construction 136

INTEL EXHIBIT 1008 Page 10
IPR2023-00787

Printed-Circuit Boards 140
Building It 153
JTAG 157

7. Adding Peripherals Using SPI 160
Serial Peripheral Interface 160

8. Adding Peripherals Using I²C 174
Overview of I²C 174
Adding a Real-Time Clock with I²C 178
Adding a Small Display with I²C 179

9. Serial Ports 180
UARTs 180
Error Detection 182
Old Faithful: RS-232C 183
RS-422 190
RS-485 192

10. IrDA 196
Introduction to IrDA 196

11. USB 203
Introduction to USB 204
USB Packets 206
Physical Interface 208
Implementing a USB Interface 211

12. Networks 215
Controller Area Network (CAN) 215
Ethernet 219

13. Analog 226
Amplifiers 226
Analog to Digital Conversion 229
Interfacing an External ADC 233
Temperature Sensor 235
Light Sensor 237
Accelerometer 240
Pressure Sensors 242

INTEL EXHIBIT 1008 Page 11
IPR2023-00787

Magnetic-Field Sensor 244
Digital to Analog Conversion 245
PWM 248
Motor Control 249
Switching Big Loads 256

14. The PIC Microcontrollers . . . 258
A Tale of Two Processors 258
Starting Simple 260
A Bigger PIC 263
PIC-Based Environmental Datalogger 265
Motor Control with a PIC 271

15. The AVR Microcontrollers . . . 277
The AVR Architecture 278
The ATtiny15 Processor 280
Downloading Code 287
A Bigger AVR 289
AVR-Based Datalogger 290
Bus Interfacing 291

16. 68HC11 . . . 316
Architecture of the 68HC11 316
A Simple 68HC11-Based Computer 317

17. MAXQ . . . 327
Architectural Overview 327
Schematics 329

18. 68000-Series Computers . . . 334
The 68000 Architecture 335
A Simple 68000-Based Computer 339

19. DSP-Based Controllers . . . 348
The DSP56800 351
A DSP56805-Based Computer 353
JTAG 361

Index . . . 363

INTEL EXHIBIT 1008 Page 12
IPR2023-00787

Preface

[Enlightenment] resides as comfortably in the circuits of a digital computer … as at the top of a mountain or in the petals of a flower.

—Robert M. Pirsig
Zen and the Art of Motorcycle Maintenance

Welcome to the second edition of *Designing Embedded Hardware*. In these pages, I hope to give you an understanding of the design process for creating computer hardware. Just as there is beauty in well-written software, there is beauty in well-designed hardware. With embedded computers, you get to understand the machine at all levels, at once aware of currents flowing through circuit traces and software executing complex algorithms. In fact, it is not possible to write embedded software without understanding the hardware, nor is it possible to design hardware without understanding software. You become involved with the machine to a degree beyond that which is possible with desktop computers. Best of all, it's a lot of fun.

In selecting chips and designs for this book, I have tried to choose, where possible, parts that are both trivial to use yet exceptionally useful. I have no connection, financial or otherwise, with any of the companies or businesses mentioned in this book, and I receive no benefits from them. Every component or product included in this book is there based on its own merits. You may notice a prevalence of components from certain manufacturers. This simply reflects my personal preference for using their chips, based on my experience. Such companies produce chips that are easy to use, are reliable and robust, have great technical support, and provide thorough and comprehensive technical data. In other words, they meet all the necessary requirements for inclusion in a book for beginners.

When the first edition of *Designing Embedded Hardware* was published, I deliberately left out software. There were two reasons for this. First, there are many good books already written on C programming, embedded firmware development in C, porting Linux to embedded systems, coding in Python, writing Java software, and so on. (And of course, the best of these are naturally published by O'Reilly.) The second

INTEL EXHIBIT 1008 Page 13
IPR2023-00787

reason is that assembly language, that most basic of programming tools, is so different from processor to processor that it would not have been possible to cover all the instruction sets of the processors in the book, let alone do them justice. However, I have decided to include some software in this edition. I won't even attempt to cover the instructions of each processor in this book. What I will do is show some simple assembly language techniques. While the instructions may be wildly different between architectures, the basic concepts are the same.

Also new to this book is a chapter on the Forth programming language. Forth is a relatively old language that has faded from the forefront of software development, and as such, it's rare to find a book giving the language good coverage. Forth is a very useful tool for embedded system development to which many engineers have yet to be exposed. The language is the basis of the Open Firmware standard and is used by design engineers at Apple, Sun, and many other manufacturers. It's a useful language to know, and it is worth taking the time to learn.

Many of the designs in this book look easy, and they are. They are intended as simple building blocks, allowing you to mix and match to achieve the embedded systems you need. I hope you will find this book useful. Once you've finished reading it, go and build something!

—John Catsoulis
Brisbane, Australia
January 2005
jtc@embedded.com.au
http://www.embedded.com.au

Organization of This Book

This book is informally divided into four sections. The first covers fundamental concepts and introductory material. The second section gives an overview of assembly language and Forth. From there, we'll look at peripherals and how to add functionality to your embedded systems. Finally, we'll look at a variety of processors widely used in embedded systems, and look at the design process for integrating them into computers.

Chapter 1 gives an overview of computer architectures and explains what constitutes an embedded system. Chapters 2 and 3 explore software with assembly language and Forth.

Chapter 4 provides some background electronics theory and introduces some important concepts. If you're already electronics-savvy, then you can skip on to Chapter 5, where we'll look at providing power for your embedded system. We'll also look at how to protect your embedded computer against electrical interference and other gremlins that can cause it grief. In Chapter 6, you'll see how to physically produce and debug an embedded computer system.

INTEL EXHIBIT 1008 Page 14
IPR2023-00787

Chapters 7 and 8 cover SPI and I²C, two protocols that allow a wide range of small peripherals to be added to microcontrollers. Chapters 9, 10, and 11 cover serial interfaces. These give our embedded system access to host computers and to external peripherals such as modems. We'll look at RS-232C, RS-422, Infrared communication, and USB.

Networks are covered in Chapter 12, where we'll see how to add a low-cost industrial network (CAN) to our embedded computer. Also in Chapter 12, we learn how to add an Ethernet port to our embedded system, by which we can connect to other computers, servers, and gateways and, through them, to the wider Internet.

In Chapter 13, we'll look at real-world interfacing. We'll see how to convert analog signals into digital values for processing and, conversely, how to convert digital values back into analog voltages. We'll also see how to interface sensors to our embedded system, whereby we can measure temperature, light, pressure, acceleration and magnetic fields. Also in Chapter 13, we'll look at Pulse Width Modulation and motor control. We'll see how to use an embedded computer to control small electric motors.

Chapter 14 begins the microprocessor section of the book, where we'll look at the first of our embedded processor architectures, the Microchip PIC. In subsequent chapters, we'll meet a variety of processors, from tiny standalone, 8-bit microcontrollers to 32-bit, bus-based chips with some computing grunt. While it is not possible to cover every embedded processor (as there are literally many hundreds), the chips chosen are representative of various classes of processor. The skills you learn will be adaptable to whatever processor you choose for your application.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact O'Reilly for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Designing Embedded Hardware*, by John Catsoulis. Copyright 2005 O'Reilly Media, Inc., 0-596-00755-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact the publisher at *permissions@oreilly.com*.

INTEL EXHIBIT 1008 Page 15
IPR2023-00787

Conventions

The conventions used in this book are as follows:

Main text

```
Source Code
```

Signal (active high)

Signal (active low)

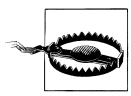
Hexadecimal numbers are denoted with the prefix *0x*, and sometimes with the prefix *$*, where appropriate for certain processors.

Binary numbers are denoted by the prefix *%*.

K is 1,024, while k is 1,000.



This icon indicates a tip, suggestion, or general note.



This icon indicates a warning or caution.

Safari® Enabled



When you see a Safari® Enabled icon on the cover of your favorite technology book, it means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

INTEL EXHIBIT 1008 Page 16
IPR2023-00787

There is a web page for this book, which lists errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/dbhardware2/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

To contact the author directly with comments or questions, send email to:

jtc@embedded.com.au

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

I'd like to give a special thank you to my editors, Andy Oram and Mike Hendrickson. I'd also like to thank Jon Orwant, editor of the first edition. In the past, I have often read in prefaces how authors thank their editors for the help they gave. It is only now that I understand the depth and significance of this help.

As you have no doubt already noticed, O'Reilly publishes beautifully presented books. I would like to thank the production team, especially Sanders Kleinfeld, for their hard work. This book is as much the result of their efforts as it is mine. I'd also like to thank David Chu for all his help.

Thank you to Dallas Semiconductor, Kathy Vehorn, Don Loomis, Mike Quarles, and Moe Rubenzahl for their assistance and for allowing me access to pre-release versions of the MAXQ processor. Thank you to Peter Paine, Donna Mack and Cooper Tools, Karen Rolan and Fluke Corporation, and Tektronix for allowing me to use their images in the book. Thank you also to Rupert Baines of Picochip for his assistance.

Geoff McDonald has been a great friend and has made many helpful suggestions regarding the content of this book. He also proofread the book, and I thank him for all his help.

Thanks to Michael, Mary, Renee, and Mitchell Lees. Michael did significant proofreading and offered many helpful comments.

I'd like to thank Dr. Jeff O'Keefe for his long friendship and support over the years. He's been a good friend ever since we were undergrads together, blowing up integrated circuits and irradiating lecturers in second-year lab!

Thank you to Prof. Neil Bergmann, Dr. John Williams, Keith Ball, Denis Bill, Barry Bettridge, and the staff of the School of ITEE at the University of Queensland for their help and support.

INTEL EXHIBIT 1008 Page 17
IPR2023-00787

I'd like to thank my friends and colleagues David Nicholls, Peter Stewart, Mark Gentile, Professor John Devlin, Richard Wiltshire, Michelle and Robert Salier, Addy and Derek Clark, Kam Tam, Phil McDonald, and Vamsi Madasu.

Finally and most importantly, I'd like to thank my extended family for their love and support. Most especially, I'd like to thank my sister Kris, and my two nephews, Andrew and James, whose love and good humor have made life worth living. I'd also like to thank Chris and Jeff Goopy for always being there, and my cousins Theo and Maree; David and Jenevieve; Michael, Andrew and Karen; Antony; and Fiona, Drew, Ashley, and Max for their care and support. A special thank you to my uncles, Vince and Dave Catsoulis, who have shown me the meaning of love, honor, and strength of character. I owe them much.

INTEL EXHIBIT 1008 Page 18
IPR2023-00787

CHAPTER 1

An Introduction to Computer Architecture

Each machine has its own, unique personality which probably could be defined as the intuitive sum total of everything you know and feel about it. This personality constantly changes, usually for the worse, but sometimes surprisingly for the better...

—Robert M. Pirsig
Zen and the Art of Motorcycle Maintenance

This book is about designing and building specialized computers. We all know what a computer is. It's that box that sits on your desk, quietly purring away (or rattling if the fan is shot), running your programs and regularly crashing (if you're not running some variety of Unix). Inside that box is the electronics that runs your software, stores your information, and connects you to the world. It's all about processing information. Designing a computer, therefore, is about designing a machine that holds and manipulates data.

Computer systems fall into essentially two separate categories. The first, and most obvious, is that of the desktop computer. When you say "computer" to someone, this is the machine that usually comes to her mind. The second type of computer is the embedded computer, a computer that is integrated into another system for the purposes of control and/or monitoring. Embedded computers are far more numerous than desktop systems, but far less obvious. Ask the average person how many computers he has in his home, and he might reply that he has one or two. In fact, he may have 30 or more, hidden inside his TVs, VCRs, DVD players, remote controls, washing machines, cell phones, air conditioners, game consoles, ovens, toys, and a host of other devices.

In this chapter, we'll look at computer architecture in general. This is applicable to both embedded and desktop computers, because the primary difference between an embedded machine and a general-purpose computer is its application. The basic principles of operation and the underlying architectures are fundamentally the same.

INTEL EXHIBIT 1008 Page 19
IPR2023-00787

Both have a processor, memory, and often several forms of input and output. The primary difference lies in their intended use, and this is reflected in the system design and their software. Desktop computers can run a variety of application programs, with system resources orchestrated by an operating system. By running different application programs, the functionality of the desktop computer is changed. One moment, it may be used as a word processor; the next it is an MP3 player or a database client. Which software is loaded and run is under user control.

In contrast, the embedded computer is normally dedicated to a specific task. In many cases, an embedded system is used to replace application-specific electronics. The advantage of using an embedded microprocessor over dedicated electronics is that the functionality of the system is determined by the software, not the hardware. This makes the embedded system easier to produce, and much easier to evolve, than a complicated circuit.

The embedded system typically has one application and one application only, which is permanently running. The embedded computer may or may not have an operating system, and rarely does it provide the user with the ability to arbitrarily install new software. The software is normally contained in the system's nonvolatile memory, unlike a desktop computer where the nonvolatile memory contains boot software and (maybe) low-level drivers only.

Embedded hardware is often much simpler than a desktop system, but it can also be far more complex too. An embedded computer may be implemented in a single chip with just a few support components, and its purpose may be as crude as a controller for a garden-watering system. Alternatively, the embedded computer may be a 150-processor, distributed parallel machine responsible for all the flight and control systems of a commercial jet. As diverse as embedded hardware may be, the underlying principles of design are the same.

This chapter introduces some important concepts relating to computer architecture, with specific emphasis on those topics relevant to embedded systems. Its purpose is to give you grounding before moving on to the more hands-on information that begins in Chapter 2. In this chapter, you'll learn about the basics of processors, interrupts, the difference between RISC and CISC, parallel systems, memory, and I/O.

Concepts

Let's start at the beginning.

In essence, a computer is a machine designed to process, store, and retrieve data. Data may be numbers in a spreadsheet, characters of text in a document, dots of color in an image, waveforms of sound, or the state of some system, such as an air conditioner or a CD player. *All data is stored in the computer as numbers*. It's easy to forget this when we're deep in C code, contemplating complex algorithms and data structures.

INTEL EXHIBIT 1008 Page 20
IPR2023-00787

The computer manipulates the data by performing operations on the numbers. Displaying an image on a screen is accomplished by moving an array of numbers to the video memory, each number representing a pixel of color. To play an MP3 audio file, the computer reads an array of numbers from disk and into memory, manipulates those numbers to convert the compressed audio data into raw audio data, and then outputs the new set of numbers (the raw audio data) to the audio chip.

Everything that a computer does, from web browsing to printing, involves moving and processing numbers. The electronics of a computer is nothing more than a system designed to hold, move, and change numbers.

A computer system is composed of many parts, both hardware and software. At the heart of the computer is the processor, the hardware that executes the computer programs. The computer also has memory, often several different types in one system. The memory is used to store programs while the processor is running them, as well as store the data that the programs are manipulating. The computer also has devices for storing data, or exchanging data with the outside world. These may allow the input of text via a keyboard, the display of information on a screen, or the movement of programs and data to or from a disk drive.

The software controls the operation and functionality of the computer. There are many "layers" of software in the computer (Figure 1-1). Typically, a given layer will only interact with the layers immediately above or below it.

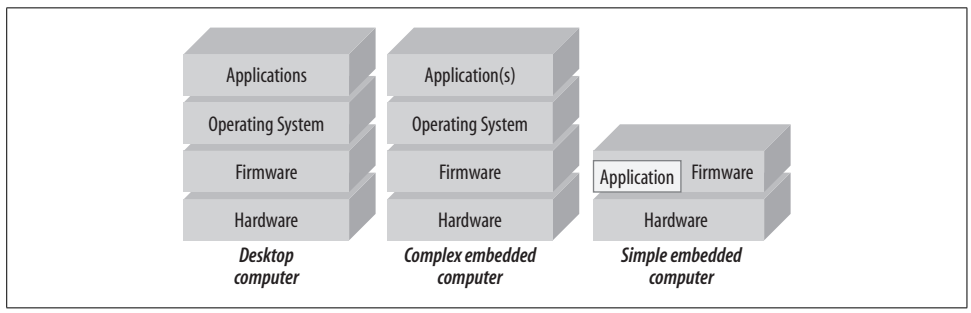


Figure 1-1. Software layers

At the lowest level, there are programs that are run by the processor when the computer first powers up. These programs initialize the other hardware subsystems to a known state and configure the computer for correct operation. This software, because it is permanently stored in the computer's memory, is known as *firmware*.

The *bootloader* is located in the firmware. The bootloader is a special program run by the processor that reads the operating system from disk (or nonvolatile memory or network interface) and places it in memory so that the processor may then run it. The bootloader is present in desktop computers and workstations, and may be present in some embedded computers.

INTEL EXHIBIT 1008 Page 21
IPR2023-00787

Above the firmware, the operating system controls the operation of the computer. It organizes the use of memory and controls devices such as the keyboard, mouse, screen, disk drives, and so on. It is also the software that often provides an interface to the user, enabling her to run application programs and access her files on disk. The operating system typically provides a set of software tools for application programs, providing a mechanism by which they too can access the screen, disk drives, and so on. Not all embedded systems use or even need an operating system. Often, an embedded system will simply run code dedicated to its task, and the presence of an operating system is overkill. In other instances, such as network routers, an operating system provides necessary software integration and greatly simplifies the development process. Whether an operating system is needed and useful really depends on the intended purpose of the embedded computer and, to a lesser degree, on the preference of the designer.

At the highest level, the *application software* constitutes the programs that provide the functionality of the computer. Everything below the application is considered *system software*. For embedded computers, the boundary between application and system software is often blurred. This reflects the underlying principle in embedded design that a system should be designed to achieve its objective in as simple and straightforward a manner as possible.

Processors

The processor is the most important part of a computer, the component around which everything else is centered. In essence, the processor is the computing part of the computer. A processor is an electronic device capable of manipulating data (information) in a way specified by a sequence of instructions. The instructions are also known as *opcodes* or *machine code*. This sequence of instructions may be altered to suit the application, and, hence, computers are programmable. A sequence of instructions is what constitutes a program.

Instructions in a computer are numbers, just like data. Different numbers, when read and executed by a processor, cause different things to happen. A good analogy is the mechanism of a music box. A music box has a rotating drum with little bumps, and a row of prongs. As the drum rotates, different prongs in turn are activated by the bumps, and music is produced. In a similar way, the bit patterns of instructions feed into the execution unit of the processor. Different bit patterns activate or deactivate different parts of the processing core. Thus, the bit pattern of a given instruction may activate an addition operation, while another bit pattern may cause a byte to be stored to memory.

A sequence of instructions is a machine-code program. Each type of processor has a different *instruction set*, meaning that the functionality of the instructions (and the bit patterns that activate them) varies. Processor instructions are often quite simple, such as “add two numbers” or “call this function.” In some processors, however,

INTEL EXHIBIT 1008 Page 22
IPR2023-00787

they can be as complex and sophisticated as "if the result of the last operation was zero, then use this particular number to reference another number in memory, and then increment the first number once you've finished." This will be covered in more detail in the section on CISC and RISC processors, later in this chapter.

Basic System Architecture

The processor alone is incapable of successfully performing any tasks. It requires memory (for program and data storage), support logic, and at least one I/O device ("input/output device") used to transfer data between the computer and the outside world. The basic computer system is shown in Figure 1-2.

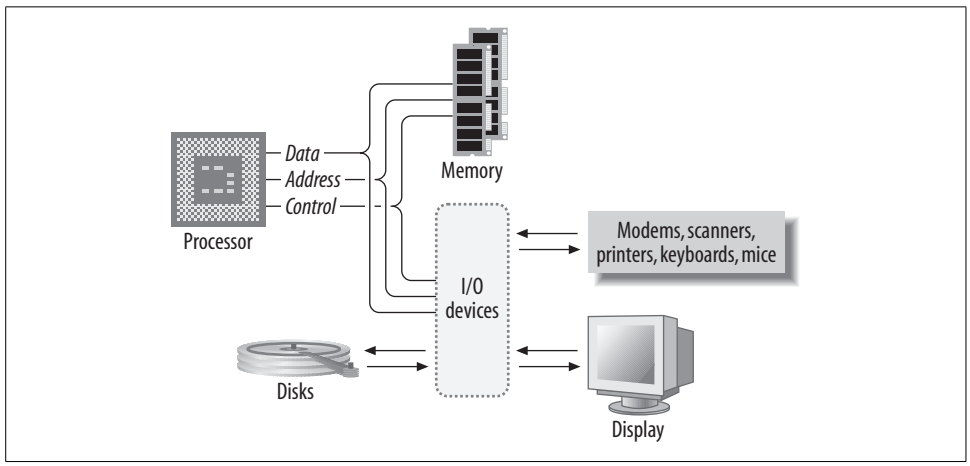


Figure 1-2. Basic computer system

A *microprocessor* is a processor implemented (usually) on a single, integrated circuit. With the exception of those found in some large supercomputers, nearly all modern processors are microprocessors, and the two terms are often used interchangeably. Common microprocessors in use today are the Intel Pentium series, Freescale/IBM PowerPC, MIPS, ARM, and the Sun SPARC, among others. A microprocessor is sometimes also known as a *CPU* (*Central Processing Unit*).

A *microcontroller* is a processor, memory, and some I/O devices contained within a single, integrated circuit, and intended for use in embedded systems. The buses that interconnect the processor with its I/O exist within the same integrated circuit. The range of available microcontrollers is very broad. They range from the tiny PICs and AVRs (to be covered in this book) to PowerPC processors with built-in I/O, intended for embedded applications. In this book, we will look at both microprocessors and microcontrollers.

Microcontrollers are very similar to *System-on-Chip* (*SoC*) processors, intended for use in conventional computers such as PCs and workstations. SoC processors have a

INTEL EXHIBIT 1008 Page 23
IPR2023-00787

different suite of I/O, reflecting their intended application, and are designed to be interfaced to large banks of external memory. Microcontrollers usually have all their memory on-chip and may provide only limited support for external memory devices.

The memory of the computer system contains both the instructions that the processor will execute and the data it will manipulate. The memory of a computer system is never empty. It always contains something, whether it be instructions, meaningful data, or just the random garbage that appeared in the memory when the system powered up.

Instructions are read (fetched) from memory, while data is both read from and written to memory, as shown in Figure 1-3.

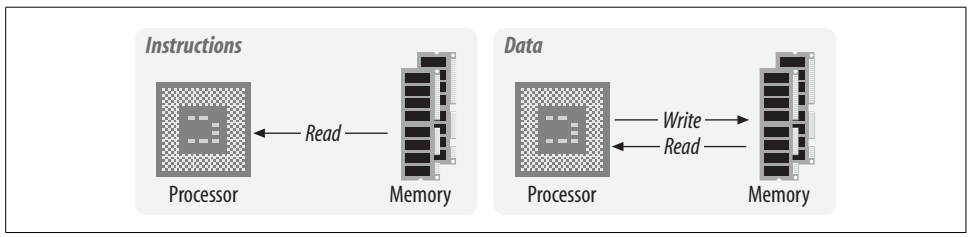


Figure 1-3. Data flow

This form of computer architecture is known as a *Von Neumann machine*, named after John Von Neumann, one of the originators of the concept. With very few exceptions, nearly all modern computers follow this form. Von Neumann computers are what can be termed control-flow computers. The steps taken by the computer are governed by the sequential control of a program. In other words, the computer follows a step-by-step program that governs its operation.



There are some interesting non–Von Neumann architectures, such as the massively parallel Connection Machine and the nascent efforts at building biological and quantum computers, or neural networks.

A classical Von Neumann machine has several distinguishing characteristics:

There is no real difference between data and instructions. A processor can be directed to begin execution at a given point in memory, and it has no way of knowing whether the sequence of numbers beginning at that point is data or instructions. The instruction 0x4143 may also be data (the number 0x4143, or the ASCII characters "A" and "C"). The processor has no way of telling what is data or what is an instruction. If a number is to be executed by the processor, it is an instruction; if it is to be manipulated, it is data.

Because of this lack of distinction, the processor is capable of changing its instructions (treating them as data) under program control. And because the processor has no way of distinguishing between data and instruction, it will

INTEL EXHIBIT 1008 Page 24
IPR2023-00787

blindly execute anything that it is given, whether it is a meaningful sequence of instructions or not.

Data has no inherent meaning. There is nothing to distinguish between a number that represents a dot of color in an image and a number that represents a character in a text document. Meaning comes from how these numbers are treated under the execution of a program.

Data and instructions share the same memory. This means that sequences of instructions in a program may be treated as data by another program. A compiler creates a program binary by generating a sequence of numbers (instructions) in memory. To the compiler, the compiled program is just data, and it is treated as such. It is a program only when the processor begins execution. Similarly, an operating system loading an application program from disk does so by treating the sequence of instructions of that program as data. The program is loaded to memory just as an image or text file would be, and this is possible due to the shared memory space.

Memory is a linear (one-dimensional) array of storage locations. The processor's memory space may contain the operating system, various programs, and their associated data, all within the same linear space.

Each location in the memory space has a unique, sequential address. The address of a memory location is used to specify (and select) that location. The memory space is also known as the *address space*, and how that address space is partitioned between different memory and I/O devices is known as the *memory map*. The address space is the array of all addressable memory locations. In an 8-bit processor (such as the 68HC11) with a 16-bit address bus, this works out to be 2^{16} = 65,536 = 64K of memory. Hence, the processor is said to have a 64K address space. Processors with 32-bit address buses can access 2^{32} = 4,294,967,296 = 4G of memory.

Some processors, notably the Intel x86 family, have a separate address space for I/O devices with separate instructions for accessing this space. This is known as *ported I/O*. However, most processors make no distinction between memory devices and I/O devices within the address space. I/O devices exist within the same linear space as memory devices, and the same instructions are used to access each. This is known as *memory-mapped I/O* (Figure 1-4). Memory-mapped I/O is certainly the most common form. Ported I/O address spaces are becoming rare, and the use of the term even rarer.

Most microprocessors available are standard Von Neumann machines. The main deviation from this is the *Harvard architecture*, in which instructions and data have different memory spaces (Figure 1-5) with separate address, data, and control buses for each memory space. This has a number of advantages in that instruction and data fetches can occur concurrently, and the size of an instruction is not set by the size of the standard data unit (word).

INTEL EXHIBIT 1008 Page 25
IPR2023-00787

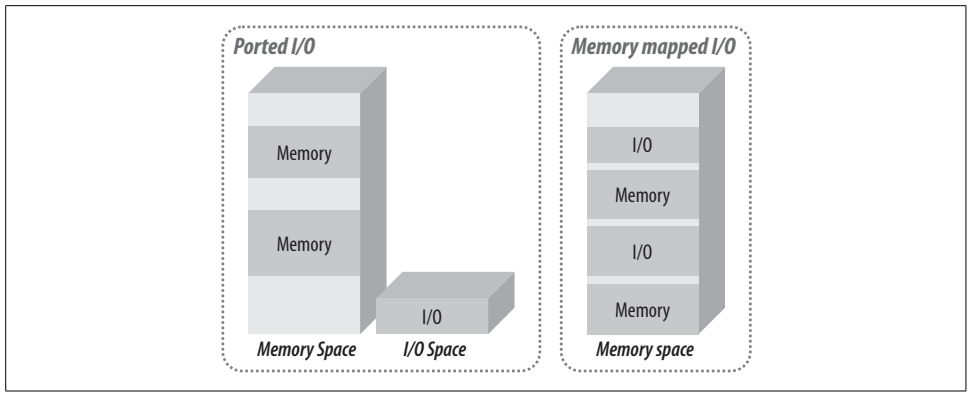


Figure 1-4. Ported versus memory-mapped I/O spaces

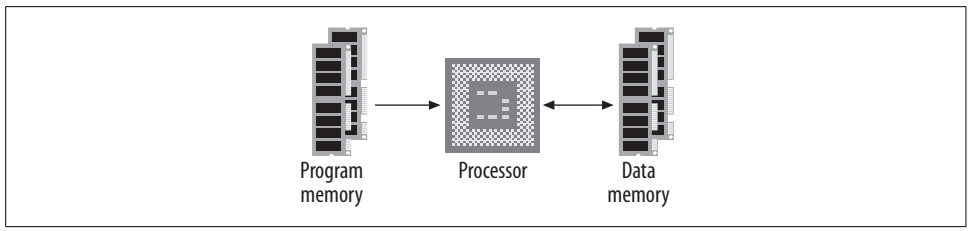


Figure 1-5. Harvard architecture

Buses

A bus is a physical group of signal lines that have a related function. Buses allow for the transfer of electrical signals between different parts of the computer system and thereby transfer information from one device to another. For example, the data bus is the group of signal lines that carry data between the processor and the various subsystems that comprise the computer. The "width" of a bus is the number of signal lines dedicated to transferring information. For example, an 8-bit-wide bus transfers 8 bits of data in parallel.

The majority of microprocessors available today (with some exceptions) use the three-bus system architecture (Figure 1-6). The three buses are the *address bus*, the *data bus*, and the *control bus*.

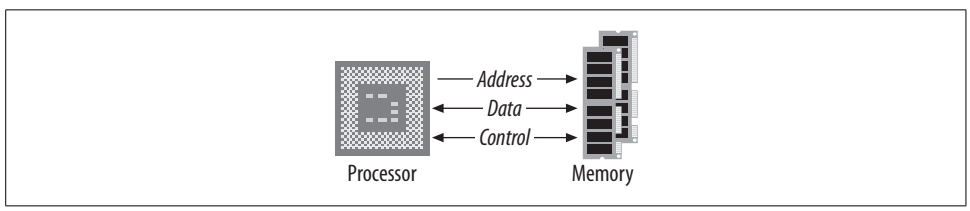


Figure 1-6. Three-bus system

INTEL EXHIBIT 1008 Page 26
IPR2023-00787

The data bus is bidirectional, the direction of transfer being determined by the processor. The address bus carries the address, which points to the location in memory that the processor is attempting to access. It is the job of external circuitry to determine in which external device a given memory location exists and to activate that device. This is known as *address decoding*. The control bus carries information from the processor about the state of the current access, such as whether it is a write or a read operation. The control bus can also carry information back to the processor regarding the current access, such as an address error. Different processors have different control lines, but there are some control lines that are common among many processors. The control bus may consist of output signals such as read, write, valid address, etc. A processor usually has several input control lines too, such as reset, one or more interrupt lines, and a clock input.



A few years ago, I had the opportunity to wander through, in, and around CSIRAC (pronounced "sigh-rack"). This was one of the world's first digital computers, designed and built in Sydney, Australia, in the late 1940s. It was a massive machine, filling a very big room with the type of solid hardware that you can really kick. It was quite an experience looking over the old machine. I remember at one stage walking *through* the disk controller (it was the size of small room) and looking up at a mass of wires strung overhead. I asked what they were for. "That's the data bus!" came the reply.

CSIRAC is now housed in the museum of the University of Melbourne. You can take an online tour of the machine, and even download a simulator, at *http://www.cs.mu.oz.au/csirac*.

Processor operation

There are six basic types of access that a processor can perform with external chips. The processor can write data to memory or write data to an I/O device, read data from memory or read data from an I/O device, read instructions from memory, and perform internal manipulation of data within the processor.

In many systems, writing data to memory is functionally identical to writing data to an I/O device. Similarly, reading data from memory constitutes the same external operation as reading data from an I/O device, or reading an instruction from memory. In other words, the processor makes no distinction between memory and I/O.

The internal data storage of the processor is known as its *registers*. The processor has a limited number of registers, and these are used to hold the current data/operands that the processor is manipulating.

ALU

The Arithmetic Logic Unit (ALU) performs the internal arithmetic manipulation of data in the processor. The instructions that are read and executed by the processor control the data flow between the registers and the ALU. The instructions also

INTEL EXHIBIT 1008 Page 27
IPR2023-00787

control the arithmetic operations performed by the ALU via the ALU's control inputs. A symbolic representation of an ALU is shown in Figure 1-7.

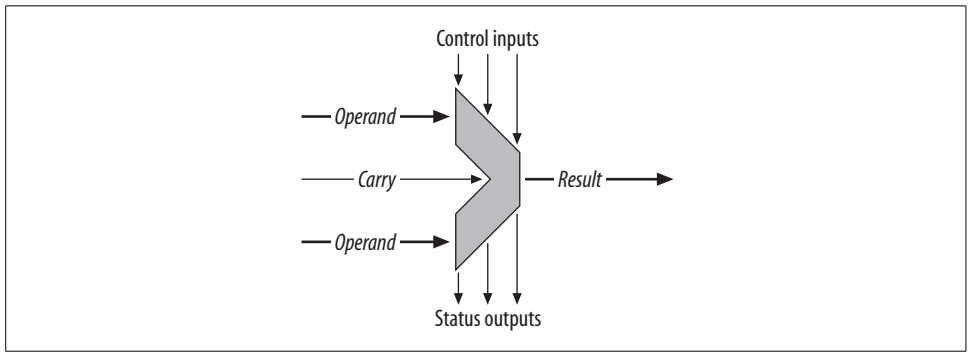


Figure 1-7. ALU block diagram

Whenever instructed by the processor, the ALU performs an operation (typically one of addition, subtraction, NOT, AND, OR, XOR, shift left/right, or rotate left/right) on one or more values. These values, called *operands*, are typically obtained from two registers, or from one register and a memory location. The result of the operation is then placed back into a given destination register or memory location. The status outputs indicate any special attributes about the operation, such as whether the result was zero, negative, or if an overflow or carry occurred. Some processors have separate units for multiplication and division, and for bit shifting, providing faster operation and increased throughput.

Each architecture has its own unique ALU features, and this can vary greatly from one processor to another. However, all are just variations on a theme, and all share the common characteristics just described.

Interrupts

Interrupts (also known as *traps* or *exceptions* in some processors) are a technique of diverting the processor from the execution of the current program so that it may deal with some event that has occurred. Such an event may be an error from a peripheral, or simply that an I/O device has finished the last task it was given and is now ready for another. An interrupt is generated in your computer every time you type a key or move the mouse. You can think of it as a hardware-generated function call.

Interrupts free the processor from having to continuously check the I/O devices to determine whether they require service. Instead, the processor may continue with other tasks. The I/O devices will notify it when they require attention by asserting one of the processor's interrupt inputs. Interrupts can be of varying priorities in some processors, thereby assigning differing importance to the events that can interrupt the processor. If the processor is servicing a low-priority interrupt, it will pause it in order to service a higher-priority interrupt. However, if the processor is servicing an

INTEL EXHIBIT 1008 Page 28
IPR2023-00787

interrupt and a second, lower-priority interrupt occurs, the processor will ignore that interrupt until it has finished the higher-priority service.

When an interrupt occurs, the usual procedure is for the processor to save its state by pushing its registers and program counter onto the stack. The processor then loads an *interrupt vector* into the program counter. The interrupt vector is the address at which an *interrupt service routine* (*ISR*) lies. Thus, loading the vector into the program counter causes the processor to begin execution of the ISR, performing whatever service the interrupting device required. The last instruction of an ISR is always a *Return from Interrupt* instruction. This causes the processor to reload its saved state (registers and program counter) from the stack and resume its original program. Interrupts are largely transparent to the original program. This means that the original program is completely "unaware" that the processor was interrupted, save for a lost interval of time.

Processors with *shadow registers* use these to save their current state, rather than pushing their register bank onto the stack. This saves considerable memory accesses (and therefore time) when processing an interrupt. However, since only one set of shadow registers exists, a processor servicing multiple interrupts must "manually" preserve the state of the registers before servicing the higher interrupt. If it does not, important state information will be lost. Upon returning from an ISR, the contents of the shadow registers are swapped back into the main register array.

Hardware interrupts

There are two ways of telling when an I/O device (such as a serial controller or a disk controller) is ready for the next sequence of data to be transferred. The first is *busy waiting* or *polling*, where the processor continuously checks the device's status register until the device is ready. This wastes the processor's time but is the simplest to implement. For some time-critical applications, polling can reduce the time it takes for the processor to respond to a change of state in a peripheral.

A better way is for the device to generate an interrupt to the processor when it is ready for a transfer to take place. Small, simple processors may only have one (or two) interrupt inputs, so several external devices may have to share the interrupt lines of the processor. When an interrupt occurs, the processor must check each device to determine which one generated the interrupt. (This can also be considered a form of polling.) The advantage of interrupt polling over ordinary polling is that the polling occurs only when there is a need to service a device. Polling interrupts is suitable only in systems that have a small number of devices; otherwise, the processor will spend too long trying to determine the source of the interrupt.

The other technique of servicing an interrupt is by using *vectored interrupts*,* by which the interrupting device provides the interrupt vector that the processor is to

* Note that this is different from an interrupt vector stored in memory.

INTEL EXHIBIT 1008 Page 29
IPR2023-00787

take. Vectored interrupts reduce considerably the time it takes the processor to determine the source of the interrupt. If an interrupt request can be generated from more than one source, it is therefore necessary to assign priorities (levels) to the different interrupts. This can be done in either hardware or software, depending on the particular application. In this scheme, the processor has numerous interrupt lines, with each interrupt corresponding to a given interrupt vector. So, for example, when an interrupt of priority 7 occurs (interrupt lines corresponding to "7" are asserted), the processor loads vector 7 into its program counter and starts executing the service routine specific to interrupt 7.

Vectored interrupts can be taken one step further. Some processors and devices support the device by actually placing the appropriate vector onto the data bus when they generate an interrupt. This means the system can be even more versatile, so that instead of being limited to one interrupt per peripheral, each device can supply an interrupt vector specific to the event that is causing the interrupt. However, the processor must support this function, and most do not.

Some processors have a feature known as a *fast hardware interrupt*. With this interrupt, only the program counter is saved. It assumes that the ISR will protect the contents of the registers by manually saving their state as required. Fast interrupts are useful when an I/O device requires a very fast response from a processor and cannot wait for the processor to save all its registers to the stack. A special (and separate) interrupt line is used to generate fast interrupts.

Software interrupts

A software interrupt is generated by an instruction. It is the lowest-priority interrupt and is generally used by programs to request a service to be performed by the system software (operating system or firmware).

So why are software interrupts used? Why isn't the appropriate section of code called directly? For that matter, why use an operating system to perform tasks for us at all? It gets back to compatibility. Jumping to a subroutine (calling a function) is jumping to a specific address in memory. A future version of the system software may not locate the subroutines at the same addresses as earlier versions. By using a software interrupt, our program does not need to know where the routines lie. It relies on the entry in the vector table to direct it to the correct location.

CISC and RISC

There are two major approaches to processor architecture: *Complex Instruction Set Computer* (*CISC*, pronounced "Sisk") processors and *Reduced Instruction Set Computer* (*RISC*) processors. Classic CISC processors are the Intel x86, Motorola 68xxx, and National Semiconductor 32xxx processors, and, to a lesser degree, the

INTEL EXHIBIT 1008 Page 30
IPR2023-00787

Intel Pentium. Common RISC architectures are the Freescale/IBM PowerPC, the MIPS architecture, Sun's SPARC, the ARM, the Atmel AVR, and the Microchip PIC.

CISC processors have a single processing unit, external memory, and a relatively small register set and many hundreds of different instructions. In many ways, they are just smaller versions of the processing units of mainframe computers from the 1960s.

The tendency in processor design throughout the late 70s and early 80s was toward bigger and more complicated instruction sets. Need to input a string of characters from an I/O port? Well, with CISC (80x86 family), there's a *single instruction* to do it! The diversity of instructions in a CISC processor can run to well over 1,000 opcodes in some processors, such as the Motorola 68000. This had the advantage of making the job of the assembly-language programmer easier, since you had to write fewer lines of code to get the job done. As memory was slow and expensive, it also made sense to make each instruction do more. This reduced the number of instructions needed to perform a given function, and thereby reduced memory space and the number of memory accesses required to fetch instructions. As memory got cheaper and faster, and compilers became more efficient, the relative advantages of the CISC approach began to diminish. One main disadvantage of CISC is that the processors themselves get increasingly complicated as a consequence of supporting such a large and diverse instruction set. The control and instruction decode units are complex and slow, the silicon is large and hard to produce, and they consume a lot of power and therefore generate a lot of heat. As processors became more advanced, the overheads that CISC imposed on the silicon became oppressive.

A given processor feature when considered alone may increase processor performance but may actually decrease the performance of the total system, if it increases the total complexity of the device. It was found that by streamlining the instruction set to the most commonly used instructions, the processors become simpler and faster. Fewer cycles are required to decode and execute each instruction, and the cycles are shorter. The drawback is that more (simpler) instructions are required to perform a task, but this is more than made up for in the performance boost to the processor. For example, if both cycle time and the number of cycles per instruction are each reduced by a factor of four, while the number of instructions required to perform a task grows by 50%, the execution of the processor is sped up by a factor of eight.

The realization of this led to a rethink of processor design. The result was the RISC architecture, which has led to the development of very high-performance processors. The basic philosophy behind RISC is to move the complexity from the silicon to the language compiler. The hardware is kept as simple and fast as possible.

A given complex instruction can be performed by a sequence of much simpler instructions. For example, many processors have an `xor` (exclusive OR) instruction

INTEL EXHIBIT 1008 Page 31
IPR2023-00787

for bit manipulation, and they also have a `clear` instruction to set a given register to zero. However, a register can also be set to zero by `xor`-ing it with itself. Thus, the separate `clear` instruction is no longer required. It can be replaced with the already present `xor`. Further, many processors are able to clear a memory location directly by writing a zero to it. That same function can be implemented by clearing a register and then storing that register to the memory location. The instruction to load a register with a literal number can be replaced with the instruction for clearing a register, followed by an `add` instruction with the literal number as its operand. Thus, six instructions (`xor`, `clear` *`reg`*, `clear` *`memory`*, `load` *`literal`*, `store`, and `add`) can be replaced with just three (`xor`, `store`, and `add`).

So the following CISC assembly pseudocode:

```
clear 0x1000    ; clear memory location 0x1000
load  r1,#5     ; load register 1 with the value 5
```

becomes the following RISC pseudocode:

```
xor   r1,r1      ; clear register 1
store r1,0x1000  ; clear memory location 0x1000
add   r1,#5      ; load register 1 with the value 5
```

The resulting code size is bigger, but the reduced complexity of the instruction decode unit can result in faster overall operation. Dozens of such code optimizations exist to give RISC its simplicity.

RISC processors have a number of distinguishing characteristics. They have large register sets (in some architectures numbering over 1,000), thereby reducing the number of times the processor must access main memory. Often-used variables can be left inside the processor, reducing the number of accesses to (slow) external memory. Compilers of high-level languages (such as C) take advantage of this to optimize processor performance.

By having smaller and simpler instruction decode units, RISC processors have fast instruction execution, and this also reduces the size and power consumption of the processing unit. Generally, RISC instructions will take only one or two cycles to execute (this depends greatly on the particular processor). This is in contrast to instructions for a CISC processor, whose instructions may take many tens of cycles to execute. For example, one instruction (integer multiplication) on an 80486 CISC processor takes 42 cycles to complete. The same instruction on a RISC processor may take just one cycle. Instructions on a RISC processor have a simple format. All instructions are generally the same length (which makes instruction decode units simpler).

RISC processors implement what is known as a "load/store" architecture. This means that the only instructions that actually reference memory are `load` and `store`. In contrast, many (most) instructions on a CISC processor may access or manipulate memory. On a RISC processor, all other instructions (aside from `load` and `store`) work on the registers only. This facilitates the ability of RISC processors to complete

INTEL EXHIBIT 1008 Page 32
IPR2023-00787

(most of) their instructions in a single cycle. Consequently, RISC processors do not have the range of addressing modes that are found on CISC processors.

RISC processors also often have pipelined instruction execution. This means that while one instruction is being executed, the next instruction in the sequence is being decoded, while the third one is being fetched. At any given moment, several instructions will be in the pipeline and in the process of being executed. Again, this provides improved processor performance. Thus, even though not all instructions may be completed in a single cycle, the processor may issue and retire instructions on each cycle, thereby achieving effective single-cycle execution. Some RISC processors have overlapped instruction execution, in which `load` operations may allow the execution of subsequent, unrelated instructions to continue before the data requested by the `load` has been returned from memory. This allows these instructions to overlap the `load`, thereby improving processor performance.

Due to their low power consumption and computing power, RISC processors are becoming widely used, particularly in embedded computer systems, and many RISC attributes are appearing in what are traditionally CISC architectures (such as with the Intel Pentium). Ironically, many RISC architectures are adding some CISC-like features, and so the distinction between RISC and CISC is blurring.

An excellent discussion of RISC architectures and processor performance topics can be found in Kevin Dowd and Charles Severance's *High Performance Computing* (O'Reilly).

So, which is better for embedded and industrial applications, RISC or CISC? If power consumption needs to be low, then RISC is probably the better architecture to use. However, if the available space for program storage is small, then a CISC processor may be a better alternative, since CISC instructions get more "bang" for the byte.

Digital Signal Processors

A special type of processor architecture is that of the *Digital Signal Processor* (*DSP*). These processors have instruction sets and architectures optimized for numerical processing of array data. They often extend the Harvard architecture concept further by not only having separate data and code spaces, but also by splitting the data spaces into two or more banks. This allows concurrent instruction fetch and data accesses for multiple operands. As such, DSPs can have very high throughput and can outperform both CISC and RISC processors in certain applications.

DSPs have special hardware well suited to numerical processing of arrays. They often have *hardware looping*, whereby special registers allow for and control the repeated execution of an instruction sequence. This is also often known as *zero-overhead looping*, since no conditions need to be explicitly tested by the software as part of the looping process. DSPs often have dedicated hardware for increasing the speed of arithmetic operations. High-speed multipliers, Multiply-And-Accumulate (MAC) units, and barrel shifters are common features.

INTEL EXHIBIT 1008 Page 33
IPR2023-00787

DSP processors are commonly used in embedded applications, and many conventional embedded microcontrollers include some DSP functionality.

Memory

Memory is used to hold data and software for the processor. There is a variety of memory types, and often a mix is used within a single system. Some memory will retain its contents while there is no power, yet will be slow to access. Other memory devices will be high-capacity, yet will require additional support circuitry and will be slower to access. Still other memory devices will trade capacity for speed, yielding relatively small devices, yet will be capable of keeping up with the fastest of processors.

Memory chips can be organized in two ways, either in *word-organized* or *bit-organized* schemes. In the word-organized scheme, complete nybbles, bytes, or words are stored within a single component, whereas with bit-organized memory, each bit of a byte or word is allocated to a separate component (Figure 1-8).

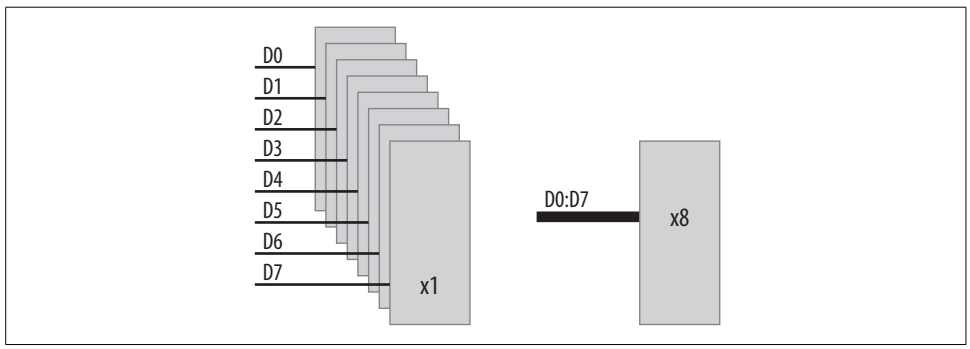


Figure 1-8. Eight bit-organized 8×1 devices and one word-organized 8×8 device

Memory chips come in different sizes, with the width specified as part of the size description. For instance, a DRAM (dynamic RAM) chip might be described as being 4M×1 (bit-organized), whereas a SRAM (static RAM) may be 512K×8 (word-organized). In both cases, each chip has exactly the same storage capacity, but organized in different ways. In the DRAM case, it would take eight chips to complete a memory block for an 8-bit data bus, whereas the SRAM would only require one chip. However, because the DRAMs are organized in parallel, they are accessed simultaneously. The final size of the DRAM block is (4M×1)×8 devices, which is 32M. It is common practice for multiple DRAMs to be placed on a *memory module*. This is the common way that DRAMs are installed in standard computers.

The common widths for memory chips are x1, x4, and x8, although x16 devices are available. A 32-bit-wide bus can be implemented with thirty-two x1 devices, eight x4 devices, or four x8 devices.

INTEL EXHIBIT 1008 Page 34
IPR2023-00787

RAM

RAM stands for *Random Access Memory*. This is a bit of a misnomer, since most (all) computer memory may be considered "random access." RAM is the "working memory" in the computer system. It is where the processor may easily write data for temporary storage. RAM is generally *volatile*, losing its contents when the system loses power. Any information stored in RAM that must be retained must be written to some form of permanent storage before the system powers down. There are special nonvolatile RAMs that integrate a battery-backup system, such that the RAM remains powered even when the rest of the computer system has shut down.

RAMs generally fall into two categories: *static RAM* (also known as *SRAM*) and *dynamic RAM* (also known as *DRAM*).

SRAMs use pairs of logic gates to hold each bit of data. SRAMs are the fastest form of RAM available, require little external support circuitry, and have relatively low power consumption. Their drawbacks are that their capacity is considerably less than DRAM, while being much more expensive. Their relatively low capacity requires more chips to be used to implement the same amount of memory. A modern PC built using nothing but SRAM would be a *considerably* bigger machine and would cost a small fortune to produce. (It would be *very* fast, however.)

DRAM uses arrays of what are essentially capacitors to hold individual bits of data. The capacitor arrays will hold their charge only for a short period before it begins to diminish. Therefore, DRAMs need continuous refreshing, every few milliseconds or so. This perpetual need for refreshing requires additional support and can delay processor access to the memory. If a processor access conflicts with the need to refresh the array, the refresh cycle must take precedence.

DRAMs are the highest-capacity memory devices available and come in a wide and diverse variety of subspecies. Interfacing DRAMs to small microcontrollers is generally not possible, and certainly not practical. Most processors with large address spaces include support for DRAMs. Connecting DRAMs to such processors is simply a case of "connecting the dots" (or pins, as the case may be). For those processors that do not include DRAM support, special DRAM controller chips are available that make interfacing the DRAMs very simple indeed.

Many processors have instruction and/or data *caches*, which store recent memory accesses. These caches are (often, but not always) internal to the processors and are implemented with fast memory cells and high-speed data paths. Instruction execution normally runs out of the instruction cache, providing for fast execution. The processor is capable of rapidly reloading the caches from main memory should a cache miss occur. Some processors have logic that is able to anticipate a cache miss and begin the cache reload prior to the cache miss occurring. Caches are implemented using very fast SRAM and are most often used in large systems to compensate for the slowness of DRAM.

INTEL EXHIBIT 1008 Page 35
IPR2023-00787

ROM

ROM stands for *Read-Only Memory*. This is also a bit of a misnomer, since many (modern) ROMs can also be written to. ROMs are *nonvolatile memory*, requiring no power to retain their contents. They are generally slower than RAM, and considerably slower than fast static RAM.

The primary purpose of ROM within a system is to hold the code (and sometimes data) that needs to be present at power-up. Such software is generally known as *firmware* and contains software to initialize the computer by placing I/O devices into a known state. It may contain either a bootloader program to load an operating system off disk or network or, in the case of an embedded system, it may contain the application itself.

Many microcontrollers contain on-chip ROM, thereby reducing component count and simplifying system design.

Standard ROM is fabricated (in a simplistic sense) from a large array of diodes. The unwritten bit state for a ROM is all 1s, each byte location reading as 0xFF. The process of loading software into a ROM is known as *burning the ROM*. This term comes from the fact that the programming process is performed by passing a sufficiently large current through the appropriate diodes to "blow them," or *burn* them, thereby creating a zero at that bit location. A device known as a *ROM burner* can accomplish this, or, if the system supports it, the ROM may be programmed in-circuit. This is known as *In-System Programming* (*ISP*) or, sometimes, *In-Circuit Programming* (*ICP*).

One-Time Programmable (*OTP*) ROMs, as the name implies, can be burned once only. Computer manufacturers typically use them in systems where the firmware is stable and the product is shipping in bulk to customers. *Mask-programmable* ROMs are also one-time programmable, but unlike OTPs, they are burned by the chip manufacturer prior to shipping. Like OTPs, they are used once the software is known to be stable and have the advantage of lowering production costs for large shipments.

EPROM

OTP ROMs are great for shipping in final products, but they are wasteful for debugging, since with each iteration of code change, a new chip must be burned and the old one thrown away. As such, OTPs make for a very expensive development option. No sane person uses OTPs for development work.

A (slightly) better choice for system development and debugging is the *Erasable Programmable Read-Only Memory*, or EPROM. Shining ultraviolet light through a small window on the top of the chip can erase the EPROM, allowing it to be reprogrammed and reused. They are pin- and signal-compatible with comparable OTP

INTEL EXHIBIT 1008 Page 36
IPR2023-00787

and mask devices. Thus, an EPROM can be used during development, while OTPs can be used in production with no change to the rest of the system.

EPROMs and their equivalent OTP cousins range in capacity from a few kilobytes (exceedingly rare these days) to a megabyte or more.

The drawback with EPROM technology is that the chip must be removed from the circuit to be erased, and the erasure can take many minutes to complete. The chip is then inserted into the burner, loaded with software, and then placed back in-circuit. This can lead to very slow debug cycles. Further, it makes the device useless for storing changeable system parameters. EPROMs are relatively rare these days. You can still buy them, but flash-based memory (to be discussed shortly) is far more common and is the medium of choice.

EEROM

EEROM is *Electrically Erasable Read-Only Memory*, also known as *EEPROM* (*Electrically Erasable Programmable Read-Only Memory*). Very rarely, it is also called *Electrically Alterable Read-Only Memory* (*EAROM*). EEROM can be pronounced as either "e-e ROM" or "e-squared ROM," or sometimes just "e-squared" for short.

EEROMs can be erased and reprogrammed in-circuit. Their capacity is significantly smaller than standard ROM (typically only a few kilobytes), and so they are not suited to holding firmware. Instead, they are typically used for holding system parameters and mode information to be retained during power-off.

It is common for many microcontrollers to incorporate a small EEROM on-chip for holding system parameters. This is especially useful in embedded systems and may be used for storing network addresses, configuration settings, serial numbers, servicing records, and so on.

Flash

Flash is the newest ROM technology and is now dominant. Flash memory has the reprogrammability of EEROM and the large capacity of standard ROMs. Flash chips are sometimes referred to as "flash ROMs" or "flash RAMs." Since they are not like standard ROMs or standard RAMs, I prefer just to call them "flash" and save on the confusion.

Flash is normally organized as sectors and has the advantage that individual sectors may be erased and rewritten without affecting the contents of the rest of the device. Typically, before a sector can be written, it must be erased. It can't just be written over as with a RAM.

There are several different flash technologies, and the erasing and programming requirements of flash devices vary from manufacturer to manufacturer.

INTEL EXHIBIT 1008 Page 37
IPR2023-00787

Input/Output

The address space of the processor can contain devices other than memory. These are input/output devices (I/O devices, also known as *peripherals*) and are used by the processor to communicate with the external world. Some examples are serial controllers that communicate with keyboards, mice, modems, etc.; parallel I/O devices that control some external subsystem; or disk-drive controllers, video and audio controllers, or network interfaces.

There are three main ways in which data may be exchanged with the external world:

Programmed I/O
: The processor accepts or delivers data at times convenient to it (the processor).

Interrupt-driven I/O
: External events control the processor by requesting the current program be suspended and the external event be serviced. An external device will interrupt the processor (assert an interrupt control line into the processor), at which time the processor will suspend the current task (program) and begin executing an interrupt service routine. The service of an interrupt may involve transferring data from input to memory, or from memory to output.

Direct Memory Access (DMA)
: DMA allows data to be transferred from I/O devices to memory directly without the continuous involvement of the processor. DMA is used in high-speed systems, where the rate of data transfer is important. Not all processors support DMA.

DMA

DMA is a way of streamlining transfers of large blocks of data between two sections of memory, or between memory and an I/O device. Let's say you want to read in 100M from disk and store it in memory. You have two options.

One option is for the processor to read one byte at a time from the disk controller into a register and then store the contents of the register to the appropriate memory location. For each byte transferred, the processor must read an instruction, decode the instruction, read the data, read the next instruction, decode the instruction, and then store the data. Then the process starts over again for the next byte.

The second option in moving large amounts of data around the system is DMA. A special device, called a *DMA Controller* (*DMAC*), performs high-speed transfers between memory and I/O devices. Using DMA bypasses the processor by setting up a *channel* between the I/O device and the memory. Thus, data is read from the I/O device and written into memory without the need to execute code to perform the transfer on a byte-by-byte (or word-by-word) basis.

INTEL EXHIBIT 1008 Page 38
IPR2023-00787

In order for a DMA transfer to occur, the DMAC must have use of the address and data buses. There are several ways in which this could be implemented by the system designer. The most common approach (and probably the simplest) is to suspend the operation of the processor and for the processor to "release" its buses (the buses are tristate). This allows the DMAC to "take over" the buses for the short period required to perform the transfer. Processors that support DMA usually have a special control input that enables a DMAC (or some other processor) to request the buses.

There are four basic types of DMA:

Standard block transfer
: Accomplished by the DMA controller performing a sequence of memory transfers. The transfers involve a load operation from a source address followed by a store operation to a destination address. Standard block transfers are initiated under software control and are used for moving data structures from one region of memory to another.

Demand-mode transfers
: Similar to standard mode except that the transfer is controlled by an external device. Demand-mode transfers are used to move data between memory and I/O or vice versa. The I/O device requests and synchronizes the movement of data.

Fly-by transfer
: Provides high-speed data movement in the system. Instead of using multiple bus accesses as with conventional DMA transfers, fly-by transfers move data from source to destination in a single access. The data is not read into the DMAC before going to its destination. During a fly-by transfer, memory and I/O are given different bus control signals. For example, an I/O device is given a read request at the same time that memory is given a write request. Data moves from the I/O device straight into the memory device.

Data-chaining transfers
: Allow DMA transfers to be performed as specified by a linked-list in memory. Data chaining is started by specifying a pointer to a *descriptor* in memory. The descriptor is a table specifying byte count, source address, destination address, and a pointer to the next descriptor. The DMAC loads the relevant information about the transfer from this table and begins moving data. The transfer continues until the number of bytes transferred is equal to the entry in the byte-count field. On completion, the pointer to the next descriptor is loaded. This continues until a null pointer is found.

To illustrate the use of DMA, let's consider the example of a fly-by transfer of data from a hard-disk controller to memory. A DMA transfer begins by the processor configuring the DMAC for the transfer. This setup involves specifying the source, destination, and size of the data, as well as other parameters. The disk controller generates a request for service to the DMAC (not the processor). The DMAC then

INTEL EXHIBIT 1008 Page 39
IPR2023-00787

generates a HOLD or BR (bus request) to the processor. The processor completes the current instruction; places the address, control, and data buses in a high-impedance state (*floats*, tristates, or releases them); and responds to the DMAC with a HOLD-acknowledge or BG (bus granted) and enters a dormant state. Upon receiving a HOLD-acknowledge, the DMAC places the address of the memory location where the transfer to memory will begin onto the address bus and generates a WRITE to the memory while the disk controller places the data on the data bus. Hence, a direct memory access is accomplished from the disk controller to the memory.

In a similar fashion, transfers from memory to I/O devices are also possible. DMACs are capable of handling block transfers of data. The DMAC automatically increments the address on the address bus to point to each successive memory location as the I/O device generates (or receives) data. Once the transfer is complete, the buses are returned to the processor and it resumes normal operation.

Not all DMA controllers support all forms of DMA. Some DMA controllers simply read data from a source, hold it internally, and then store it to a destination. They perform the transfer in exactly the same way that a processor would. The advantage in using a DMA controller instead of a processor is that if the transfer were to be performed by the processor, each transfer would still have program fetches associated with it. Thus, even though the transfer takes place by sequential reads and writes, the DMA controller does not also have to fetch and execute code, thereby providing a faster transfer than a processor.

Support for DMA is normally not found in small microcontrollers. Some mid-range processors (16-bit, low-end 32-bit) may have DMA support. All high-end processors (32-bit and above) will have DMA support, and many include a DMA controller on-chip. Similarly, peripherals intended for small-scale computers will not provide DMA support, whereas peripherals intended for high-speed and powerful computers definitely *will* have DMA support.

Parallel and Distributed Computers

Some embedded applications require greater performance than is achievable from a single processor. For cost reasons, it may not be practical to implement a design with the latest superscalar RISC processor, or perhaps the application lends itself to distributed processing where the tasks are run across several communicating machines. It may make more sense to use a fleet of lower-cost processors, distributed throughout the installation. It is becoming increasingly common to see embedded systems implemented using parallel processors.

Introduction to parallel architectures

The traditional architecture for computers follows the conventional, Von Neumann serial architecture. Computers based on this form usually have a single, sequential

INTEL EXHIBIT 1008 Page 40
IPR2023-00787

processor. The main limitation of this form of computing architecture is that the conventional processor is able to execute only one instruction at a time. Algorithms that run on these machines must therefore be expressed as a sequential problem. A given task must be broken down into a series of sequential steps, each to be executed in order, one at a time.

Many problems that are computationally intensive are also highly parallel. An algorithm that is applied to a large data set characterizes these problems. Often the computation for each element in the data set is the same and is only loosely reliant on the results from computations on neighboring data. Thus, speed advantages may be gained from performing calculations in parallel for each element in the data set, rather than sequentially moving through the data set and computing each result in a serial manner. Machines with multitudes of processors working on a data structure in parallel often far outperform conventional computers in such applications.

The *grain* of the computer is defined as the number of processing elements within the machine. A *coarsely grained* machine has relatively few processors, whereas a *finely grained* machine may have tens of thousands of processing elements. Typically, the processing elements of a finely grained machine are much less powerful than those of a coarsely grained computer. The processing power is achieved through the brute-force approach of having such a large number of processing elements.

There are several different forms of parallel machine. Each architecture has its own advantages and limitations, and each has its share of supporters.

SIMD computers

Single-Instruction Multiple-Data (*SIMD*) computers are highly parallel machines, employing large arrays of simple processing elements. In an SIMD machine, each processing element has a small amount of local memory. The instructions executed by the SIMD computer are broadcast from a central instruction server to every processing element within the machine. In this way, each processor executes the same instruction as all other processing elements within the machine. Since each processor executes the instruction on its local data, all elements within the data structure are worked upon simultaneously.

The SIMD machine is generally used in conjunction with a conventional computer. An example of this was the Connection Machine (CM-1) by Thinking Machines Corporation that used either a VAX minicomputer or a Silicon Graphics or Sun workstation as the "host" computer. The CM-1 was a finely grained SIMD computer with up to 64K of processing elements that appeared as a block of 64K of "intelligent memory" to the host system. An application running on the host downloaded a data set into the processor array of the CM-1, each processor within the CM-1 acting as a single memory unit. The host then issued instructions to each processing element of the CM-1 simultaneously. After the computations were completed, the host then read back the result from the CM-1 as though it were conventional memory.

INTEL EXHIBIT 1008 Page 41
IPR2023-00787

The primary advantage of the SIMD machine is that simple and cheap processing elements are used to form the computer. Thus, significant computing power is available using inexpensive, off-the-shelf components. In addition, since each processor is executing the same instructions and therefore sharing a common instruction fetch, the architecture of the machine is somewhat simpler. Only one instruction store is required for the entire computer.

The use of multiple processing elements, each executing the same instructions in unison, is also the SIMD's main disadvantage. Many problems do not lend themselves to being broken down into a form suitable for executing on an SIMD computer. In addition, the data sets associated with a given problem may not match well with a given SIMD architecture. For example, an SIMD machine with 10k processing elements does not mesh well with a data set of 12k data elements.

MIMD computers

The other major form of parallel machine is the *Multiple-Instruction Multiple-Data* (*MIMD*) computer. These machines are typically coarsely grained collections of semi-autonomous processors, each with their own local memory and local programs. An algorithm being executed on an MIMD computer is typically broken up into a series of smaller sub-problems, each executed on a processor of the MIMD machine. By giving each processing element in the MIMD machine identical programs to execute, the MIMD machine may be treated as an SIMD computer. The grain of an MIMD computer is much less than that of an SIMD machine. MIMD computers tend to use a smaller number of very powerful processors, rather than a large number of less powerful ones.

MIMD computers can be of one of two types: *shared-memory MIMD* and *message-passing MIMD*. Shared-memory MIMD systems have an array of high-speed processors, each with local memory or cache, and each with access to a large, global memory (Figure 1-9). The global memory contains the data and programs to be executed by the machine. Also in this memory is a table of processes (or sub-programs) awaiting execution. Each processor will fetch a process and associated data into its local memory or cache and will run semi-autonomously of the other processors in the system. Process communication also takes place through the global memory.

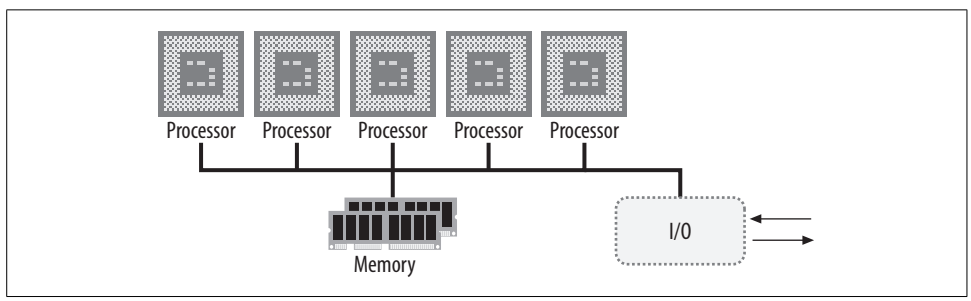


Figure 1-9. Shared-memory MIMD

INTEL EXHIBIT 1008 Page 42
IPR2023-00787

A speed advantage is gained by sharing the program among several, powerful processors. However, logic within the system must arbitrate between processors for access to the shared memory and associated shared buses of the system. In addition, allowances must be made for a processor attempting to access data in global memory that is out of date. If processor A reads a process and data structure into its local memory and subsequently modifies that data structure, processor B attempting to access the same data structure in main memory must be notified that a more recent version of the data structure exists. Such arbitration is implemented in processors like the (now extinct) Motorola MC88110, which was intended for use in shared-memory MIMD machines.

An alternative MIMD architecture is that of the message-passing MIMD computer (Figure 1-10). In this system, each processor has its own local, main memory. No global memory exists for the machine. Each processing element (processor with local memory) either loads, or has loaded into it, the programs (and associated data) that it is to execute. Each process runs autonomously on its local processor, and interprocess communication is achieved by message-passing through a common medium. The processors may communicate through a single, shared bus (such as Ethernet, CAN, or SCSI) or by using a more elaborate interprocessor connection architecture, such as 2-D arrays, N-dimensional hypercubes, rings, stars, trees, or fully interconnected systems.

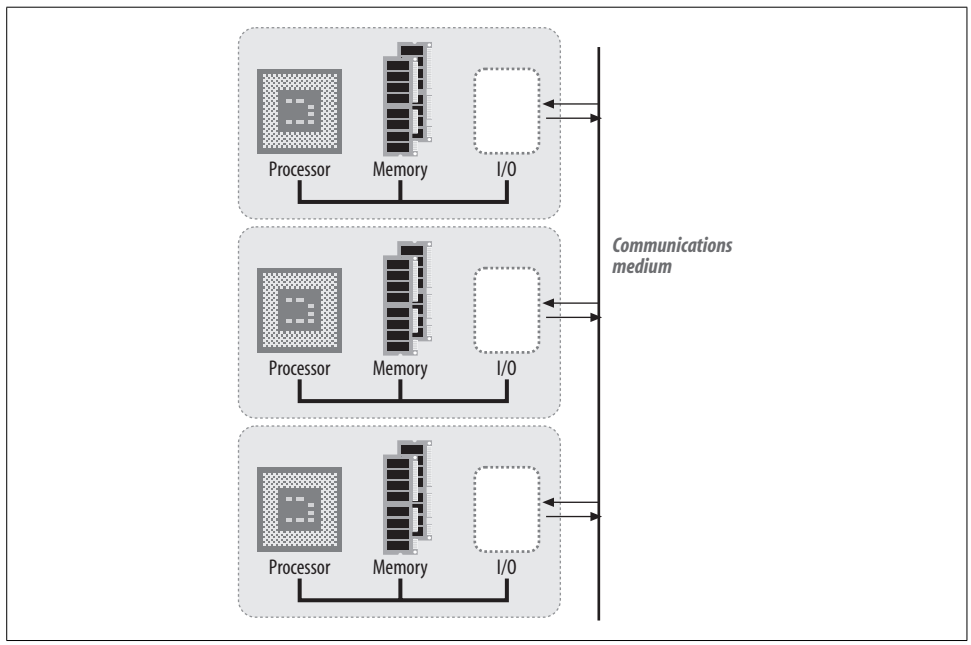


Figure 1-10. Message-passing MIMD

INTEL EXHIBIT 1008 Page 43
IPR2023-00787

Such machines do not suffer the bus-contention problems of shared-memory machines. However, the most effective and efficient means of interconnecting the processing nodes of a message-passing MIMD machine is still a major area of research. Each different architecture has its own merits, and which is best for a given application depends to a certain degree on what that application is. Problems that require only a limited amount of interprocess communication may work effectively on a machine without high interconnectivity, whereas other applications may weigh down the communications medium with their message passing. If a percentage of a processing node's time is spent in message-routing for its neighbors, a machine with a high degree of interprocess communication but a low degree of interconnectivity may spend most of its time dealing in message passing, with little time spent on actual computation.

The ideal interconnection architecture is that of the fully interconnected system, where every processing node has a direct communications link with every other processing node. However, this is not always practical, due to the costs and logistics of such a high degree of interconnectivity. A solution to this problem is to provide each processing element in the machine with a limited number of connections, based on the assumption that a processing element will not need or be able to communicate with every other processing element in the machine simultaneously. These limited connections from each processing node may then be interconnected using a *crossbar switch*, thereby providing full interconnectivity for the machine through only a limited number of links per node.

A *distributed* machine is composed of individual computers networked together as a loosely coupled MIMD parallel machine. Projects such as *Beowulf* and even *SETI@Home* can be considered MIMD machines. Distributed machines are common in the embedded world. A collection of small processing nodes may be distributed across a factory, providing local monitoring and control, and together forming a parallel machine executing the global control algorithm. The avionics of commercial and military aircraft are also distributed parallel computers.

Now let's take a look at computer applications and how they relate to the architecture of the machine.

Embedded Computer Architecture

What a computer is used for, what tasks it must perform, and how it interacts with humans and other systems determine the functionality of the machine and, therefore, its architecture, memory, and I/O.

An arbitrary desktop computer (not necessarily a PC) is shown in Figure 1-11. It has a large main memory to hold the operating system, applications, and data, and an interface to mass storage devices (disks and DVD/CD-ROMs). It has a variety of I/O devices for user input (keyboard, mouse, and audio), user output (display interface

INTEL EXHIBIT 1008 Page 44
IPR2023-00787

and audio), and connectivity (networking and peripherals). The fast processor requires a system manager to monitor its core temperature and supply voltages, and to generate a system reset.

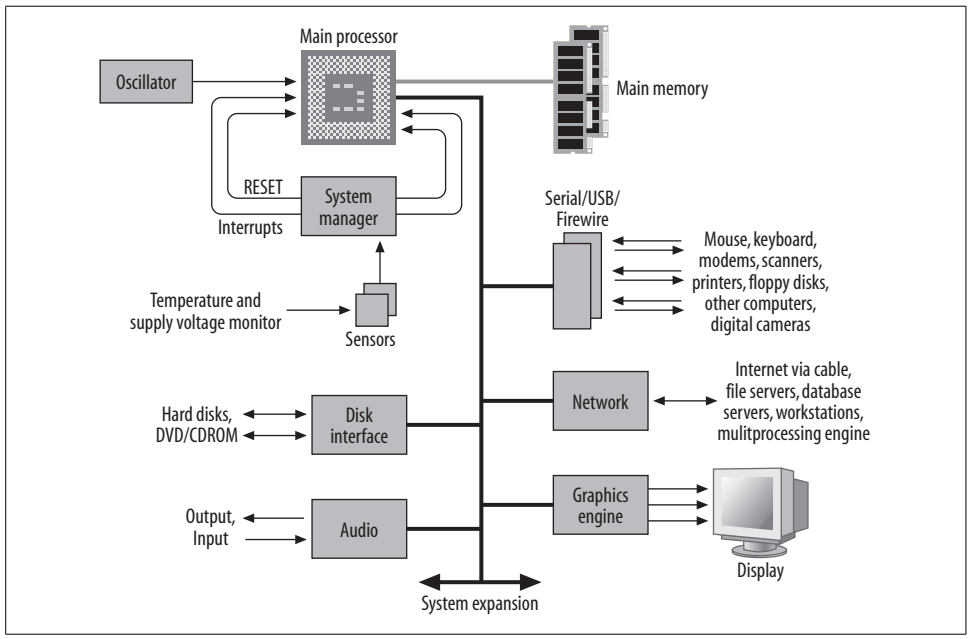


Figure 1-11. Block diagram of a generic computer

Large-scale embedded computers may also take the same form. For example, they may act as a network router or gateway, and so will require one or more network interfaces, large memory, and fast operation. They may also require some form of user interface as part of their embedded application and, in many ways, may simply be a conventional computer dedicated to a specific task. Thus, in terms of hardware, many high-performance embedded systems are not that much different from a conventional desktop machine.

Smaller embedded systems use microcontrollers as their processor, with the advantage that this processor will incorporate much of the computer's functionality on a single chip. An arbitrary embedded system, based on a generic microcontroller, is shown in Figure 1-12.

The microcontroller has, at a minimum, a CPU, a small amount of internal memory (ROM and/or RAM), and some form of I/O, which is implemented within a microcontroller as subsystem blocks. These subsystems provide the additional functionality for the processor and are common across many processors. The subsystems that you will typically find in microcontrollers will be discussed in the coming chapters.

INTEL EXHIBIT 1008 Page 45
IPR2023-00787

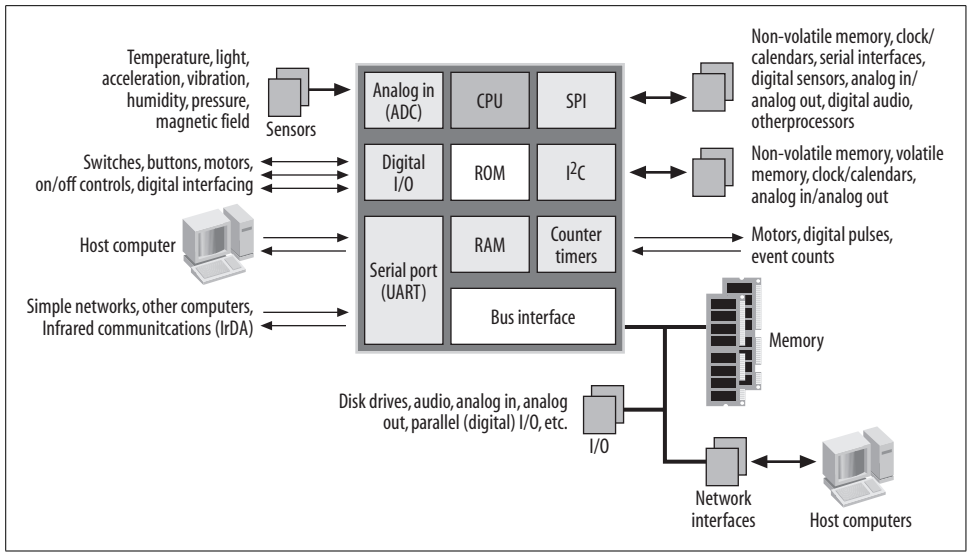


Figure 1-12. Block diagram of an embedded computer

For the moment, though, let's take a quick tour and examine the purposes for which they can be used.

The most common I/O is *digital I/O*, commonly called *general-purpose I/O*, or *GPIO*. These are ports that may be configured by software, on a pin-by-pin basis, as either a digital input or digital output. As digital inputs, they may be used to read the state of switches or push buttons, or to read the digital status of another device. As outputs, they may be used to turn external devices on or off, or to convey status to an external device. For example, a digital output may be used to activate the control circuitry for a motor, turn a light on or off, or perhaps activate some other device such as a water valve for a garden-watering system. Used in combination, the digital inputs and outputs may be used to synthesize an interface and protocol to another chip. Most microcontrollers have other subsystems besides digital I/O but provide the ability to convert the other subsystems to general-purpose digital I/O if the functionality of the other subsystems is not required. This gives you great versatility as a system designer in how you use your microcontroller within your application.

Many microcontrollers also have analog inputs, allowing sensors to be sampled for monitoring or recording purposes. Thus, an embedded computer may measure light levels, temperature, vibration or acceleration, air or water pressure, humidity, or magnetic field, to name just some. Alternatively, the analog inputs may be used to monitor simple voltages, perhaps to ensure the reliable operation of a larger system.

Some microcontrollers have serial ports, which enable the embedded computer to be interfaced to a host computer, a modem, another embedded system, or perhaps a simple network. Specialized forms of serial interface, such as SPI and I2C, provide a

INTEL EXHIBIT 1008 Page 46
IPR2023-00787

simple way of expanding the microcontroller's functionality. They allow peripherals to be interfaced to the microcontroller, providing access to such devices as off-chip memories (for data or parameter storage), clock/calendar chips (for timekeeping), sensors with digital interfaces, external analog input or output, and even audio chips and other processors. Most microcontrollers have timers and counters. These may be used to generate internal interrupts at regular intervals for multitasking, to generate external triggers for off-chip systems, or to provide control pulses for motors. Alternatively, they may be used to count external triggers (pulses) from another system. A few microcontrollers also include network interfaces, such as USB, Ethernet, or CAN. In this book, we'll look at many of these peripheral subsystems in detail and see how to utilize them to increase an embedded computer's functionality.

Some of the larger microcontrollers also provide a bus interface, bringing the internal address, data, and control buses to the outside world. This allows the processor to be interfaced to a huge variety of possible peripherals in very much the same way as a conventional processor. All of the possible devices and interfaces described previously may also be implemented through the bus interface and the appropriately chosen peripheral. A bus interface provides enormous possibility.

The mix of I/O subsystems that microcontrollers may have varies considerably. Some microcontrollers are intended for simple digital control and may have only digital I/O. Others may be intended for industrial applications, and may have digital I/O, analog input, motor control, and networking. The choice of microcontroller (and there are literally thousands of subspecies available from dozens of manufacturers) depends on your processing needs and your interfacing requirements. Choose the one that best suits your purposes.

INTEL EXHIBIT 1008 Page 47
IPR2023-00787

CHAPTER 2
Assembly Language

For the things we have to learn before we can do them,
we learn by doing them.
—Aristotle
Nichomachean Ethics

This chapter is about writing assembly-language software. This is a difficult subject to present, as it is a very diverse topic. There are many processors covered in this book. The assembly language of one processor bears no relation to that of another. To delve into the assembly language of each would require an entire book in its own right (or several books). Hence, I'm not even going to attempt to cover each one's instruction set. Instead, I'm going to concentrate on just two processors, the 68HC11 and the PIC, and use these as vehicles to show you some basic assembly-language techniques. I'm not going to dissect the instruction sets in detail. The User's Manual/datasheets for the processors give full descriptions of the instructions and their operation.

Assembly-language programming works down at the machine level, and as such, you really get a feel for what the processor is doing and how your computer actually works. While assembly-language programming can be fun, it can also be a daunting task to code major applications in it. To this end, assembly is usually used in only two instances. The first is for the development of simple test software during the early prototype development of a system. Such software may simply initialize the machine to a known state and perform some simple task such as flashing a LED or outputting the venerable "hello world" to a terminal or host.

With assembly language, you work with data in single bytes or words at a time. Processors store data within memory, and this is usually implemented using external chips in the computer system. To manipulate data, processors use registers within the CPU.

No computer can understand assembly directly. Back in the olden days, when computers were steam-driven and tended by gnomes, software was compiled manually.

INTEL EXHIBIT 1008 Page 48
IPR2023-00787

Each instruction mnemonic was looked up and converted to the appropriate opcode by the programmer. While it is certainly character-building, converting from assembly to opcodes is very tiresome, particularly where large programs are concerned. To make life easier, special compilers, called *assemblers*, take mnemonics and convert them to opcodes.

Assembly language has been described as the "nuts and bolts language," because you are writing code directly for the processor. For a lot of the software you will write, a high-level language like C will be the language of choice. High-level languages make developing software much easier, and your code is also portable (to a degree) between different target machines. Compilers of high-level languages convert your source code down to machine opcodes. Thus, by using a compiler, the programmer is relieved of having to know the specific details of the processor, and of having to code his program directly in machine code.

So there are good reasons for using a high-level language, yet programmers often write directly in assembly language. Why? Assembly and machine code, because they are "hand-written," can be finely tuned to get optimum performance out of the processor and computer hardware. This can be particularly important when dealing with time-critical operations with I/O devices. Further, coding directly in assembly can sometimes (but not always) result in a smaller code space. So, if you're trying to cram complex software into a small amount of memory and need that software to execute quickly and efficiently, assembly language may be your best (and only) choice. The drawback, of course, is that the software is harder to maintain and has zero portability to other processors. A good software engineer can create more efficient code than the average optimizing C compiler; however, a good optimizing compiler will probably produce tighter code than a mediocre assembly-language software engineer. Typically, you can include *in-line assembly* within your C code, and thereby get the best of both worlds.

At the mere mention of assembly language, many a die-hard programmer begins to quiver in fear, as if just invited into a tiger's cage. However, assembly-language programming is not that hard and can often be a lot of fun. Think of it as being "as one" with the processor.

That said, this is a book predominantly about hardware, not software. In this chapter, we'll take a quick tour of assembly-language programming, just to give you an overview. In the next chapter, we'll take a look at the Forth programming language, widely used in many embedded systems. For more information on embedded software development, I highly recommend two O'Reilly books: *Programming Embedded Systems* by Michael Barr and Anthony Massa, and *Programming with GNU Software* by Mike Loukides and Andy Oram. If you want to build a Linux-based embedded computer, I recommend *Building Embedded Linux Systems* by Karim Yaghmour, also published by O'Reilly.

Let's start by looking at the internal storage of the processor.

INTEL EXHIBIT 1008 Page 49
IPR2023-00787

Registers

Registers are the internal (working) storage for the processor. The number of registers varies significantly among processor architectures. Typically, the processor will have one or more *accumulators*. These are registers that may have arithmetic operations performed on them. In some architectures, all the registers function as accumulators, whereas in others, some registers are dedicated for storage only and have limited functionality.

Some processors have *index registers* that can function as pointers into the memory space. In some architectures, all general-purpose registers can act as index registers; in others, dedicated index registers exist.

All processors have a *program counter* (also known as an *instruction pointer*) that tracks the location in memory of the next instruction to be fetched and executed. All processors have a *status register* (also known as a *condition-code register*, or *CCR*) that consists of various status bits (flags) that reflect the current operational state. Such flags might indicate whether the result of the last operation was zero or negative, whether a carry occurred, if an interrupt is being serviced, etc.

Some processors also have one or more control registers consisting of configuration bits that affect processor operation and the operating modes of various internal subsystems. Many peripherals also have registers that control their operation and registers that contain the results of operations. These peripheral registers are normally mapped into the address space of the processor.

Some processors have banks of *shadow registers*, which save the state of the main registers when the processor begins servicing an interrupt. By using shadow registers, there is no need to save processor state to the stack. This can save significant time in servicing an interrupt. (We'll look at stack operations in more detail shortly.)

Processors are commonly 8-bit, 16-bit, 32-bit, or 64-bit, referring to the width of their registers. An 8-bit processor is invariably low-cost and is suitable for relatively simple control and monitoring applications. If more processing power is required, the larger processors are preferable, although cost and system complexity increase accordingly.

Machine Code

Everything that a processor deals with is expressed as numbers in memory. That applies to data, and to software as well. Your compiled C program is converted to a sequence of numbers that is meaningful to the processor as a sequence of instructions. A program that a processor (in this case, a 68HC11) can execute might look something like:

```
86 41 B7 01 00 BD 02 00
```


INTEL EXHIBIT 1008 Page 50
IPR2023-00787

This is known as *machine code*, and each numeric instruction is called an *opcode*. Humans find such programs very hard to write and even harder to understand. To make this easier to cope with, we use a notation called *assembly language*. Assembly-language instructions equate directly to their machine code counterparts. Since machine code is difficult to read and write (for a human), a *mnemonic* is used to represent the opcode. For example, the 68HC11 instruction `86 41` is more easily understood by its mnemonic `LDAA #$41`. (The dollar sign [$] represents a hex value in some assembly languages.) This is still a bit cryptic, so we usually add comments on the righthand side to help us follow what is going on. The previous machine code sequence written as 68HC11 assembly would be:

```
Machine code   Assembly    Comment
86 41          LDAA #$41   ; anything after a semi-colon
B7 01 00       STAA $0100  ; is a comment
BD 02 00       JSR $0200   ;
```

`LDAA #$41` means "load accumulator A with the immediate value (signified by the #) of `0x41`." *Immediate* means that the operand is to be treated as a number, rather than as a pointer to an address. With some processors, this is known as a *literal*. `LDAA` corresponds to the machine opcode `86`.

`STAA $0100` tells the processor to store the contents of accumulator A to the *memory location* (no # this time) `0x0100`.

`JSR $0200` means "jump to the subroutine at address `0x0200`." A subroutine is essentially a procedure or function call. In some processors, the instruction to do this is call, rather than jump.

The instructions of a Freescale Semiconductor 68HC11 series processor (Chapter 16) can be from one to three bytes in length. The first byte is always an opcode and may be followed by one or two bytes of data. For example, consider the following assembly-language program:

```
Machine code   Assembly    Comment
CE 10 02       LDX #$1002  ; load index register X with 0x1002
5F             CLRB        ; clear accumulator B
86 41          LDAA  #$41  ; load accumulator A with 0x41.
```

Note the length of each instruction. The first instruction takes three bytes, the second takes only one, and the third is two bytes in length. The number of bytes an instruction takes depends on what it does. This is true for CISC processors, but not RISC. In order to achieve simpler instruction decode units, RISC processors have instructions of a fixed size, with operands buried inside the instruction. For example, all instructions of a PIC16 processor are 14 bits wide, regardless of what operation they perform.

Different processors use different assemblies. No two are alike. The previous examples are written in 68HC11 assembly. This assembly is applicable only to the 6800 family of processors, of which the 68HC11 is a member. Other assembly languages, because they are based on very different processor hardware, have very different

INTEL EXHIBIT 1008 Page 51
IPR2023-00787

syntax. Some examples of different versions of the same operation are given in Table 2-1. In each case, a register is loaded with a byte value of 0x41. (Note also the different ways of expressing hex notation.)

Table 2-1. Comparison of some different assembly languages

Processor	Instruction	
Motorola 6800/68HC11	LDAA	#$41
Intel x86	mov	al,41h
Motorola 680x0	move.b	#$41,D0
PIC16xx	movlw	0x41
Motorola 56000	move	#$0041,A
Intel 80960	lda	0x41,r4

Signed Numbers

Before getting into the nuts and bolts of assembly, let's take a quick look at how signed numbers are represented within the machine. Negative and positive numbers can be represented in binary (or hex) number systems by the state of the most significant bit. If the most significant bit is set, this indicates that the number is negative.



It is up to the programmer whether to treat the content of a register or memory location as signed or unsigned. In C, we are used to explicitly declaring a variable or pointer as either `signed` or `unsigned`. In assembly, it's up to you.

An 8-bit accumulator can have (in decimal) 0 to 255 (unsigned) or -128 to +127 (signed). Given a positive number, we use *two's complement* to convert this to its negative equivalent. Taking the two's complement of a number is done by inverting the bits (a *one's compliment*) and adding one to the result. So 0xFF is 1111 1111 (in binary) and is therefore negative. 0xFF is used to represent -1, 0xFE is -2, etc., all the way to 0x80, which is -128 (decimal).

For example, -3 as an 8-bit hex value is calculated in this way:

```
              0 0 0 0   0 0 1 1  (0x03)
Invert bits: 1 1 1 1   1 1 0 0  (0xFC)
Add 1 gives: 1 1 1 1   1 1 0 1  (0xFD)
```

So -3 is 0xFD in hex.

Note that we are not limited to using signed (positive and negative) numbers. We are free to ignore the sign bit and treat the byte as an unsigned number, giving us a range of 0x00 to 0xFF (255 decimal). It all depends on what we want the byte to represent.

16-bit, 32-bit, and 64-bit signed numbers work in the same way. The most significant bit represents the sign, and the remaining bits constitute the number.

INTEL EXHIBIT 1008 Page 52
IPR2023-00787

Addressing Modes

The different ways in which an instruction can reference a register or memory location are known as the *addressing modes* of the processor. The types of addressing modes available within different architectures vary. To illustrate some of the more common modes, let's take a look at those available in the 68HC11 architecture:

Inherent
: The instruction deals purely with registers. `CLRB` clears the B accumulator, for example.

Immediate/Literal
: The instruction has a literal number as an operand.

Direct
: The instruction accesses a memory location, specified by a short address. In other words, direct addressing provides access to a subset of the total address space. On a given processor with a 16-bit address bus, a direct access may, for example, specify an address within the first 256 bytes. On a 32-bit processor, a direct access may specify an address within the first 64K of memory, for example. Direct addressing is used (when possible) to reduce the length of instruction-referencing memory. This can reduce code size, and therefore instruction fetch time, in time-critical applications. Many processors, especially RISC, do not use direct addressing.

Extended
: The instruction accesses a memory location, specified by the full address. So `LDAA $B098` means load accumulator A with the contents of address `0xB098` in memory.

Indexed
: The instruction uses the contents of a register as a pointer into memory. If, for example, the X register is equal to `0x5034`, then `LDAA 0,X` means "load accumulator A with the contents of the location pointed to by X." In this example, this means "load accumulator A with the contents of address `0x5034`." Whereas `LDAA 2,X` means "load accumulator A with the contents of the memory location pointed to by X, but with an offset of 2." So if X = `0x5034`, then `2,X` is `0x5036`. So `LDAA 2,X` loads from address `0x5036`. Indexed addressing is useful when your program needs to reference data in a table or list. The index is an easy way of moving through the data structure.

Relative
: An offset is specified as part of the addressing. A branch instruction uses relative addressing to add (or subtract) a value from the program counter. For example, `bra 02` (branch) adds 2 to the program counter, whereas `bra $FF` adds –1 to the program counter. Note that the instruction counter always points to the next instruction, not the one currently being executed.

INTEL EXHIBIT 1008 Page 53
IPR2023-00787

Big-Endian and Little-Endian Addressing

Microprocessors are either *big endian* or *little endian* in their architecture. This refers to the way in which the processor stores data (16 bits or greater) to memory. A big-endian processor stores the most significant byte at the least significant address, as illustrated in Figure 2-1. In each case, the data has been stored to address `0x0100`.

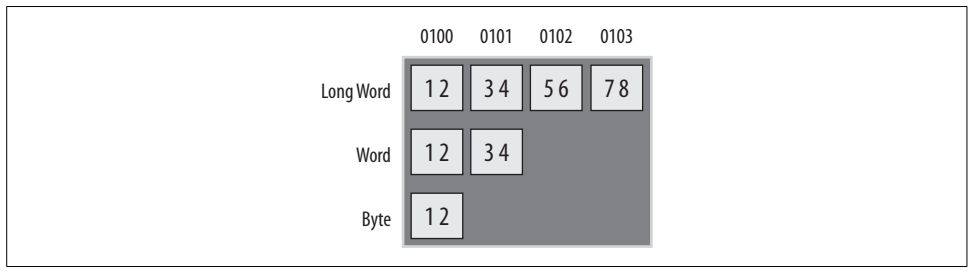


Figure 2-1. Big endian

A little-endian processor stores the most significant byte at the most significant address, as shown in Figure 2-2.

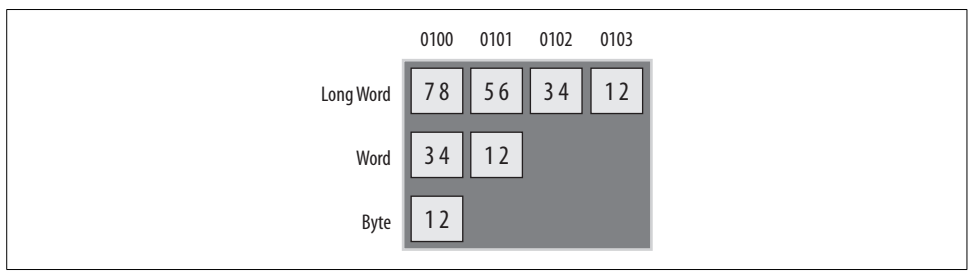


Figure 2-2. Little endian

With the little-endian scheme, the least significant data travels over the least significant part of the data bus and is stored at the least significant memory location. For a programmer, it is conceptually easier to understand in terms of data path. The inconvenience of little endian is that data appears "backwards" in the computer's memory if you display a block of locations. Storing the value `0x12345678` to memory results in `0x78563412` in the memory space. Note that a little-endian processor will read this data back correctly; it's just that it makes it harder to understand the numbers if a human is looking at the memory directly. Alternatively, a big-endian processor storing `0x12345678` to memory results in `0x12345678` sitting inside the memory chip. This appears (to a human) to make more sense. Neither scheme has much advantage over the other in terms of operation; they are just two different ways of doing the same thing. When you're doing high-level programming on a system, the "endian-ness" makes little difference. The only time you are really exposed to it is when you are

INTEL EXHIBIT 1008 Page 54
IPR2023-00787

dealing with multibyte data and accessing it on a byte-by-byte basis. However, when you are developing and debugging hardware and low-level firmware, you come across it all the time, and so an understanding of big endian and little endian is important.

Coding in Assembly

Assembly-language programming is not difficult, particularly with the smaller microcontroller and microprocessor architectures. The main difficulty many people seem to have is knowing how to tackle a given programming problem. As with any other language, there are always many ways in which a program may be written. There is rarely a single "right way" (although there may be a "most efficient way"). It is simply a matter of breaking down the problem into a series of tiny steps.

For example, let's say we want a 68HC11 processor to add three numbers together (`0x10`, `0x1F`, and `0x0C`) and store the result at address `0x0027`. This problem is easily broken down into four steps. We take the first number (step one), add the second number (step two), add the third number (step three), and store the result (step four). We want to do some arithmetic, so we choose an accumulator to hold our numbers, since accumulators are registers designed for this type of operation. We start by loading an accumulator with the first number:

```
LDAA   #$10   ; load first number into the A accumulator
```

The A accumulator now contains the number `0x10` (16 in decimal). Remember that the "#" in the instruction means that the addressing mode is immediate. In other words, the processor is to load the actual number `0x10`, not use the operand as a pointer. We then want to add the number `0x1F`. So our next instruction is:

```
ADDA   #$1F   ; add the second number to the A accumulator
```

This causes `0x1F` to be added to accumulator A. Again note the "#." Similarly, to add the third number, we have:

```
ADDA   #$0C   ; add the third number to the A accumulator
```

Finally, we store the result at address `0x0027`:

```
STAA   $0027  ; store the contents of the A accumulator to address 0x0027
```

This time, note that there is no "#." This is an extended instruction. It means "Store the contents of accumulator A to the memory location `0x0027`." It does not treat `0x0027` as a number, but rather as the address of a location in memory.

So our complete program is:

```
LDAA   #$10   ; load first number in the A accumulator
ADDA   #$1F   ; add the second number to the A accumulator
ADDA   #$0C   ; add the third number to the A accumulator
STAA   $0027  ; store the contents of the A accumulator to address 0x0027
```


INTEL EXHIBIT 1008 Page 55
IPR2023-00787

To make this program into a subroutine (a callable function or procedure), we simply add a label at the start and a return-from-subroutine (RTS) instruction at the end. (We'll look at subroutines in more detail shortly.) A *label* is used by the assembler during compilation to identify a given address location. It has no direct meaning in the machine code:

```
add_numbers
   LDAA   #$10   ; load first number in the A accumulator
   ADDA   #$1F   ; add the second number to A
   ADDA   #$0C   ; add the third number to A
   STAA   $0027  ; store the contents of A to address 0x0027
   RTS           ; return to main program
```

To call our subroutine from our main program, we simply use a jump-to-subroutine instruction (JSR) with the label as an operand:

```
JSR    add_numbers
```

For comparison, this is what the same subroutine looks like in PIC16 assembly:

```
add_numbers
   movlw  0x10   ; load first number in the w accumulator
   addlw  0x1F   ; add the second number to the w accumulator
   addlw  0x0C   ; add the third number to the w accumulator
   movwf  0x27   ; store the content of w to address 0x27
   return        ; return to main program
```

movlw is "move literal to w," and, similarly, addlw is "add literal to w." movwf is "move w to register file (f)." (Microchip calls the processor's on-chip RAM "registers.") The PIC's return instruction is exactly the same as the 68HC11's RTS. Both perform exactly the same function; just the mnemonic is different.

To call our PIC subroutine, we use the PIC's call instruction with the label as an operand:

```
call   add_numbers
```

The program can be converted to machine code by looking up each instruction in the datasheet or by using an assembler running on a host computer. Let's manually assemble the 68HC11 subroutine so that you can see the process. (Opcodes are always assumed to be hex and carry no prefix.)

LDAA (immediate) is 86 and takes two bytes, the other byte being the data operand, 0x10. Similarly, ADDA (immediate) is 8B and also takes two bytes. Looking up the remaining opcodes in the datasheet or programmer's reference gives us:

```
0100  86 10    LDAA #$10  ; load first number in the A accumulator
0102  8B 1F    ADDA #$1F  ; add the second number to the A accumulator
0104  8B 0C    ADDA #$0C  ; add the third number to the A accumulator
0106  B7 00 27 STAA $0027 ; store the contents of A to address 0x0027
0109  39       RTS        ; return to main program
```


INTEL EXHIBIT 1008 Page 56
IPR2023-00787

Here I have (arbitrarily) assumed that the program will start at address 0x0100 and have added the address of each instruction at the far left. Note how the addresses change in relation to the size of the instructions.

If I want to explicitly tell an assembler to start the code at address 0x0100, I could use the org *assembler directive*:

```
org  $0100
```

The org directive always precedes the first instruction at that address. org is useful for locating code at specific locations, such as on-chip ROM.

Let's now see how this program is executed by the processor by manually stepping through the code. Assuming that the program counter points to address 0x0100, the first step taken by the processor is loading the byte located at address 0x0100 and decoding it as an instruction. The processor does this by placing 0x0100 on the address bus and performing a read cycle. The memory device responds by placing the content of address 0x0100 (86) onto the data bus. This is then read by the processor into its instruction decode unit. 86 is a LDAA (immediate) instruction, so this causes the processor to load the next byte of the instruction (at address 0x0101) and place this in the A accumulator. As part of the instruction decode in the processor, the program counter is incremented by the size of the current instruction so that it points to the next instruction. Hence, the program counter is now 0x0102, as the first instruction occupied two bytes.

The processor loads the instruction at 0x0102 by placing this address on the address bus and performing a read cycle. The processor decodes it to be an ADDA (immediate). The processor then fetches the next byte and adds this to accumulator A. Similarly, it executes the next ADDA instruction and adds the byte at 0x0105 to the A accumulator.

The program counter now points to address 0x0106. The processor fetches this instruction by placing 0x0106 on the address bus and performing a read cycle. The memory device responds by placing B7 on the data bus. The processor decodes B7 to be an extended store operation. It then loads the next *two* bytes by placing 0x0107 on the address bus and performing a read, then placing 0x0108 on the address bus and performing another read. The processor takes the two bytes that were read and constructs the address 0x0027. The processor then places 0x0027 on the address bus, places the contents of accumulator A on the data bus, and performs a write cycle. The memory chip or peripheral mapped by the address decoder to address 0x0027 latches and stores the written data.

Normally, the program counter is incremented as each instruction is executed, so that it points to the next instruction. Some instructions, such as jump (JMP) or branch (BRA) directly modify the program counter, thus giving you control over the flow of the program. These instructions effectively reload the program counter with a new

INTEL EXHIBIT 1008 Page 57
IPR2023-00787

value (as with jump) or add to the program counter (branch), so that it points to a different location in memory. This means that the next instruction to be executed will come from a new address, not the address immediately following the current instruction.

Disassembly

Disassembly is the conversion from a sequence of machine code back to the mnemonics that represent that code. This is done when we have a machine code program (perhaps written by someone else) and we want to know what it does and how it works.

For example, suppose we have the following sequence of bytes that constitute a 68HC11 machine code program:

```
8E 56 78 86 56 84 06 36 4C 36
```

We start by assuming that the first byte is an opcode. By looking up the opcode 0x8E in the Motorola 6800 (or 68HC11) datasheet, you will find that it is the LDS instruction (load stack pointer) and that it takes three bytes (one for opcode, two for data). Therefore, if the first byte is the instruction, the next two bytes are its associated data. So that gives us the first instruction:

```
8E 56 78  LDS #$5678 ; load stack pointer with the number 0x5678
```

If the first instruction was three bytes long, then the fourth byte in the sequence must be the second instruction. Therefore, the next opcode is 0x86, which, according to the datasheet, is LDAA # (load accumulator A with an immediate value) and takes two bytes.

So we now have:

```
8E 56 78  LDS  #$5678 ; load stack pointer with the number 0x5678
86 56     LDAA #$56   ; load Acc A with 0x56
```

The next opcode is 0x84, which is an ANDA # instruction taking two bytes. Then we have a 0x36 (PSHA), a 0x4C (INCA), and finally another 0x36 (PSHA).

So, from:

```
8E 56 78 86 56 84 06 36 4C 36
```

our disassembled program is:

```
8E 56 78  LDS  #$5678 ; load stack pointer with the number 0x5678
86 56     LDAA #$56   ; load Acc A with 0x56
84 06     ANDA #$06   ; AND Acc A with 0x06
36        PSHA        ; push Acc A onto the stack
4C        INCA        ; increment Acc A
36        PSHA        ; push Acc A onto the stack
```


INTEL EXHIBIT 1008 Page 58
IPR2023-00787

Position-Independent Code

When we jump to an address or subroutine within our program, we could use an absolute address. For example, `JSR $0200` jumps to a subroutine at the absolute address `0x0200`. This means that for our program to work, that subroutine must always be at address `0x0200`. A program (and hence its subroutines) could be located (or relocated) by the computer's operating system to anywhere in memory. If this were to happen, `JSR $0200` would no longer jump to the location of our subroutine because that subroutine would no longer be at address `0x0200`. Therefore the program would crash. For this reason, it is good practice to employ *position-independent code*. This method of programming avoids the use of absolute addresses (except where appropriate). It means that rather than jumping to the absolute address of our subroutine, we branch to the subroutine relative to the program counter's contents. It sounds complicated, but all it really means is that we should use `BRA` (or `BSR`) rather than `JMP` (or `JSR`). By using a branch rather than a jump, we are adding a number to the program counter rather than replacing its contents. This means that we are not jumping to an absolute position in memory, but branching relative to our current location.

Absolute addressing is used when we branch or jump to some code that we know will always be at a given address. For example, a subroutine that is located in ROM (permanent memory) is not going to move somewhere else.

Loops

It is often useful to repeat an instruction or series of instructions. Using branch instructions allows us to redirect the program counter and hence control the flow of the program. Here is a simple example of a loop that executes 15 times:

```
        LDAA #$0F  ; load the count into the A accumulator
again           ;
        DECA       ; count down (decrement the A accumulator)
        BNE again  ; repeat until we have counted down to zero
```

`DECA` decrements the A accumulator and sets the status flags as appropriate. If the result of the decrement was a zero in the A accumulator, the zero flag in the CCR (Condition Code Register) is set; otherwise, the flag is cleared. The `BNE` instruction is "branch not equal (to zero)." This checks the state of the zero flag. If it is clear (not equal to zero), then the branch is taken; otherwise, execution continues on with the next instruction after `BNE`. Hence, the above code fragment will start with the A accumulator equal to 15 and will decrement this value until is has reached zero. At this point, the loop will terminate. Now, to get the loop to actually perform work for you, place your code between the `again` label and the `DECA` instruction.

INTEL EXHIBIT 1008 Page 59
IPR2023-00787

In machine code, this is:

```
Address Opcodes       Assembly
0100    86 0F         LDAA #$0F
0102    4A      again DECA
0103    26 FD         BNE  again
0105    ...           ; address of instruction after BNE
```

Note the instruction BNE. Where did the 0xFD come from? We want the processor to branch back to address 0x0102. As with all instructions, when executing the BNE the program counter points to the next instruction, so as BNE is being executed, the program counter contains 0x0105. We want the program counter to be 0x0102, so we need to subtract 3 from it. We do this by adding –3, which in 8-bit hex is 0xFD. (0xFF is -1, 0xFE is -2, 0xFD is -3, and so on.) To branch forward, we add a positive number. To branch backward, we add a negative number. In both cases, we must remember to take into account that the program counter will be pointing to the address *after* the branch when determining the value to add.

Masking

It is often useful to examine the state of certain bits (such as the status of a peripheral chip or a flag). We are interested in the state of a given bit, but the state of the other bits is unknown (and unimportant to us).

For example, assume we want to check the state of bit 0 in a byte that has been read from a status register. To do this, we need to compare the byte with a constant. But what constant should we choose? In both the following examples of a byte read from a status register, bit 0 is set. Comparing the byte with the constant 0x65 will work in the first instance but not the second:

```
0 1 1 0 0 1 0 1 (0x65)
1 1 0 0 1 1 0 1 (0xCD)
```

There is no single number with which we can compare the byte that will work in all instances. We need to set all the other bits to a known state before we can do a comparison. To do this, it is necessary to *mask* out any bits in which we are not interested. This may be accomplished by using an AND instruction.

In the previous example, we need to AND the byte from the status register with 0x01 to clear the other bits. This will preserve bit 0 (regardless of its state) and clear all other bits. The result of this operation is 0x01 if bit 0 was set or 0x00 if bit 0 was clear:

```
0 1 1 0 0 1 0 1   (0x65)
0 0 0 0 0 0 0 1   (0x01)  AND
```

This gives:

```
0 0 0 0 0 0 0 1   (0x01)
```


INTEL EXHIBIT 1008 Page 60
IPR2023-00787

The following operation:

```
1 1 0 0 1 1 0 1   (0xCD)
0 0 0 0 0 0 0 1   (0x01)  AND
```

gives:

```
0 0 0 0 0 0 0 1   (0x01)
```

Finally, this operation:

```
1 0 0 0 1 1 0 0   (0x8C)
0 0 0 0 0 0 0 1   (0x01)  AND
```

gives:

```
0 0 0 0 0 0 0 0   (0x00)
```

In each of these examples, the state of bit 0 in the original byte is preserved. All other bits are cleared. Therefore, to determine the state of bit 0, we need now only compare the byte with `0x01`. If it is equal, the bit is set; otherwise, the bit is clear.

So let's convert this into assembly. Here I am assuming that we want to check the status of an I/O device located at address `0x0700`:

```
check   LDAB $0700 ; load status register located at 0x0700 into B
        ANDB #$01  ; mask out unwanted bits
        CMPB #$01  ; check to see if bit 0 is set
        BNE  check ; if it is not, go back and check once more
```

This code fragment loops until bit 0 is found to be set.

Indexed Addressing

It is often useful to use a pointer to reference a section of memory. The 68HC11 has two index registers, X and Y. These are the equivalent of pointers in C.

For example, let's say we want to fill the address range `0x0200` to `0x2FF` with the number `0x60`. One way would be to load an accumulator with `0x60` and store that to address `0x0200`. Then we could store the value to address `0x0201`, and so on:

```
LDAA #$60    ;
STAA $0200   ;
STAA $0201   ;
STAA $0202   ;
STAA $0203   ;
STAA $0204   ;
STAA $0205   ;
STAA $0206   ; ... and so it goes
```

While this would do what we want, it makes for a long (and tedious) program. A simpler way is to use indexed addressing, with an index register pointing to each

INTEL EXHIBIT 1008 Page 61
IPR2023-00787

memory location in turn. The following program uses indexed addressing to achieve our goal:

```
        LDX  #$0200  ; load the X register with the number 0x0200
        LDAA #$60    ; load the A accumulator with the value to be stored
        LDAB #$FF    ; load the B accumulator with the count
loop    STAA 0,X     ; store acc A to address pointed to by X with no offset
        INX          ; increment X to point to next address
        DECB         ; count down
        BNE  loop    ; repeat until we have counted down to zero
```

The LDX instruction loads an immediate value into the 16-bit X register. This is now our pointer into memory. The A accumulator is then loaded with the value to be stored in memory, and the B accumulator is loaded with the number of locations in memory we will be accessing. This will act as the counter for our loop.

The loop begins by storing the content of the A accumulator to the memory location pointed to by the X register. The 0,X means there is no offset. (If we wanted to store the value not to the address pointed to by X, but the one three locations further on, we would use 3,X.) The X register is then incremented to point to the next address in memory, and the B accumulator is decremented since we have completed an iteration of the loop. In the process of decrementing the accumulator, the zero flag will be set or cleared, and this will be checked by the subsequent branch instruction. If the result of the decrement is not zero, then the loop isn't finished. Execution branches back to the STAA 0,X instruction, and the content of A is stored to the next location in memory. The process repeats.

Stacks

Many processors implement one or more *stacks*, which serve as temporary storage in external memory. The processor can *push* a value from a register on the stack to preserve it for later use. The processor retrieves this value by *popping* from the stack back into a register. In some processor architectures, popping is also known as *pulling*.

Most processors have a special register known as a *stack pointer*, which references the next free location on the stack. Some processors implement more than one stack and so have more than one stack pointer. Most stacks grow down through memory. (Some processors have stacks that grow up as the stack is filled.) When the processor pushes or pops a value to or from the stack, the stack pointer automatically decrements or increments to point to the next free location.

Figure 2-3 shows the steps that occur when the content of a register (in this case, 72) is pushed onto the stack. These steps are normally performed by a single instruction (for example, PSHA). First, the value is copied from the register and stored to the location pointed to by the stack pointer. The stack pointer is then decremented. Again, these operations take place automatically by executing a single push instruction.

INTEL EXHIBIT 1008 Page 62
IPR2023-00787

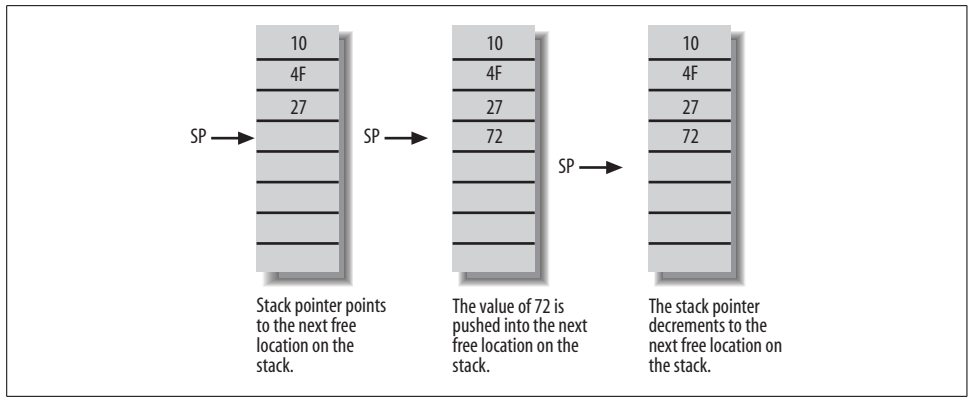


Figure 2-3. Pushing a register onto the stack

Most processors automatically save registers to the stack when events such as interrupts occur. However, the stack is also available to you as a programmer to temporarily store values. Some processors have two stacks, with one reserved for system use and the other available to the application programmer.

Stacks are particularly useful for the temporary storage of data. For example, let's say we wish to call a subroutine, but we know that the subroutine changes the contents of the B accumulator as part of its operation. If there was nothing of importance in the B accumulator before we called the subroutine, this would be of little consequence. But if we wished to preserve the B accumulator, we would push the accumulator onto the stack before we called the subroutine and then pull it back after the subroutine ends:

```
PSHB       ; save B accumulator onto the stack
BSR   fred ; branch to subroutine fred
PULB       ; restore B accumulator from the stack
```

In this way, we have used the stack as a temporary storage location for data that we required later in the program. Alternatively, a good programmer would write the subroutine to preserve any affected registers:

```
fred
      PSHB  ; save B accumulator onto the stack
            ;
            ; (subroutine code here)
            ;
      PULB  ; restore B accumulator from the stack
      RTS   ;
```

Timing of Instructions

On some processors, particularly CISC chips, different instructions take varying lengths of time to complete. Sometimes it is important to know how long a given

INTEL EXHIBIT 1008 Page 63
IPR2023-00787

section of code will take to execute, particularly if the software is interacting with a time-critical external system. The execution time can be calculated by looking up the number of cycles each instruction takes and adding them together.



A "cycle" relates to the internal clock that is driving the processor. This is not necessarily the same speed as the crystal. The datasheet will tell you how the crystal frequency relates to the internal clock for a given processor.

The following example shows some 68HC11 assembly language and the number of cycles associated with each instruction:

```
      LDAA #$0F ; 2 cycles
again DECA      ; 2 cycles
      BNE again ; 3 cycles
exit  JMP $1000 ; 3 cycles
```

From examining the code, we can see that the loop is executed 15 (0x0F) times. This loop comprises five cycles in total. Therefore, the above program will take:

```
2 + (15 * 5) + 3 cycles = 80 cycles
```

For a 2 MHz 68HC11, a cycle is 500 ns. Hence, this code fragment will take 40,000 ns (or 40 μsecs) to complete.

Knowing the timing of instructions is often very useful. For example, suppose we need to have a program that will provide a delay of approximately 9 msec. To do this, we need to load a value into a register and count it down to zero. This will provide the delay.

So a program that would do this is simply:

```
      LDX #some_number ; load X with a number
repeat DECX            ; count down            3 cycles
      BNE repeat       ; not zero, repeat      3 cycles
```

For a 2 MHz 68HC11, a cycle is 500 ns. So for 9 msec, we require 18,000 cycles. The loop takes six cycles per iteration. Therefore, we need to run through the loop 3,000 times. In hex, 3,000 is 0x0BB8. So our program becomes:

```
      LDX #$0BB8   ; load X with 3000 decimal
repeat DECX        ; count down         3 cycles
      BNE repeat   ; not zero, repeat   3 cycles
```

Now, let me make some important points here. If an interrupt occurs during the loop, the processor will service the interrupt, and this will add an unknown (and unknowable) delay to the execution. If this code was controlling something critical, such an event would be disastrous. Most processors have instructions to disable and re-enable interrupts, and you would be well advised to turn off all interrupts prior to this code executing. Of course, you have to remember to re-enable them afterward as well. The other important point is that this code will provide a delay only of 9 msec,

INTEL EXHIBIT 1008 Page 64
IPR2023-00787

provided it is running on a 68HC11 at 2 MHz. If the code were run on a computer with a processor kicking along at 2.1 MHz, we would not get our 9 msec delay. In other words, techniques like this are not very portable and are generally considered bad practice. Use it only if you really have to. Programmers using code such as this often get caught when they move their code to a newer version of the hardware. Code that used to work is suddenly broken.

A better way to provide a fixed delay is to use a hardware timer, either as an internal peripheral to the microcontroller (and most have them) or as an external device. The timer generates an interrupt to the processor, and this gives a much more precise, reliable, and portable way of implementing a timing delay.

INTEL EXHIBIT 1008 Page 65
IPR2023-00787

CHAPTER 6

Building Hardware

Hardware: the part of a computer that you can kick.
—Anon.

Before we get into the designs later in this book, let's spend some time looking at how you would produce the physical machine. Building a computer that doesn't work is really easy. You may have a perfect design and flawless code, but ignore the physical environment in which the machine will exist, and you'll have built yourself a very intricate paperweight. In this chapter, I'll also show you how to lay out a circuit board (and what to be careful of) and how you debug your hardware. We will look at how you physically produce a computer by laying out the design for the ATtiny15 computer, presented in Chapter 15. I'll assume that you're hand-building in small quantities and target the discussion accordingly. What I present here is not the state of the art in circuit board design or assembly, but guidelines for "cottage-industry" computer production. If you need to make production runs of hundreds of thousands, either you already know what you're doing (and can skip this section) or you need to talk to a professional.

Tools

To design and build an embedded computer, you have to start with the right tools. The first thing you will need to design a computer is another computer. It's a chicken-and-egg problem. You will need software tools to create the design (schematic and circuit board layout), and you will need software tools not only to write your code, but also to download it into your embedded target machine.

Development Kits

When you're developing your embedded system, it's best to start with a *development kit* from the processor's manufacturer (Figure 6-1).

INTEL EXHIBIT 1008 Page 66
IPR2023-00787

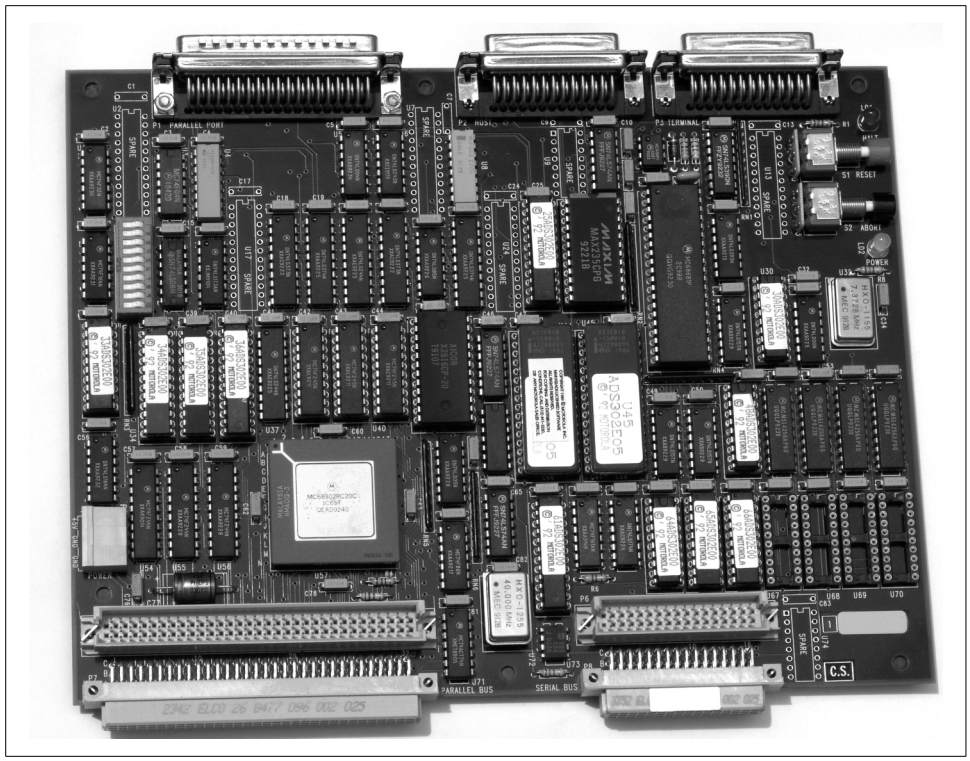


Figure 6-1. Microprocessor development system

A good development kit will not only provide you with a working example of the machine you're trying to build (and upon which you can test your code), it should also include a nice *Integrated Development Environment* (*IDE*). The IDE will have a windowing editor, a debugger, a simulator too if you're lucky, an assembler, and hopefully a C compiler as well. The kit should also come with cables and tools for downloading code to the processor and circuit schematics so you can see what a working machine should look like. However, treat the schematics with a small degree of caution. Some (but not all) semiconductor manufacturers farm out the design of their development systems to small, external companies. Some of these companies do a fantastic job, while others seem to employ stray chimpanzees as design engineers. In the latter case, the development system will work, but only through a miracle and by the grace of the digital gods. So, treat the schematics as a rough guide only. They are not always the best example of how a system should be designed.

To use the IDE, you will need a desktop computer, and here lies the bad news. Almost without exception, the IDEs will run on only one platform and under only one operating system. No prizes for guessing which one. So, if your preferred

INTEL EXHIBIT 1008 Page 67
IPR2023-00787

environment is a Unix workstation, generally you're out of luck. While the gnu tools are great, they don't support a lot of the smaller processors, and sometimes you just have to resort to the chip maker's IDE to download code into your target computer.

Development kit prices range from free (if you're at the right place at the right time) to many tens of thousands of dollars for some of the really high-end and exotic processors. For most embedded-type processors, you can expect to pay somewhere between $50 and $300, depending on the chip, the manufacturer, and their current whim. For the time they will save you, it's probably worth investing in one.

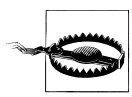
Measurement Tools

For the systems described in this book, the minimum debugging tools you will need are a *multimeter* (Figure 6-2) and an *oscilloscope*.



Figure 6-2. Fluke digital multimeter (reproduced with permission)

A multimeter allows you to measure current and voltage, but, more importantly, it also allows you to do a continuity test between two points (and verify that there is a physical, and therefore electrical, connection). However, do *not* do continuity tests if there are sensitive components in your system. The continuity test may cause them damage.



Don't assume that just because a signal is present at one end of a trace that it is present at all points along the trace. *Check everywhere* with an oscilloscope probe, and use your multimeter to confirm that signal paths are connected properly.

INTEL EXHIBIT 1008 Page 68
IPR2023-00787

An oscilloscope (Figure 6-3) allows you to view waveforms within your system, and, as such, it is your principle debugging tool. Oscilloscopes range from the crude and ancient to the expensive and sophisticated. While you don't need to spend $100k on an oscilloscope, you will need one that can accurately view waveforms. That rules out the $20 antique you picked up from Mr. Gorsky's garage sale down the road.



Figure 6-3. Tektronix oscilloscopes

You will need an oscilloscope of sufficient bandwidth to view the signals within your computer. There's no point using a 20 MHz oscilloscope to look at a 100 MHz system clock. The oscilloscope simply won't see it, and therefore neither will you. The higher the bandwidth, the more you will see. While you may think that the signals of a 4 MHz embedded processor might not require a 100 MHz oscilloscope, that oscilloscope will allow you to see the rising edges of the waveforms as rising edges (and not just vertical transitions) and view miniscule timing differences that may be having an adverse effect. It will also allow you to see fine spikes of noise or ringing on your signal lines, which may be adversely affecting your machine.

I really like the (relatively) low-cost Tektronix oscilloscopes for debugging embedded systems. HP and others also make nice units. If you're serious about developing embedded hardware, it's worth investing in one. Keep an eye out for startup companies going under—you may be able to pick up some great test gear going cheap!

INTEL EXHIBIT 1008 Page 69
IPR2023-00787

When using an oscilloscope, it is critical that you connect the ground clip of the probe to a ground connection close to (or better, *on*) your embedded system. Without this, your measurement of the signal will be affected by ground loop problems, and you will not get an accurate reading. You'll spend ages chasing phantoms, all the while missing the real problem.

Logic analyzers (Figure 6-4) are very expensive tools that allow you to monitor and diagnose a multitude of digital signals simultaneously. They are essential for developing high-speed and complex systems (especially those with buses), but you should be able to get by without them for the simpler designs in this book. Certainly, for a completely self-contained microcontroller, a logic analyzer is of no use at all.



Figure 6-4. Tektronix logic analyzers

In-Circuit Emulators

Another development tool is the *In-Circuit Emulator* (*ICE*). This is a small module with the same footprint as the processor, which is placed into the target system under development. Under the control of software executing on a PC and emulating the embedded processor, the ICE behaves just as the processor would in-circuit. This allows you to interactively debug your hardware and software. This can be especially useful in systems based on self-contained microcontrollers, where it would be otherwise difficult (impossible) to get to the system internals.

Some ICEs are better than others, and, as with everything, you get what you pay for. For really sophisticated tools that closely match the timing and electrical characteristics of the processor, expect to pay big bucks. Cheaper systems will emulate the processor's operation but will do so with completely different signal timings. Also, for each processor type around which you develop systems, you'll need a different ICE.

Some engineers use ICEs heavily during their embedded system's development process. Call me a heretic, but I get by quite well without them. The catch with an ICE is

INTEL EXHIBIT 1008 Page 70
IPR2023-00787

that no matter how good a particular tool is, it is never going to be *exactly* like the real thing. There will always be some slight difference in the electrical characteristics or in the timing. The engineers at Boeing have a saying: "Test what you fly; fly what you test." In other words, there's no substitute for the real thing.

Construction Tools

Finally, you'll need a good soldering iron, some side-cutters, a pair of pliers, and other construction tools. Don't use the same pliers you use to build your fence; get some high-quality pliers with very fine jaws. I have six pairs of pliers, all suited to very fine work, and each is slightly different. I also have a set of jeweler's tools, useful for very fine and delicate assembly work. If you can afford it, a quality binocular microscope is very useful not only for construction, but also for verification of the final assembled circuit. Figure 6-5 shows just a sampling of some of the tools I regularly use during construction.

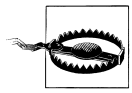


Figure 6-5. Construction tools

Soldering

Soldering is very easy to do well, and very easy to do badly. The basic skills are simple to learn. Becoming a wizard with the soldering iron is not hard to achieve.

INTEL EXHIBIT 1008 Page 71
IPR2023-00787



Safety First

The most important thing to note about solder is that it contains lead. Therefore, all soldering should be done in a well-ventilated work area, and avoid breathing the fumes! After soldering, wash your hands, especially before eating!

Solder can splatter, so always wear protective eyewear and clothes.

Solder is a metal alloy with a relatively low melting point. It is used to bond components to circuit boards and forms a conductive join. Solder is available as either *strand solder* (like a spool of wire) or *solder paste* (out of a syringe). There are two basic categories of soldering tool: the standard *soldering iron* and the *rework station*. Weller (*http://www.cooperhandtools.com/brands/weller*) and Hakko (*http://www.hakko.com*) both make excellent soldering tools. Weller has kindly supplied photographs of some of its products so you may see examples of each type of tool.

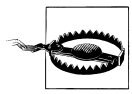
A soldering iron (Figure 6-6) is used with strand solder to mount through-hole components. Note the sponge used for cleaning the iron's tip. The sponge should be kept *very* wet at all times, and the iron's tip should be wiped regularly. (Don't squeeze out the sponge as you would a dishcloth. It needs to be really wet, although not drenched.)



Figure 6-6. Weller WES51 soldering iron (reproduced with permission)

INTEL EXHIBIT 1008 Page 72
IPR2023-00787

There are many different tips available, depending on the type of work you are doing and the temperature at which you want to solder. When the iron is cold, it is a simple matter to unscrew the tip and replace it with another. Tips eventually wear out and need replacing. Spare parts are readily available from most electronics suppliers.



Never remove the iron's tip when the iron is on. The tip contains a sensor that controls the heating element. Removing the tip will cause the iron to overheat and may damage the unit.

Also never leave a soldering iron turned on for extended periods when it is not in use. Doing so will reduce the life of the tool.

Always ensure that the iron's tip has a fine coating of solder, as this prevents oxidization and prolongs the life of the tip. As you work with the tip, it will become discolored. Simply feed some strand solder onto it when it is hot and wipe this excess off on the sponge. This will ensure that your iron's tip is kept in good condition. A well-cared-for tip will allow you to solder successfully. A poorly maintained tip will make the job of soldering very difficult indeed.

Rework stations (Figure 6-7) blow heated air through a small nozzle and are primarily used with surface-mount components and solder paste. However, it is relatively straightforward to solder some surface-mount components using a standard soldering iron. You don't necessarily need the more expensive rework stations, although they do make a lot of surface-mount work easier. Regardless of whether you invest in a rework station, you will always need a conventional soldering iron. There are many components (such as through-hole connectors) that require a standard iron.



Figure 6-7. Simple Weller WAD101 rework station (reproduced with permission)

INTEL EXHIBIT 1008 Page 73
IPR2023-00787

Integrated stations that combine both standard soldering tools and hot-air guns are also available. Figure 6-8 shows a Weller WRS3000ST unit.



Figure 6-8. Weller WRS3000ST (reproduced with permission)

For working on ball-grid array (BGA) packaged chips, you'll need access to a rework station like the Weller WQB 3000 (Figure 6-9). These allow you to accurately place components and successfully bond the many hundreds of solder balls (pads) under a BGA. If you can't afford a machine like this, there are companies that specialize in assembling electronic systems. They will be able to mount your components to your circuit board for a price.

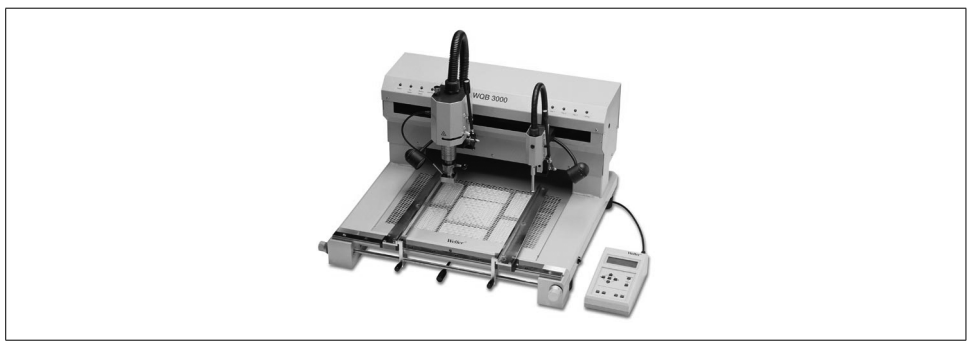


Figure 6-9. Weller WQB 3000 (reproduced with permission)

INTEL EXHIBIT 1008 Page 74
IPR2023-00787

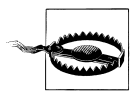
Solder fumes are toxic, and prolonged exposure to them is bad for your health. When soldering, ensure that your work area is well ventilated. It may also be worth investing in a fume extractor, such as that shown in Figure 6-10.



Figure 6-10. Weller fume extractor (reproduced with permission)

How to Solder

The key to soldering well is to control the heat and the amount of solder that flows onto component pins. Too much heat can damage a component (especially sensitive integrated circuits) and can overheat the solder as well. Read the datasheets to determine the maximum temperature (and duration) that the components can take and ensure that your soldering does not exceed this. Variable-temperature irons allow you to set the temperature, thereby avoiding overheating. The tip of your soldering iron should be thin, allowing you to do fine work. An old-style iron with a large, bulky tip (intended for electrical work) is not appropriate for soldering electronics.



Whenever you solder your PCB, make sure it is not powered! The tip of a soldering iron is grounded, and touching this to a pad with volts on it is *not* a good idea!

Similarly, when inserting or removing socketed components, ensure that the system is powered down. Most semiconductors do not appreciate being plugged into a live system.

There should be enough solder to make a good contact, but not so much that it bulges up or, worse, shorts a neighboring pin (Figure 6-11).

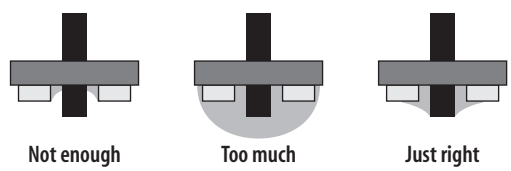


Figure 6-11. Component pins soldered to a PCB

INTEL EXHIBIT 1008 Page 75
IPR2023-00787



During the Apollo/Saturn missions, NASA found that teaching its technicians the correct way to solder saved several hundred pounds in takeoff weight.

When soldering through-hole components (such as DIP-packaged chips or connectors), place the component into its hole and ensure that it is mounted correctly and sits flat. To begin, solder one pin only, then check that the component is still seated correctly before doing the remaining pins. With the iron in one hand and a thin strand of solder in the other, bring the two together such that they meet at the pin to be soldered. Within a second, the solder will flow around the pin and you will have a good join. As soon as the solder begins to flow, remove both the iron tip and the solder strand. If you are using DIP chips, rather than soldering the chip directly into the circuit, solder a DIP socket instead. This allows you to easily remove/replace the chip if you have to, without the inconvenience (and frustration) of desoldering. Figure 6-12 shows a machined socket (left) and a "dual-wipe" socket (right). Machined sockets, while they cost more, are more reliable and will last longer.

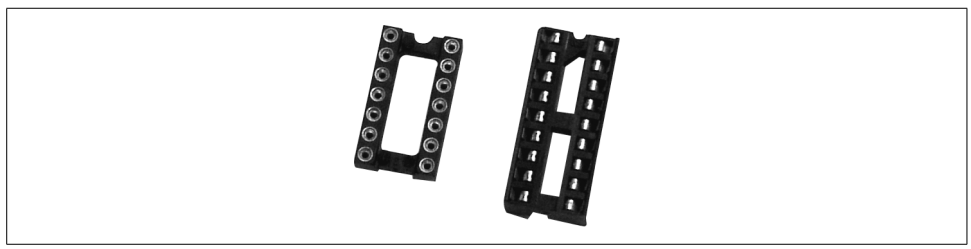


Figure 6-12. Chip sockets

Common mistakes when soldering are to heat the component pin for several seconds before applying solder (causing the component to become too hot) or to apply the solder directly to the iron and then dab the molten solder onto the pin.

Soldering surface-mount components requires a different procedure. If you're using a rework station, you will need to use solder paste. This is sold in a large syringe. Solder paste dries out easily inside the syringe, so ensure that you seal the end when it is not in use. Before soldering a surface-mount chip, place a thin squirt of solder paste along each row of pads on the PCB. If there is too much paste, it can flow under the chip and create a short, so keep the application light. You can always add a small quantity later. Place the chip onto its PCB pads and ensure that it is lined up correctly; then use the rework station to apply heated air (Figure 6-13). Too much airflow will either shift the chip off its correct orientation or, worse, blow solder paste underneath. Since solder paste is electrically conductive, this is not a good thing. Too little heat will result in poorly soldered joints, whereas too much heat can easily overheat and damage the chip. It is something of an art to get it just right, so it's best to do considerable practice before tackling the real thing.

INTEL EXHIBIT 1008 Page 76
IPR2023-00787

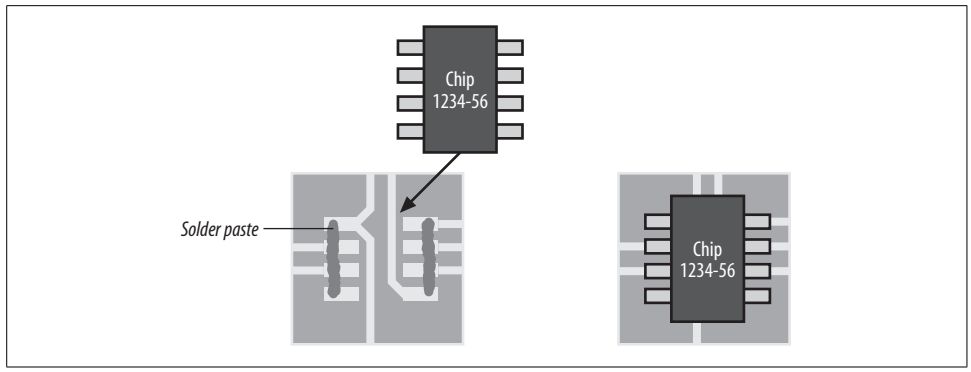


Figure 6-13. Soldering surface-mount components using a rework station

Surface-mount chips can also be soldered using a standard iron, although it's not recommended for really finely spaced chip pins. Unlike the technique with the rework station, solder paste is applied after the chip is in place. To begin, before putting the chip on the PCB, use the iron and either strand solder or solder paste to place a small dab of solder directly onto one of the pads where the chip is to be mounted. Place the chip in position, aligning it carefully, and then use the iron to heat the pin resting on the solder dab. The dab will melt and fix the chip in place. Check the alignment again to ensure that the chip did not shift. If it did, reheat the pin again, and carefully shift the chip as appropriate. Once you are happy with the alignment, place a thin squirt of solder paste down each row of pins, as far from the edge of the chip as possible. Too much paste will flow between the pins, creating shorts, so keep it light. Gently and quickly run the tip of the soldering iron down each row of pins. The solder paste will melt and flow as you go, bonding the chip to the PCB (Figure 6-14).

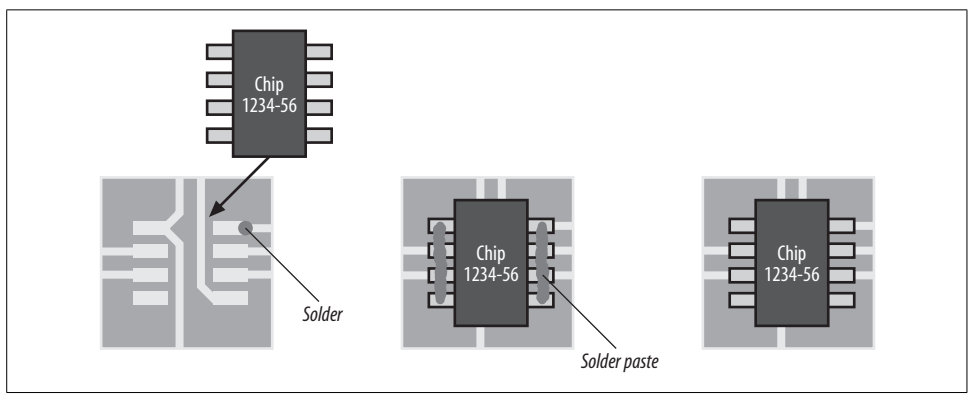


Figure 6-14. Soldering surface-mount components using a standard soldering iron

INTEL EXHIBIT 1008 Page 77
IPR2023-00787

Solder is a metal alloy and incorporates a flux to assist flow. When heating the solder, it is common for the flux to separate and flow out onto the surrounding PCB, leaving a thin brown residue. Excess flux can be removed using special solvents, available from most electronics hobby stores and suppliers. Flux removers can be nasty stuff, so keep them away from skin and plastics and use them in a well-ventilated work area. Flux residue is removed for cosmetic reasons only, making your circuit boards look more professional for your customers. However, as it is for appearances only, and since flux solvents are not good for either you or the environment, if you can avoid using them, please do so.



A Note on Pronunciation:

If you are a resident of North America, solder is pronounced "sodder." If you live anywhere else in the English-speaking world, you will pronounce it as "sol-der." So Americans, be advised that if you say the word as "sodder" to non-Americans, they may not know what you're talking about. Instead, they may think you're confessing to strange and unspeakable acts, rather than talking about bonding metals together.

So, with all that in mind, what are your options for construction? There are several ways of fabricating a computer (or any other circuit). Let's take a look at them.

Quick Construction

It is possible to build very simple circuits by just soldering the components together in free space. For example, the leads of a watch crystal can be soldered directly onto the pins of the processor, with the crystal lying across the top of the processor. Wires are soldered onto the pins bringing in ground and power, and connecting the processor's I/O to the outside world. This technique is variously referred to as a *rat's nest*, *bird's nest*, or "*what the hell is that?*" (Figure 6-15).

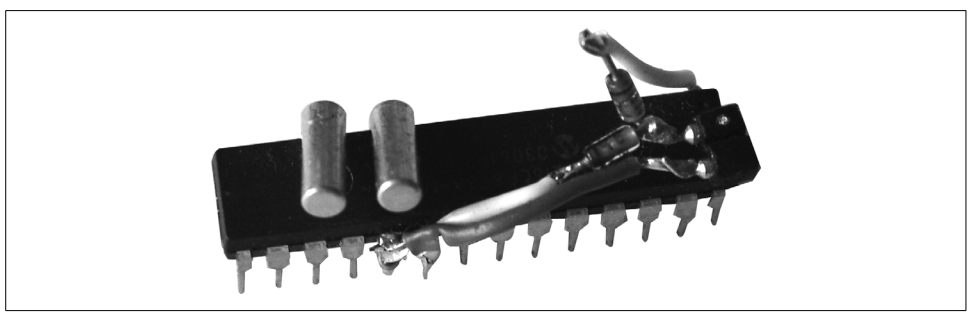


Figure 6-15. Rat's nest construction of a small microcontroller system

INTEL EXHIBIT 1008 Page 78
IPR2023-00787

This is a quick-and-dirty method, useful for rapid prototyping of extremely simple circuits. It's not really recommended, but you can get away with it in a pinch. Don't try it with anything that is even slightly complicated or running at any reasonable speed. If you do, you'll spend more time debugging the construction than debugging the actual design or code!

Breadboarding

Breadboards are plastic blocks with arrays of electrically connected holes (Figure 6-16). They are designed to hold *DIP* (Dual Inline Package) integrated circuits and discrete components. The term "breadboard" dates back to the olden days when valve radios were constructed on a base of solid wood (a cutting board for bread). The term has stuck, and the modern breadboard can still be found in electronics hobbyist stores, and even the occasional university teaching lab.

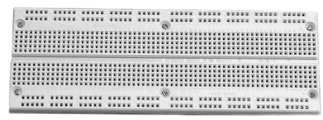


Figure 6-16. Breadboard

As a general rule, breadboards are *bad news*, and their use should be avoided at all costs. (Think of them as the hardware equivalent of COBOL.) They suffer from excessive capacitance, crosstalk, and noise susceptibility and, as such, are completely inappropriate for microprocessor system construction. It's hard enough trying to debug microprocessor hardware and software (together) without the additional complications that breadboards can add. Breadboards can also suffer from mechanical failure after extended use, leading to short circuits. Circuit interconnections on a breadboard are done with small sections of wire. These make great little antennas and will pick up every scrap of stray electromagnetic radiation, channeling it straight into your circuit! Microprocessors don't, as a rule, like "Classic Rock FM" modulated onto their data bus.

Using a breadboard is not the way to construct a robust and reliable system. If you really must, you could probably build the ATtiny15 or PIC12C805 computers (discussed in later chapters) on a breadboard, using their internal RC oscillators. But I'd advise against using breadboards for anything that uses a crystal or that has any fast-switching digital signals. While it is possible to build very low-speed microprocessor systems and general digital circuits on breadboards, *try not to*. There be dragons!

Wirewrapping

Once common as a construction technique, wirewrapping is now relatively rare. It is intended for use with DIP-packaged integrated circuits, mounted into sockets with

INTEL EXHIBIT 1008 Page 79
IPR2023-00787

long pins (0.6"). Figure 6-17 shows an embedded computer built as a student project using wirewrapping.

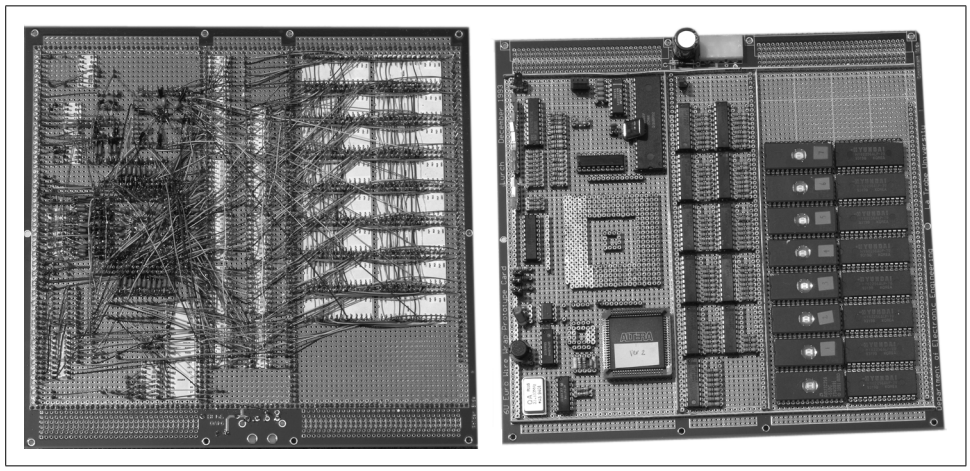


Figure 6-17. Bottom and top views of a wirewrapped embedded computer

Special tools, known as *wrapping tools*, allow you to quickly and efficiently wind thin wire around the pins. Figure 6-18 shows a low-cost, hand (manual) wrapping tool on the left and a wire stripper on the right. Automatic tools are also available.

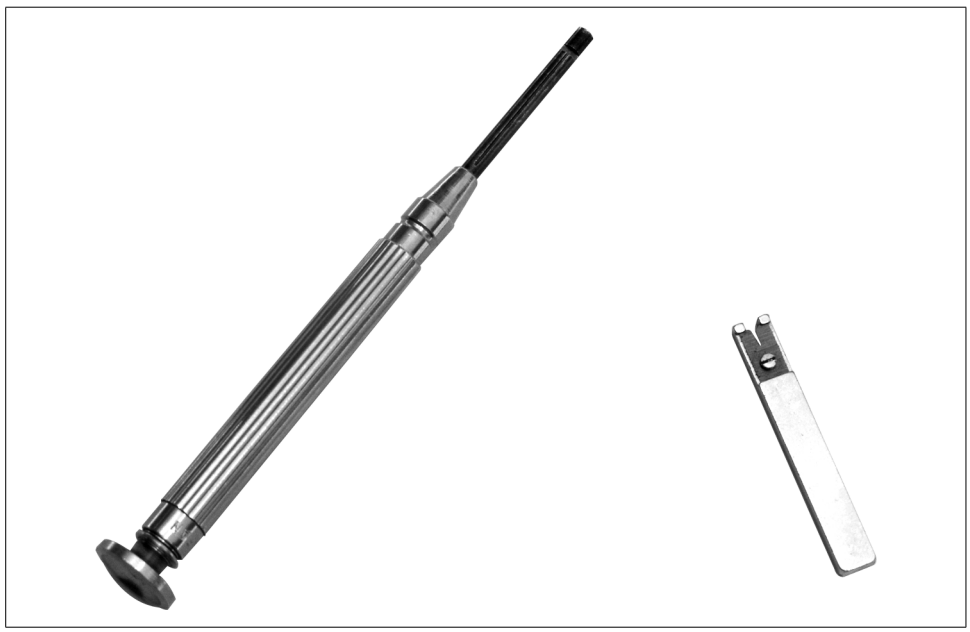


Figure 6-18. Wirewrapping tool and wire stripper

INTEL EXHIBIT 1008 Page 80
IPR2023-00787

The wire stripper is used to remove the insulation from fine-gauge wire (Figure 6-19).

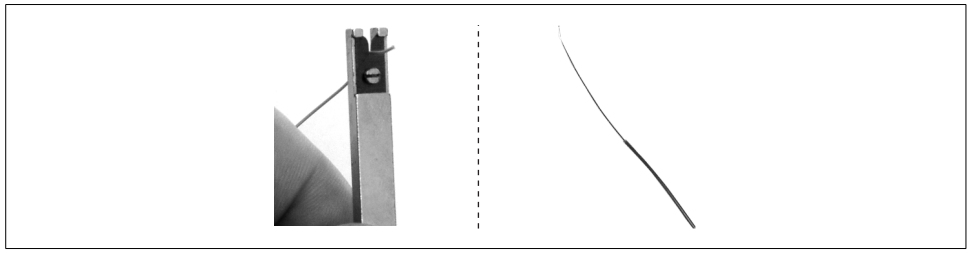


Figure 6-19. Stripping wire

The stripped wire is then inserted into the notch of the wrapping tool (Figure 6-20), and the tool is then placed over a socket's pin. The tool is rotated so that the wire is wound onto the pin.



Figure 6-20. Wirewrapping tool with wire inserted

The pins are square in cross-section, and wrapping a wire around a pin forms a *cold weld*—a tight electrical connection between wire and pin with no soldering (Figure 6-21). In this way, a circuit is constructed by individually wiring point-to-point each connection within the system.

INTEL EXHIBIT 1008 Page 81
IPR2023-00787



Figure 6-21. Close-up of wirewrapped pins (image courtesy of Peter Paine)

Figure 6-22 is a photo of the underside of a wirewrapped computer (in this case, a LISP engine designed and built by Peter Paine). Note the square cross section of the pins and the tightly wound wires.

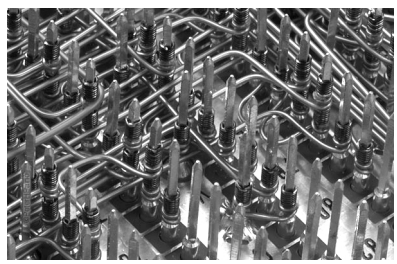


Figure 6-22. Close-up of a wirewrapped circuit (image courtesy of Peter Paine)

Wirewrapping is a very fast prototyping technique and is very robust and reliable. In the early days, NASA used wirewrapping for the construction of spacecraft avionics, and many mainframe computers were also built using the technique. Wirewrapping is good for prototyping (especially if you're unclear as to the final form of the design and expect to make lots of changes to the hardware) or for building one-off designs. If you intend to make more than one computer based on your design (and you probably will), then skip wirewrapping and do it on a printed-circuit board.

Printed-Circuit Boards

Printed-circuit boards (*PCBs*) are epoxy-bonded fiberglass sheets plated with copper. The copper plating is etched away, leaving tracks (traces) that form the

INTEL EXHIBIT 1008 Page 82
IPR2023-00787

interconnections of the circuit (Figure 6-23). PCBs are very reliable and are the only realistic option if you intend to produce more than one system. It is possible to etch your own PCBs, but commercial PCB production isn't that expensive, and it is worth the cost to get professionally produced boards.

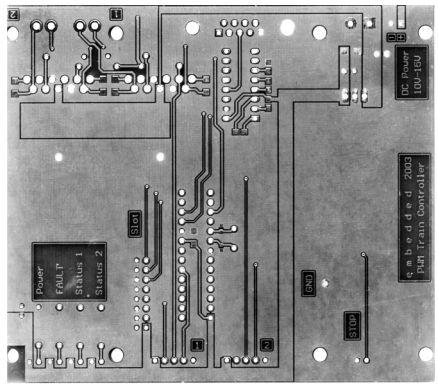


Figure 6-23. PCB for a PIC-based model train controller (Chapter 14)

EDA (*Electronic Design Automation*) software is used to create the schematic and PCB design. The most popular EDA software comes from Mentor Graphics (*http://www.mentor.com*) and Protel (*http://www.protel.com*). There is also a gnu (*http://www.gnu.org*) PCB editor, called *PCB*, which is freely available. Such programs normally come with several tools, allowing schematic entry, netlist generation (a list of what needs to be connected to what), PCB layout, manual routing (making the connections), and autorouting. There's a great temptation to use autorouters, as they simplify the process of generating the PCB by getting your workstation to do the hard work of routing. However, I prefer to lay out the circuit board myself. (I've seen some autorouters make a real mess of a design.) Routing the board manually can take a long while, but it is often worth the extra effort. It can also be very absorbing, much like spending hours in deep meditation. (It's very Zen.)

PCBs can be either single-sided (one layer), double-sided (two layers), or 4-layered, 6-layered, 8-layered, 12-layered, or more. The more layers you have, the easier it is to route your interconnections, but the costs of fabrication go up considerably with extra layers. Further, it's much easier to debug a 2-layered board than a 12-layered board. An advantage of using additional layers dedicated to power and ground planes is that your system will have greater noise immunity. While not so critical for slow 8-bit systems, they are mandatory for high-speed computers.

Multilayered boards will be *plated through*, meaning there will be metallic connections through the holes in the board, connecting traces of different layers together, as appropriate. A *solder mask* is the (normally) green coating on circuit boards, and it prevents solder from flowing between pads and tracks during construction. It is

INTEL EXHIBIT 1008 Page 83
IPR2023-00787

possible to order commercial PCBs without plating through and without solder mask, but the small amount you will save is not worth the hassle.

The *overlay layers* (also known as *silkscreen layers*) are painted on and contain labels (such as "R30" or "RAM4") used during construction and repair to show component placement. The overlay layers are optional. If the boards are to be manually populated with components by someone else, the overlay layers are helpful during construction. If you're building them yourself, then you can easily do without the overlays and save a few bucks.



A useful trick if you're skipping the overlays is to place component information as text on the copper layers. Just be sure to avoid making contact with the circuit tracks!

The external copper layers are called *top* and *bottom* (no surprises there). Traditionally, the top layer was called the *component layer*, and the bottom layer was called the *solder layer*, since components used to be mounted on top and their pins soldered underneath. However, many modern circuit boards place surface-mount components on both sides and are soldered on both sides. Thus, the terms "component layer" and "solder layer" are seeing less use.

There are also internal copper layers for multilayer boards, *mechanical layers* (indicating any special physical features), the *keepout layer* (showing the actual PCB shape), and others. In four-layer boards, it is common practice to use the outer layers for signals and the internal layers for power and ground. This not only provides shielding, but also minimizes the current-loop area, thereby giving your design greater stability.

The five types of objects that can be placed on a copper layer are tracks, individual pads, components (arrays of pads grouped together), vias, and fills.

Tracks are used to interconnect components. Track width is expressed in thousandths of an inch ("mils") or in millimeters (mm). Tracks can be of varying thickness, and often a PCB will have different widths for different tracks. The fatter the track, the more current it can carry. The thinner the track, the easier it is to fit more tracks in a given space, and, therefore, the easier it is to route the PCB. As a general guide, Table 6-1 gives a list of the current-carrying capacity of different track widths for a temperature rise of +10˚C.

Table 6-1. Track width versus current flow

Mils	mm	Amps
8	0.2	0.5
12	0.3	0.75
20	0.5	1.25

INTEL EXHIBIT 1008 Page 84
IPR2023-00787

Table 6-1. Track width versus current flow (continued)

Mils	mm	Amps
50	1.25	2.5
100	2.5	4
200	5	7
325	8.12	10

Check with the company doing your PCB fabrication as to what tolerances it can manufacture. There's no point in doing a PCB with 4 mils tracks if your local PCB fab company can only etch as small as 8 mils.

Pads are used to mount component pins, and they can be round, rectangular, or oval. They consist of a hole and a copper surround. A pad for a component in a DIP, for example, will be a multilayered pad, meaning that the pad appears on all copper layers, and the hole is drilled through the entire PCB. A surface-mount component will have pads that appear on one layer only (Figure 6-24). An array of pads grouped together to form a component package is known as a *footprint*. Surface-mount components have holes of zero diameter (in other words, they aren't drilled). These components are small, with "gull-wing" pins that mount flat on the PCB. They are less susceptible to noise interference than the older DIP-style of packaging. However, DIP (through-hole) components may be easily mounted in sockets and are therefore easily removed during debugging. DIPs are sometimes preferable (although not always feasible) during early development, while surface-mount is the only option for production. Surface-mount components radiate less noise and are less susceptible to external interference. Conversely, DIPs have terrible noise-related characteristics.

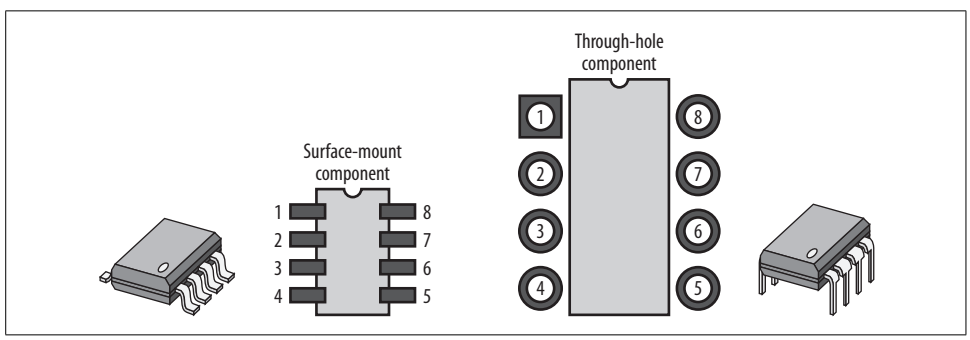


Figure 6-24. Footprints of surface-mount and through-hole (multilayer) components

Tracks entering a pad should "aim" directly for the pad center, as shown in Figure 6-25, and *not* as in Figure 6-26.

When specifying the pads for a component, ensure that the pad size is large enough to accommodate the pins and to allow enough space on which to solder. Also ensure that the holes (for through-hole components) are large enough to take the pins. A

INTEL EXHIBIT 1008 Page 85
IPR2023-00787

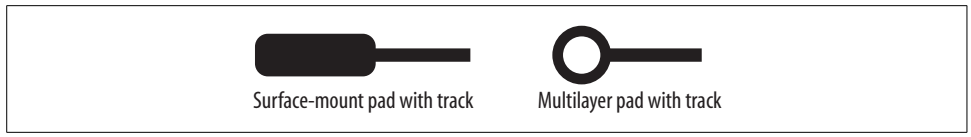


Figure 6-25. Surface-mount and through-hole pads

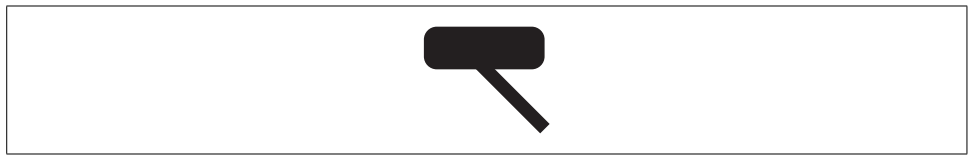
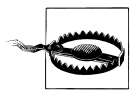


Figure 6-26. The incorrect way for a track to enter a surface-mount pad

standard DIP pin will happily go into a 0.7 mm hole, while a DB connector requires 0.9 mm holes for the signal pins and 3 mm holes for the mounting pins.



Don't assume that the libraries of predefined components that came with your PCB CAD software have the pads, spacings, or holes right. It is not uncommon for CAD libraries to get it very wrong. (No kidding.) There's nothing worse than getting a beautiful new PCB back and finding that you can't mount the components! So, check and recheck.

When routing tracks around pads, ensure that there is sufficient clearance, as shown in Figure 6-27. Tracks should always change direction by 45-degree turns. Some PCB editing programs allow you to do a *design rule check* (also known as an *electrical rule check*) to ensure that correct clearances are maintained and that there are no potential shorts. (It's no guarantee that there won't be a problem, but it's a start!)

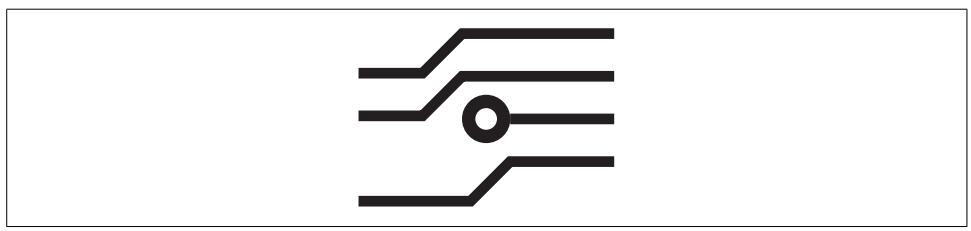


Figure 6-27. Routing tracks around a component pad

Avoid close passes and right-angle turns, as shown in Figure 6-28.

Closely spaced pads on surface-mount components can present a problem. Often the tracks leaving a surface-mount device are too close together to easily route tracks. The solution is to *fan out* the tracks, thereby giving greater spacing. This is shown in a simplified form in Figure 6-29.

INTEL EXHIBIT 1008 Page 86
IPR2023-00787

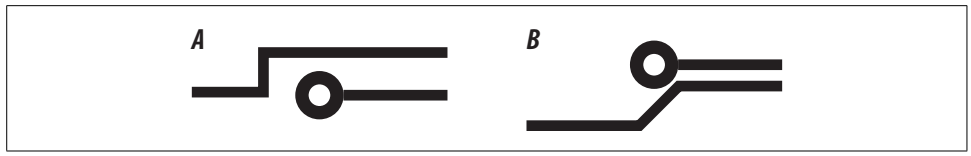


Figure 6-28. How not to route tracks around pads

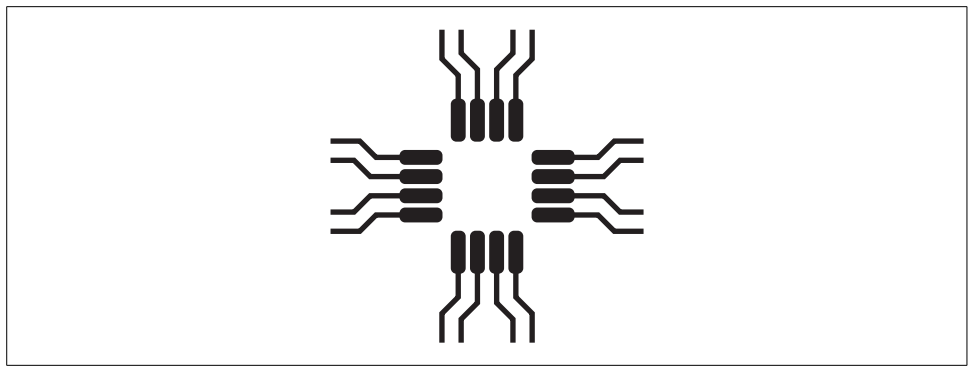


Figure 6-29. Fan out from a surface-mount device

Vias are used to connect tracks on different layers together (Figure 6-30). They are, in effect, little pads. Vias can either be through-hole vias appearing on all layers (Figure 6-31), or *blind* or *buried vias*, appearing only on the layers they are interconnecting and on intermediate layers. Making the vias as small as possible aids in routing the PCB, but check with your PCB manufacturer as to how small you can go. Remember to ensure that the outside diameter of the via is sufficiently bigger than the hole, such that the entire via is not drilled out during fabrication. If space permits, a useful trick is to make the vias with 0.4 mm holes. That way, if there is a bug in the PCB layout or a manufacturing fault, you can use the vias to solder in wire-wrap wire and manually make (or remake) a connection.

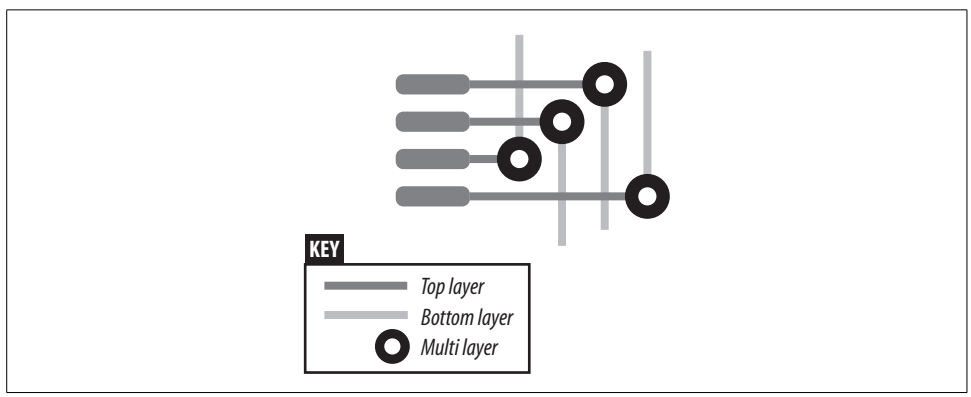


Figure 6-30. Vias connecting top and bottom layer tracks together

INTEL EXHIBIT 1008 Page 87
IPR2023-00787

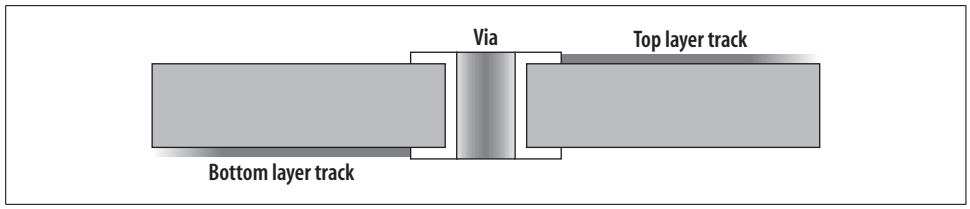


Figure 6-31. Via in cross section

Fills are used to provide shielding to certain sections of the PCB, and also for circuit paths that carry a lot of current. It is common to place ground fills in and around analog sections of the circuit to isolate them from digital crosstalk.

Laying Out a PCB

The first thing to note when laying out a PCB is that someone (or some robot) is going to have to assemble it. As tempting as it might be to cram everything into the smallest space possible, remember the limitations of whoever (or whatever) will be building it. That's not to say you should make the PCB as big as possible—just be realistic. Also, don't bring components and tracks right to the edge of the PCB. Leave a spacing of 5 mm (200 mils) around the outside. If your PCB is to go inside a case, or be mounted in some way, make sure you have the dimensions correct and don't forget to add mounting holes.

There are two schools of thought in placing components (especially integrated circuits) on a PCB. The first is that all components should be placed with the same orientation. For instance, the "pin 1" of each chip should point to the upper lefthand corner of the PCB. This simplifies populating (or *stuffing*) the board with components, especially if this is to be done by a contract manufacturer in an automated process. Having varying orientations may add to the expense of production if you're not hand-assembling the boards yourself. Some people also think that this makes a board look neater.

The second school of thought is that you orientate the chips so as to optimize the routing process. The pinouts of different chips are not necessarily conducive to uniform orientation, and spinning one chip 90 or 180 degrees to its neighbor may greatly simplify the routing of tracks between the two. This can lead to a smaller board size, fewer vias, and shorter track lengths. This then results in lower PCB cost, less noise, less crosstalk, and better noise immunity, which is especially important in higher-speed systems.

Whatever you decide about orientation, group related components together. Put the voltage regulator and its support components near the power connector. Any analog circuitry (such as sensors or amplifier circuits) should be as far from the power

INTEL EXHIBIT 1008 Page 88
IPR2023-00787

connector and its support components as possible. By placing chips into functional groups, routing is made easier. This may seem like obvious stuff, but it's amazing how often it's ignored.

The clocks and high-speed signals should be routed first, to ensure that they take the most direct path possible from source to destination. Where appropriate, place shielding (fills) to isolate these signals from other parts of the PCB. This should be done prior to routing other connections; otherwise, there may not be sufficient space later on. In particular, tracks should never be routed under or around crystals, oscillators, or any clock-generation circuit, and these components should be isolated by fills (connected to ground) from the rest of the circuit. Crystals should lie flat against the PCB (rather than being mounted vertically), and a ground plane should be placed under them to shield from emissions.

For high-speed signals, make sure there is a ground return path close to the track so that the current-loop area is minimized. Allow as much space as possible between high-speed tracks. Having two rapidly changing signals in close proximity will result in crosstalk, and this will cause unreliable operation. Every track has an inherent impedance (resistance); although small, it can affect the transmission of fast signals. In particular, a via or sharp corner represents a change of impedance along the track, and this can cause signal reflections. Therefore, it's important to keep the number of vias to an absolute minimum and avoid right-angle turns in tracks. If you need to make a track turn 90 degrees, use two 45-degree turns in succession.

In high-speed systems, you need power and ground planes that are continuous. In other words, you need planes that cover the entire PCB with no breaks. Any break in the power or ground plane makes the current-loop area larger, and this can increase inductance and radiation. This means that for high-speed systems, you really need to use four or more layers on the PCB. For low-speed microcontrollers, you can get by without separate planes or by providing fills in and around components on the signal layers.

When routing buses (such as data and address), keep the tracks running parallel if possible (Figure 6-32). This is bad practice for clock signals, since it can induce crosstalk in neighboring tracks, but is appropriate for buses. The reason is that bus signals will change state together and will then hold that state until the next transaction. The device receiving the bus signals will sample their state only when they are stable (unchanging). Since crosstalk is generated when a signal state changes, running parallel buses is not a problem. By keeping the bus tracks parallel, the signals travel approximately the same distance for each track. A track that takes a different path completely will have a trace-length mismatch, and this can increase signal *skew*, in which the time it takes for a signal to propagate is shifted. This can adversely affect signal quality in high-speed systems.

INTEL EXHIBIT 1008 Page 89
IPR2023-00787

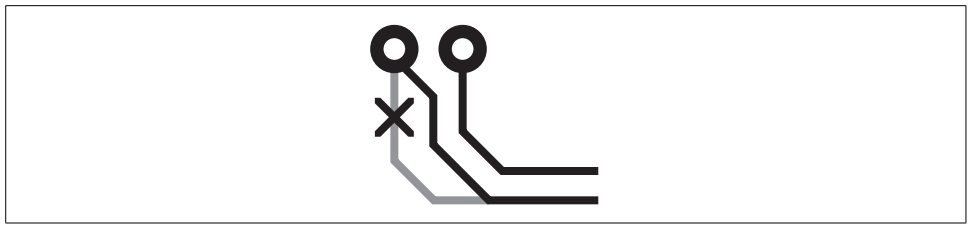


Figure 6-32. Keep buses parallel to minimize skew

Stubs are short tracks that leave a main track to connect to a component (Figure 6-33). A stub represents an impedance mismatch for a signal and can result in reflections. A better way is to place the component such that the pad lies on the primary signal track (Figure 6-34).

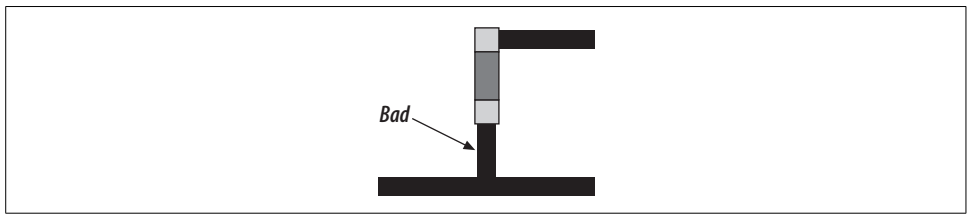


Figure 6-33. Avoid stub tracks

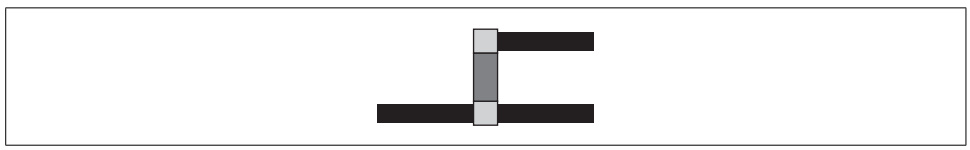


Figure 6-34. Place surface-mount components directly onto tracks, where possible

All power and ground traces should be as fat as possible, and, if feasible, separate power and ground planes (layers) should be used. The *power ground* (ground coming in with the power supply) should be separate from the *signal ground* or *digital ground* (the ground running to all your chips), and both should be separate from the *analog ground* (the ground for your analog components), if one is present. They should all be connected together, but only at one point. This helps isolate the digital and analog sections from each other's noise, as well as from the power-supply noise. Decoupling capacitors should be as close as possible to each power pin of each integrated circuit. Figure 6-35 shows two components on a two-layer PCB, with power and ground tracks routed horizontally through the middle and decoupling capacitors placed close to the power pins.

INTEL EXHIBIT 1008 Page 90
IPR2023-00787

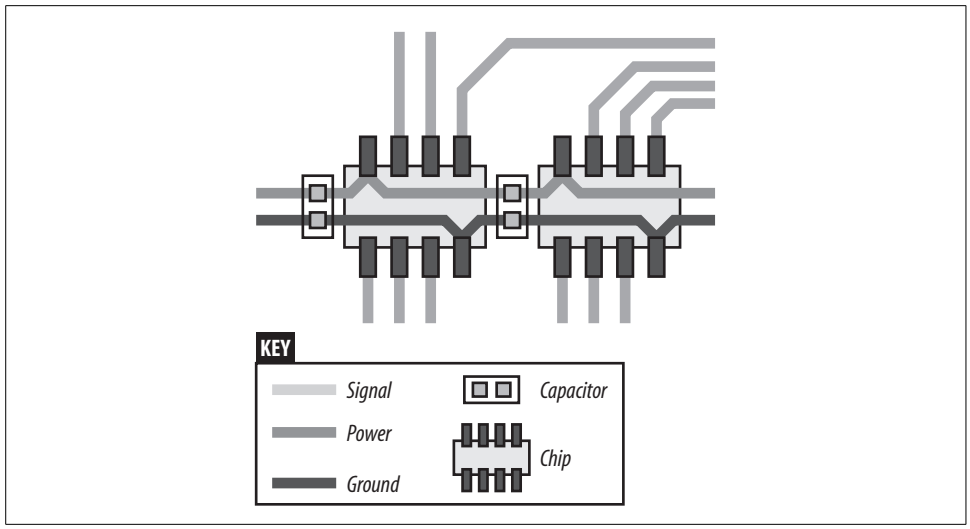


Figure 6-35. Decoupling capacitors should be placed as close as possible to the chips' power and ground pins



A very useful and simple idea is to place a single pad in the middle of a ground plane (or fill) off to the side of the board. Solder in a post (or even a wirewrap pin), and use this to connect the ground lead of your oscilloscope probe. This can make the debugging process a lot less awkward. You must always use the ground lead of your oscilloscope probe. Without it, you can't get an accurate picture of the timing and voltages of your signals. (Voltage is the potential difference between two points, so you must have a reference.) It's very important.

So, with all that in mind, let's create a simple circuit board for an AVR computer. This design is covered in detail in Chapter 15, but for the moment, we'll just use it as an example so that we can see the process of producing a printed-circuit board.

We start with the schematic (Figure 6-36). This design brings together the voltage regulator circuit (top), the AVR processor (left) with a status LED (far left), and the in-circuit programming interface (right).

Note that no connection is made for pin 2 of the connector. This is the +5 V supply provided by the in-circuit programmer. Since our embedded system has its own supply of +5 V (**VCC**), the external source is not required. (If we were building a 3 V version of this computer, then we would need to use the programmer's +5 V supply, and we'd have to disable the output from the voltage regulator during programming.)

INTEL EXHIBIT 1008 Page 91
IPR2023-00787

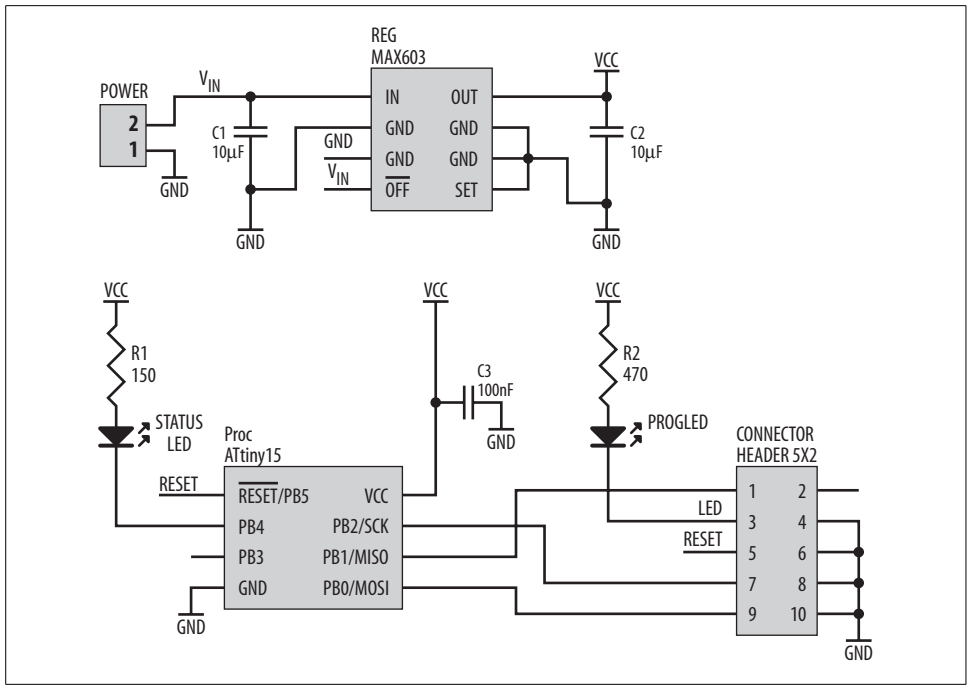


Figure 6-36. The full schematic, including connector for I/O and programming

From this design, we use our schematic editor to generate a *netlist* file, which tells the PCB editing software what interconnections need to be made.

Importing the netlist file into the PCB editor automatically loads the component *footprints*. These are manually rearranged to provide optimum placement, ensuring the shortest track runs between components (Figure 6-37). Note how related components are placed together, such as the voltage regulator, C1, C2, and the power connector. The silkscreen (overlay) layer shows the outline of the components, as well as the component labels. In this example, surface-mount components have been used for the resistors, capacitors, and LEDs, while the two integrated circuits are DIPs. Note the three-pad triangular LEDs ("STATUS" and "PROGRAM"). Only two of the pads are connected to the internal LED; the third is unused. This tiny circuit board measures just 2" by 0.6".

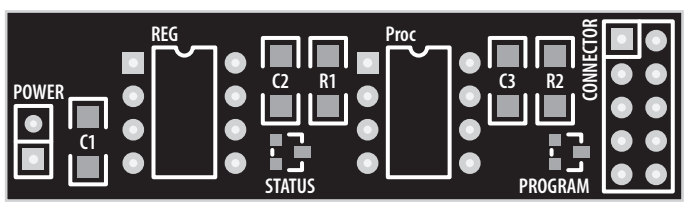


Figure 6-37. Components placed onto PCB

INTEL EXHIBIT 1008 Page 92
IPR2023-00787

Once you're satisfied with the component placement (and this may need tweaking as you go), the connections are routed. Figure 6-38 shows the PCB with manually routed connections. In this case, the overlay layer has been "turned off" for clarity. Note the use of fills for power and ground connections. For such a simple circuit, operating at low speed with no external system buses, the PCB layout is relatively trivial.

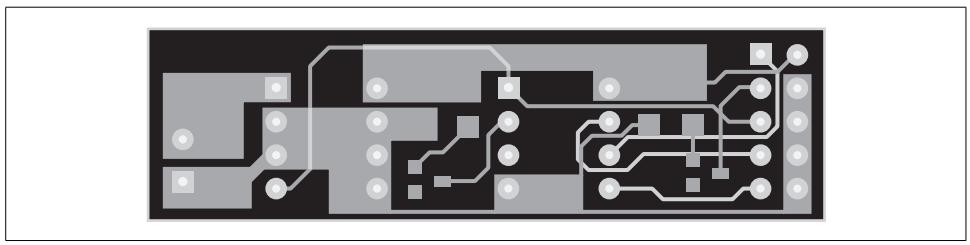


Figure 6-38. Manually routed board, using fills for power and ground

Just for comparison, Figure 6-39 shows the same board, but this time using the EDA software's autorouter to make the connections. Note the bizarre track loop near pin 8 of the processor, the strange meandering track paths, and the unnecessary via in the middle of the PCB.

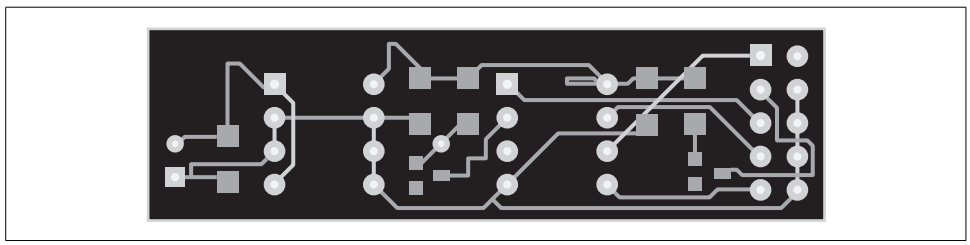


Figure 6-39. The psychedelic results of using an autorouter

This is a simple board, so even an autorouter can make most of the connections. On complex boards, the average autorouter gives up about halfway through, after first making a complete mess. There are autorouters that do a *much* better job than this, but they are very expensive.

For greater noise immunity, a polygon plane is placed on the bottom layer of the manually routed PCB to act as a Faraday shield (Figure 6-40). Note how the polygon has "flowed" around the component pins, yet has connected to the ground pins. In this way, the polygon is a ground plane, providing a (small) degree of noise immunity for the system. The "void" region in the middle right is where it could not reach due to the prerouted tracks. If you are designing a four-layer board, the polygon fill would be placed on a separate layer and should have no discontinuities at all, except where it flows around the pads of through-hole components.

INTEL EXHIBIT 1008 Page 93
IPR2023-00787

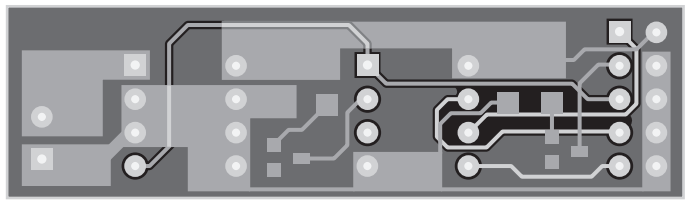


Figure 6-40. Manually routed PCB, with Faraday shield

Using all surface-mount parts (where possible) makes the design even smaller. Figure 6-41 shows the same design (manually routed), but this time using surface-mount versions of the processor and regulator, and even smaller-sized resistors and capacitors. The new board size is just 1" by 0.5". (The pads for the regulator are covered by fills and so are not apparent in the figure. Once the PCB is fabricated, the pads stand out easily.) The only "through-hole" components are the power connector (lower left) and the I/O connector (upper right). Note the four vias needed to route the tracks. For the previous design, the pads of the DIP components effectively acted as vias. On the all-surface-mount version, since just about everything is on the same layer, vias are required to take tracks to the bottom layer of the PCB.

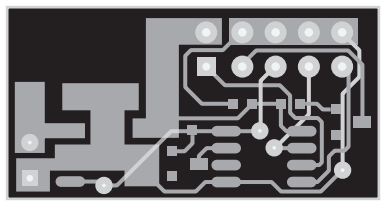


Figure 6-41. Surface-mount PCB

You could make this board even smaller (perhaps 0.5" by 0.5") by placing surface-mount components on both sides of the PCB. This adds to the cost of construction if you're having it professionally done.

Figure 6-42 shows the surface-mount PCB, now with a Faraday shield.

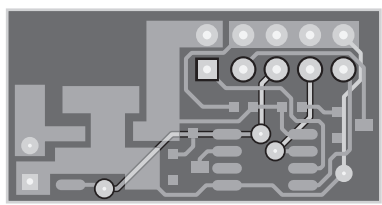


Figure 6-42. Surface-mount PCB, with Faraday shield

INTEL EXHIBIT 1008 Page 94
IPR2023-00787

Before sending off your circuit board design to be fabricated, print it out and carefully look at it. Check clearances to ensure that there are no potential shorts. Just because there's a whisker gap between a track and a via on the screen doesn't mean there won't be a short between the two when the PCB is made. Give enough clearance to make this an impossibility. Good practice is to set your clearances to be equal to or greater than the minimum track width to which the PCB manufacturer can etch. Anything finer, and you're asking for trouble.



When your design is printed out, place the physical components on the paper and check for clearances. Just because the component outlines in the CAD package didn't collide does not mean the physical chips won't. It is much easier to solve these problems before the PCB is fabricated than after.

Building It

Once the PCB has been fabricated and checked carefully to ensure that all pads and tracks are intact and properly etched, do the construction one step at a time and check everything as you go. Using a multimeter, do a continuity test between the ground pads and the ground pin on the power connector.

Powering Up for the First Time

Start construction by soldering in the power connector, voltage regulator, and its support components, including the "power" LED if you've included one in your design.

Once you've soldered in the components needed for the power supply, power up the board and check that it is operational. Also check that you have power on every pad on the board where you expect power to be, and check the ground pads to make sure there is no power where you expect no power to be.

Next, solder in the power-decoupling capacitors for the ICs. Add the processor's oscillator and decoupling capacitors. If the oscillator is a module, check its operation with an oscilloscope. Does it have the right waveform on its output pin?

If IC sockets are used, solder these next and then insert the components. If you're using a processor that needs to be externally reprogrammed, then a socket is a good idea.

Add in the Processor

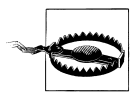
Make sure once again that the system is powered down. Using a multimeter's continuity test function, check that the processor footprint has the appropriate power and ground connections to the right pins. Solder the processor to the board (or plug it

INTEL EXHIBIT 1008 Page 95
IPR2023-00787

into a socket if you've used one), remembering to ensure that the system is powered down before you do so. If your processor needs to be externally programmed with your code, make sure you do so before putting it into your circuit. Power it up and check the clock to the processor with an oscilloscope to confirm that it is oscillating. You should see a nice sine wave of the appropriate amplitude. Check the voltage levels you measure against those stated in the datasheet. If the oscillation doesn't have the right amplitude (perhaps due to a poor connection or a partial short), it may not be able to drive the processor.

If the system you are building uses a microcontroller with no external ROM (such as the example presented in this section), the first test software you will write will simply waggle an I/O line. Observing this with an oscilloscope will allow you to see if your system is executing code correctly. If you included a status LED in your design, turn it on! Seeing a status LED blink on for the first time in a machine you've designed and built yourself is sure to bring a smile to your face.

Once you've confirmed that the processor is operating under software control, you can begin to add in the other hardware and software components of your application.



Don't try to be too adventurous at any stage of the building process. If everything suddenly stops working, it's much easier to find the cause if you've only made one change or addition. Take things one step at a time.

Some Thoughts on Debugging

Debugging is as much an art as it is a science. You can load a workbench to its breaking point with all sorts of expensive test equipment, yet without a logical approach and a clear mind, elusive bugs will never be found. Conversely, by "right thinking," the strangest of bugs can be isolated with a minimum of tools. While it is true that the more complex the system being tested, the harder it is to nail down a fault through detection, it is also true that the most advanced and useful debugging tool you have at your disposal is your own brain. Therefore, learning to debug is learning to think carefully and clearly.

Debugging hardware can be a lot trickier than debugging software. With code, you can always put in some diagnostics to inspect the execution. That's not to say that debugging software is trivial—far from it. But with hardware, it is often either a case of everything working or nothing working. Software has the advantage of being able to be brought into operation gracefully. For hardware, you need to have an awful lot working right from the start.

The essence of debugging is establishing what works and what doesn't work. The design presented in this chapter is intentionally simple and should require little or no

INTEL EXHIBIT 1008 Page 96
IPR2023-00787

hardware debugging. But as the designs grow in complexity, finding hardware and design faults can become quite an involved problem.

For example, your embedded system may not be outputting characters through its serial port. Why? Perhaps it's a bug in the code. Maybe there's a cable fault. Maybe the RS-232C interface chip is dead. Maybe the serial chip itself is dead. It might be that there is a timing problem with the serial chip's oscillator, or a voltage-level problem. Perhaps the processor itself is not coming out of reset and therefore not executing code at all. If so, maybe it's the power-on reset circuit failing to kick in, or the brownout detector kicking in when it shouldn't. Maybe a data line between the processor and the serial chip is not connected, perhaps due to a manufacturing fault with the PCB. Or maybe it wasn't soldered correctly. Perhaps your voltage regulator isn't operating properly, or maybe you have a faulty power supply. And those are just the obvious causes that spring to mind. There are a thousand others lurking, with big teeth and a nasty disposition.

For any one problem, there is a multitude of possible causes. Debugging is therefore about isolating a fault, and this is best done by a "20 questions" approach. Divide and conquer to solve the problem.

Let's take the example of the faulty serial-port problem above. You discover the problem when you first try to test the serial port. Your simple test code fails to output a character. Is the problem in software or hardware? If hardware, is the problem with the cable, the serial chip(s), or is it a more fundamental problem with the core system? Check the cable and the terminal (or host PC) first. Disconnect the cable from the embedded computer, and with a piece of metal (a screwdriver blade will do), short out pins two and three (**Rx** and **Tx**) on the cable connector. Now type something on the terminal (or the terminal software on the PC). What comes out of the terminal should echo back through the short and appear on the screen. That will tell you whether there is a cable fault and whether the terminal is set up correctly.

If that works, then the problems lie in your embedded system. Replace your serial test code with code that does something else that is simpler (like waggle a digital I/O line, or flash a LED). That simple action will tell you volumes. (Archimedes once said, "Give me a lever long enough and I will move the world." Well, give me a status LED and enough time, and I'll debug the world too!) It will tell you whether your processor is executing code correctly, which in turn shows that the processor and ROM (if a separate chip) have power and are communicating correctly. It shows that the reset circuit, brownout detector, oscillator, voltage regulator, address decoder, and other support logic are OK. If any of these are failing, then the processor will not be executing code, and therefore that I/O line will not waggle or that LED will not flash. With that simple test, you have ruled out a plethora of possible faults.

Otherwise, you know to look elsewhere for the problem, such as checking the oscillator, reset, or voltage regulator for correct operation. Divide and conquer. If the test passed, then the fault lies with the serial chip. Most serial chips include some digital

INTEL EXHIBIT 1008 Page 97
IPR2023-00787

I/O that can be manually set (such as **RTS**). Write some test code that does this. This simple test will show whether or not you can talk to the chip. If the test passes, you know to look at either your character-output software or the RS-232 driver. If the test fails, then the problem lies in talking to the chip. Use an oscilloscope to check the chip select and other control signals going to the serial chip. Are they active? Are they reasonable? Write some software that continually "jams" a byte at a register in the serial chip. While meaningless to the serial chip, a continuous write of the same number allows you to observe the bus activity. So, your (pseudo) code to do this is:

```
          load    r1,#0x55          ; load %01010101
loop      store   serial_control    ; write it
          jump    loop              ; continuously
```

You will expect to see the above bit pattern on the data bus (and, importantly, on the appropriate pins of the serial chip) at the same time the chip select and write enable are asserted.

This will enable to you to locate a problem with the processor writing to the serial chip. Alternatively, if you can demonstrate that you can write to the chip correctly, then the problem lies either in the software or between the serial chip and the serial connector. By using the divide-and-conquer approach, you can isolate where a problem lies. Devise tests to prove each aspect of system operation.

Often you will be faced with a bug that makes no sense. Something should be working, and it is not. Everything you check seems right, but the total system just isn't working. It can be very perplexing. You have made a common error—you have made an assumption. Somewhere, even though you may not be consciously aware of it, you have assumed that some little detail is correct, when in fact it is not. This is the hardest obstacle to overcome. When you say to yourself, "It should be working, but it isn't! It doesn't make sense!" then say to yourself, "There is still something I haven't checked." Go looking for it. If you can't find it, then you haven't looked hard enough.

When designing your system and laying out the PCB, remember that you will have to debug it. So, design it with debugging in mind. Include one or more status LEDs. These are invaluable for debugging embedded hardware. Sure, you can do a lot with a remote debugger (such as gnu's `gdb`), but you have to get the hardware working to a certain level before the debugger can be made to run. Status LEDs will help you get there.

You are also going to need to look at signals with an oscilloscope, so include a ground pin on your circuit board onto which you can clip it. Also, make sure that you will be able to get an oscilloscope probe to every circuit trace on the board to examine what's going on. If you can't get to a track, you can't ensure that there's no problem with that particular signal.

So even at the design stage, think carefully about how you can test the subsystems and isolate problems and put the necessary support into your design.

INTEL EXHIBIT 1008 Page 98
IPR2023-00787

Read the Electrical Specifications!

I once designed a system that operated on a 5 V supply. It worked wonderfully. I then went to produce the same system operating on a 3.3 V supply by using 3.3 V versions of the same parts. It should have worked. It didn't. All the timing was correct; all the voltages were correct. Everything I checked was right, and yet nothing. No activity, not even a trace of a signal from anything. I couldn't understand it. There was nothing left to test.

I knew the design and code were correct because they worked beautifully in the 5 V version. It had to be something specific to the hardware in the 3.3 V system. But what? The processor had power, the regulator was working, the oscillator was going, and the reset circuit was operational. It should have been executing code. Yet, even the simplest of test software failed to go.

Somewhere, there was an incorrect assumption I was making. It took me over a week to find it, and it was as subtle as they come. In going from the 5 V system to the 3.3 V system, I had chosen the 3.3 V version of the processor. What I had *assumed* (logically, but incorrectly) was that the brownout detector (inbuilt in the processor) was designed to work at the correct levels. You'd expect that, but it was wrong. The manufacturer of the processor, when producing the 3.3 V chip, had changed the operating voltage of the device but had left the brownout detector *unchanged*. So, in the 3.3 V processor, the brownout detector kicked in if the supply was less than 4.5 V! Hence, the processor never came out of reset and therefore never executed code.

I had assumed, *incorrectly*, that everything about the 3.3 V processor was designed to work at 3.3 V. For correct operation, the 3.3 V processor needed to have the (optional) brownout detector disabled. This was not explicitly stated in the datasheet, merely implied through careful reading of the electrical specifications.

The moral of the story: don't assume *anything*, and check *everything*. If it still doesn't work, you haven't done the checking carefully enough.

JTAG

A *JTAG* (Joint Test Action Group) port provides access to the internals of the processor and, through it, the rest of the computer system. JTAG is defined under IEEE standard *1149.1a-1993 Standard Test Access Port and Boundary Scan Architecture*. It is sometimes also known as a *Test Access Port*, or *TAP*. Commercially available test suites use JTAG to provide in-circuit debug capability. The adventurous among you can also drive JTAG "manually," using the information in the aforementioned document.

The JTAG port allows for real-time debugging of hardware and software. It allows you to single-step or multi-step through code running directly on the target system. You can individually (and manually) toggle signal lines of the processor to test

INTEL EXHIBIT 1008 Page 99
IPR2023-00787

Using JTAG

To utilize JTAG, the processor you are using must specifically support it. In other words, the processor must have a JTAG interface inbuilt. JTAG is very rare in small microcontrollers, with most providing no support. However, more advanced microcontrollers and faster processors typically do include a JTAG interface, and this is an essential part of the debugging process for systems based on these chips.

external subsystems in the computer (also known as *boundary scan*). You can set breakpoints both at locations in code or for when a particular address (or device) is accessed. The JTAG port allows you to examine and modify registers and memory locations. To utilize the JTAG interface, you need to have support tools that are JTAG compliant. For more information, refer to IEEE standard 1149.1a.

Boundary scan allows you to probe the microprocessor and the circuit connections between peripherals and memories. It does this by asserting outputs (independently of the CPU) and reading the response from external devices on input pins. It is useful for testing not only interconnections on the PCB, but also for design verification and even correct timing. JTAG can operate independently of the CPU, "manually" driving outputs, and can interrogate the processor as to its manufacturer, processor type, and revision number. JTAG can also be used to disable output pins while a board is undergoing a test. Freescale Semiconductor (formerly Motorola) has added functionality to the JTAG interface through an *On-Chip Emulation module* (*OnCE*) found on many of its processors. The OnCE can let the processor run and watch system activity in response to the executing software. It can retrieve or set parameters in registers or memory, as well as provide a host of debugging features (such as setting breakpoints, single-stepping, and instruction tracing).

A JTAG port provides access to a state machine that implements the boundary-scan functionality. The state machine has four registers. These are the *Instruction Register*, the *Boundary Scan Register*, the *Device Identification Register*, and the *Bypass Register*.

A JTAG port consists of four dedicated signals (Table 6-2). If you think those signals sound suspiciously like a synchronous serial interface, you'd be right, because that is exactly what JTAG is.

INTEL EXHIBIT 1008 Page 100
IPR2023-00787

Table 6-2. JTAG signals

Signal name	Function
TDI	Test Data Input
TDO	Test Data Output
TMS	Test Mode Select
TCK	Test Clock

JTAG is becoming increasingly common due to the many advantages it brings to the system designer. Most new processor releases are including JTAG in their design. For information regarding the use of JTAG for a specific processor, check the manufacturer's technical data.

INTEL EXHIBIT 1008 Page 101
IPR2023-00787

CHAPTER 7

Adding Peripherals Using SPI

Thirty spokes meet at a nave;
Because of the hole we may use the wheel.
Clay is molded into a vessel;
Because of the hollow we may use the cup.
Walls are built around a hearth;
Because of the doors we may use the house.
Thus tools come from what exists,
But use from what does not.

—Lao Tse
Tao Te Ching

In this chapter and the next, we'll look at two low-cost interfaces used to connect peripheral chips to microcontrollers, within a single embedded system. These interfaces allow you to connect devices such as real-time clocks, nonvolatile memories for parameter storage, sensor interfaces, and much more. The interfaces are easy to use and cheap to implement, making them ideal for small, embedded applications. Some microcontrollers incorporate both types of interface, whereas others may only have one or the other. The one to use really depends on what your processor has to offer and the requirements of the particular peripheral you're using.

Serial Peripheral Interface

The *Serial Peripheral Interface* (known as *SPI*) was developed by Motorola to provide a low-cost and simple interface between microcontrollers and peripheral chips. (SPI is sometimes also known as a *four-wire* interface.) It can be used to interface to memory (for data storage), analog-digital converters, digital-analog converters, real-time clock calendars, LCD drivers, sensors, audio chips, and even other processors. The range of components that support SPI is large and growing all the time.

Unlike a standard serial port, SPI is a synchronous protocol in which all transmissions are referenced to a common clock, generated by the *master* (processor). The receiving peripheral (*slave*) uses the clock to synchronize its acquisition of the serial

INTEL EXHIBIT 1008 Page 102
IPR2023-00787

bit stream. Many chips may be connected to the same SPI interface of a master. A master selects a slave to receive by asserting the slave's chip select input. A peripheral that is not selected will not take part in a SPI transfer.

SPI uses four main signals: *Master Out Slave In* (**MOSI**), *Master In Slave Out* (**MISO**), *Serial CLocK* (**SCLK** or **SCK**) and *Chip Select* ($\overline{\textbf{CS}}$) for the peripheral. Some processors have a dedicated chip select for SPI interfacing called *Slave Select* ($\overline{\textbf{SS}}$).

MOSI is generated by the master and is received by the slave. On some chips, **MOSI** is labeled simply as *Serial In* (**SI**) or *Serial Data In* (**SDI**). **MISO** is produced by the slave, but its generation is controlled by the master. **MISO** is sometimes known as *Serial Out* (**SO**) or *Serial Data Out* (**SDO**) on some chips. The chip select to the peripheral is normally generated by simply using a spare I/O pin of the master. Figure 7-1 shows a microprocessor interfaced to a peripheral using SPI.

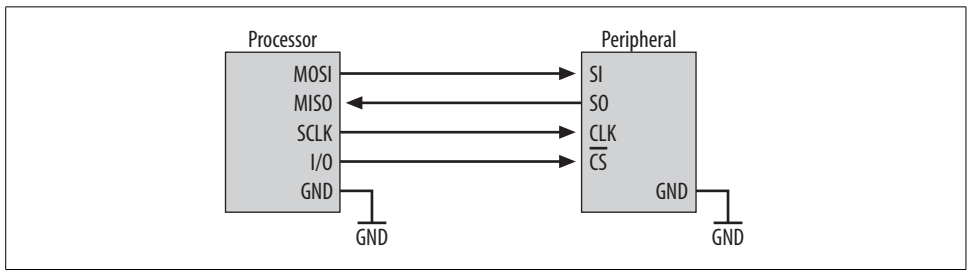


Figure 7-1. Basic SPI interface

Both masters and slaves contain a serial shift register. The master starts a transfer of a byte by writing it to its SPI shift register. As the register transmits the byte to the slave on the **MOSI** signal line, the slave transfers the contents of *its* shift register back to the master on the **MISO** signal line (Figure 7-2). In this way, the contents of the two shift registers are exchanged. Both a write and a read operation are performed with the slave simultaneously. SPI can therefore be a very efficient protocol.

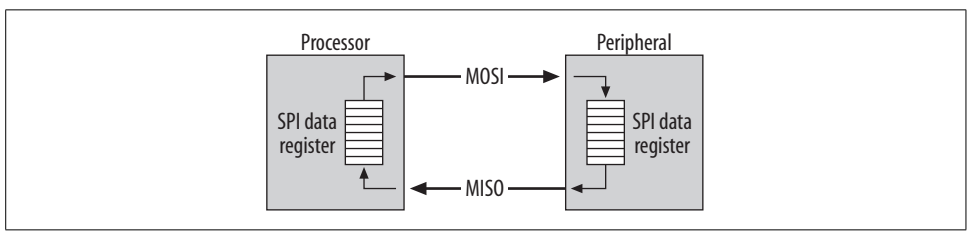


Figure 7-2. SPI transmission

If only a write operation is desired, the master just ignores the byte it receives. Conversely, if the master just wishes to read a byte from the slave, it must transfer a dummy byte in order to initiate a slave transmission.

INTEL EXHIBIT 1008 Page 103
IPR2023-00787

Some peripherals can handle multiple byte transfers, where a continuous stream of data is shifted from the master. Many memory chips with SPI interfaces work this way. With this type of transfer, the chip select for the SPI slave must remain low for the entire duration of the transmission. For example, a memory chip might expect a "write" command to be followed by four address bytes (starting address), then the data bytes to be stored. A single transfer may involve the shifting of a kilobyte or more of information.

Other slaves need only a single byte (for example, a command byte for an analog-digital converter), and some even support being *daisy-chained* together (Figure 7-3).

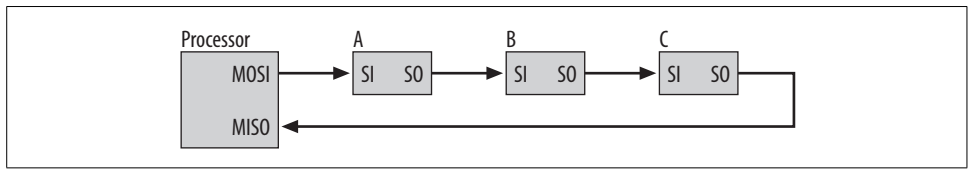


Figure 7-3. Daisy-chaining three SPI devices

In this example, the master processor transmits three bytes out of its SPI interface. The first byte is shifted into slave A. As the second byte is transferred to slave A, the first byte is shifted out of slave A and into slave B. Similarly, as the third byte is shifted into slave A, the second byte is shifted into slave B, and the first byte is shifted into slave C. If the master wishes to read a result from slave A, it must again transfer a three-byte (dummy) sequence. This will move the byte from slave A into slave B, then into slave C, and finally into the master. In the process, the master also receives bytes from slave C and slave B in turn.

Note that daisy chaining won't necessarily work with all SPI devices, especially ones that require multibyte transfers (such as memory chips). Again, it's a case of checking the slave chips' datasheets carefully to determine what you can and can't do. If the datasheet doesn't explicitly mention daisy chaining, then it's a fair bet the device doesn't support it.

SPI has four modes of operation, depending on clock polarity and clock phase. For low clock polarity, the clock (**SCK**) is low when idle and toggles high during a transfer. When configured for high clock polarity, the clock is high when idle and toggles low during a transfer.

The two clock phases are known as *clock phase zero* and *clock phase one*. For clock phase zero, **MOSI** and **MISO** outputs are valid on the rising edge of the clock (**SCK**) if the clock polarity is low (Figure 7-4). If the clock polarity is high, these outputs are valid on the falling edge of **SCK**, for clock phase zero (Figure 7-5). The "X" bit output on **MISO** is an undefined extra bit and is a consequence of the SPI interface. You don't need to worry about it, as the SPI interfaces ignore it.

INTEL EXHIBIT 1008 Page 104
IPR2023-00787

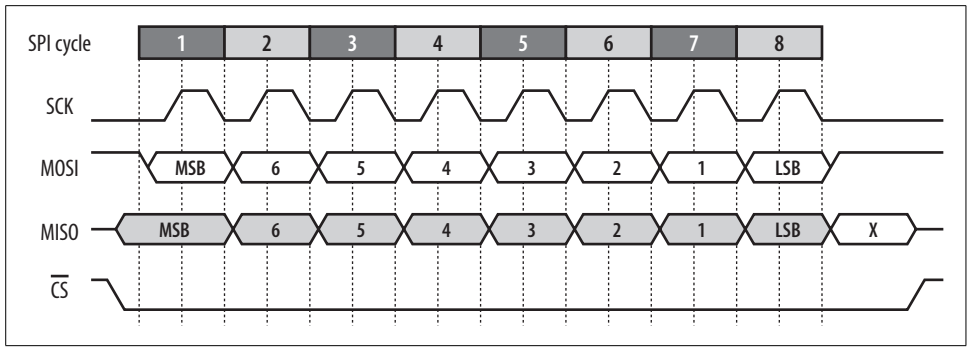


Figure 7-4. SPI timing with clock polarity low and clock phase zero

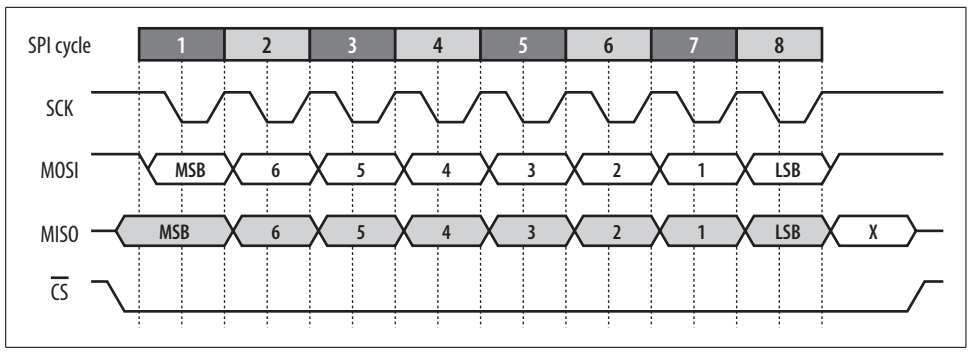


Figure 7-5. SPI timing with clock polarity high and clock phase zero

Conversely, for clock phase one, the opposite is true. **MOSI** and **MISO** are valid on the falling edge of the clock if clock polarity is low (Figure 7-6). They are valid on the rising edge of the clock if the clock polarity is high (Figure 7-7).

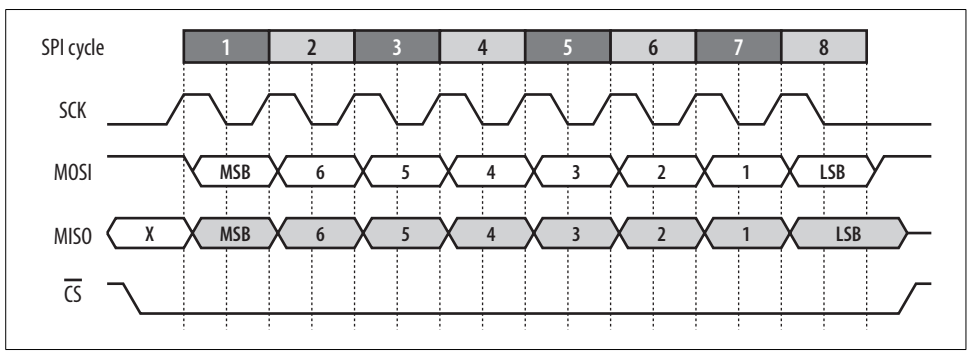


Figure 7-6. SPI timing with clock polarity low and clock phase one

INTEL EXHIBIT 1008 Page 105
IPR2023-00787

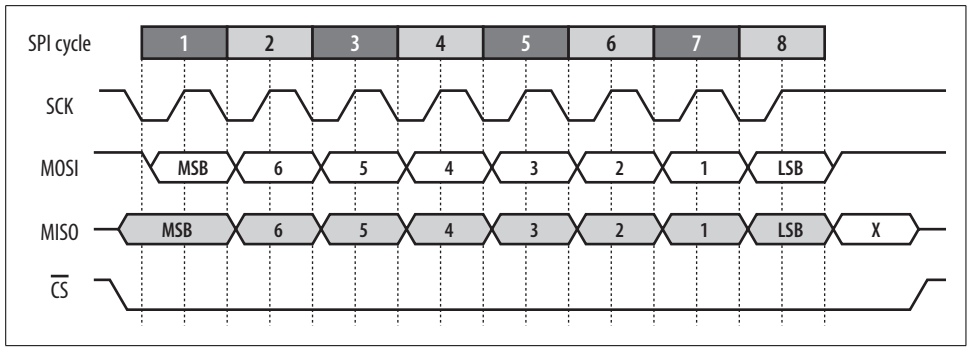


Figure 7-7. SPI timing with clock polarity high and clock phase one

SPI-Based Clock/Calendar

There is a wide variety of SPI devices available, and we'll be looking at several in the coming chapters. In the meantime, to see how a SPI interface is used to add a peripheral to a microcontroller, let's look at interfacing a processor to a clock/calendar chip. Such chips contain an oscillator module driven by a crystal, just like a processor. The oscillator module ticks over internal counters that track milliseconds, seconds, minutes, hours, days, months, and years. They are specifically designed to provide accurate timekeeping, and many have additional functions such as an "alarm" (whereby the processor is interrupted at a specific time) and a watchdog. Some also include voltage monitoring, such that the clock chip may act as a system monitor, alerting the processor should the power supply be wavering. There are a number of clock chips available (and not all are interfaced using SPI). For this example, we will use the Maxim DS1305.

The way in which we interface the clock chip to a processor is virtually identical for all other SPI devices. Some chips with SPI interfaces have special requirements, but most are very simple and straightforward. This makes SPI a very useful interface that makes increasing system functionality trivial.

The Maxim DS1305 Real-Time Clock (RTC) provides timekeeping services and tracks seconds, minutes, hours, day of the month, month, day of the week, and year. It knows which months have 30 days and which have 31. It even automatically adjusts for leap years, up to the year 2100. It can generate two interrupts to the microcontroller for time-of-day alarms. These alarms can be used to trigger a regular system event, such as a backup or user notification.

The DS1305 can run off two separate power sources and supports battery backup of its internal state. The chip can use a power supply in the range of 2 V to 5.5 V, allowing it to be powered from a variety of sources. It also has 96 bytes of static RAM, used for parameter storage. You could use the RAM for holding variables indicating system mode, secure password storage, or even authorization codes for your embedded software, just as desktop software does.

INTEL EXHIBIT 1008 Page 106
IPR2023-00787



If you are producing commercial embedded systems and have problems with late-paying customers, you can use this RAM to hold a license number. When you ship the system, you design it to work for perhaps 45 days before shutting down. When your customer pays her bill, and you supply her with the right magic number, the system comes back to life again. The system stores the license number in the RAM of the RTC and from then on works as normal.

The RAM, like the timekeeping function, is battery-backed, and so its contents will be retained for the life of the battery. This can be up to 10 years, depending on the battery chosen. Thus, the contents of the internal parameter RAM will probably last for the expected operational lifespan of an embedded system.

The DS1305 is versatile in the way it can be powered. It has three power-supply inputs—**VCC1**, **VCC2**, and **VBAT**—from which it can choose to draw power. **VCC1** is the primary supply input and is connected directly to the system's power supply. When the computer is up and running, the DS1305 draws its current from this source. **VCC2** is the secondary power source, and this can be a rechargeable battery. **VBAT** is the third power source and is for nonrechargeable batteries.

There are three, and only three, possible configurations for powering the DS1305, and it is important for correct operation that the power inputs are appropriately driven. Figure 7-8 shows the DS1305 powered by a primary DC supply connected to **VCC1** and a secondary, nonrechargeable battery connected to **VBAT**. (To keep the diagram simple, only the power pins are shown. We'll look at the data interface in a moment.) For this configuration, **VCC2** is unused and must be connected to **GND**. When **VCC1** falls below a given threshold voltage (the primary power source has failed), the internal memory and registers of the DS1305 become write-protected to prevent them from being corrupted by a failing microprocessor.

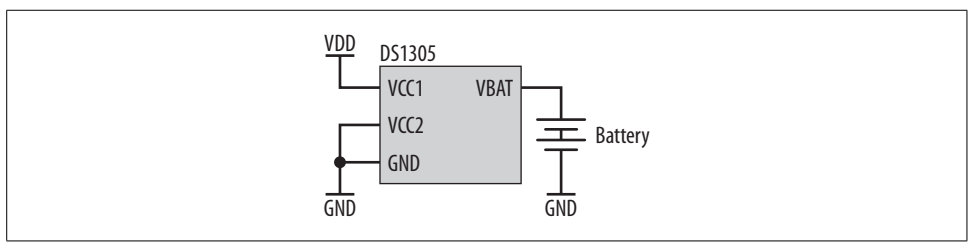


Figure 7-8. Using the DS1305 with a nonrechargeable battery

If the secondary power source is a rechargeable battery, then the DS1305 may be wired as shown in Figure 7-9. When using a rechargeable battery on **VCC2**, **VBAT** must be connected to **GND**. When the device is used in this mode, there is no automatic write protection for the DS1305 if **VCC1** fails.

INTEL EXHIBIT 1008 Page 107
IPR2023-00787

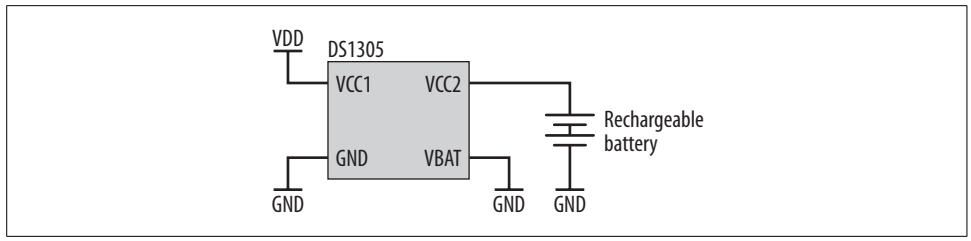


Figure 7-9. Using the DS1305 with a rechargeable battery

Finally, the DS1305 may be used with only a battery as its primary power source and no backup power supply. This is shown in Figure 7-10. For this configuration, both **VCC1** and **VBAT** are connected to ground, while the battery is connected to **VCC2**.

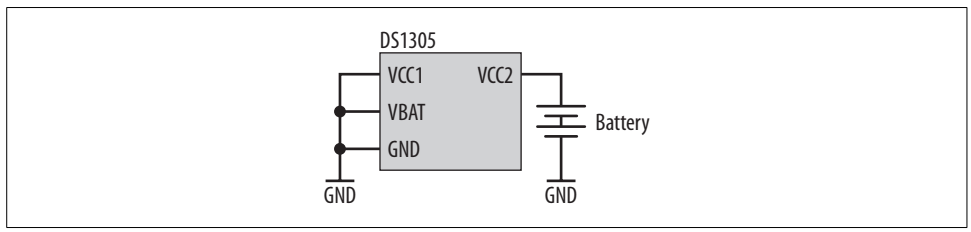


Figure 7-10. Using the DS1305 with a battery as its only power source

Using the DS1305 is very simple. The schematic showing a DS1305 interfaced to a microcontroller is shown in Figure 7-11.

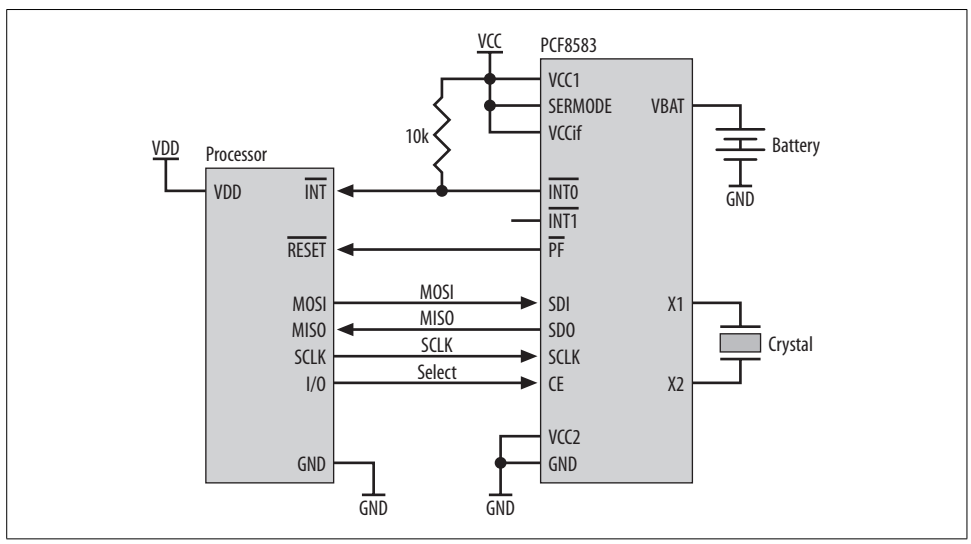


Figure 7-11. A DS1305 RTC interfaced to a microcontroller

INTEL EXHIBIT 1008 Page 108
IPR2023-00787

The serial interface of the DS1305 can operate as either a SPI port or a three-wire* port. The input **SERMODE** (SERial MODE) selects which serial mode to use. Connecting **SERMODE** to the power supply selects SPI operation. Connecting **SERMODE** to **GND** selects three-wire operation. (For three-wire operation, **SDO** and **SDI** are tied together.) The connection to a microcontroller's SPI port is straightforward, with **MOSI**, **MISO**, **SCLK**, and a chip select, as we've seen previously. There is one important difference, though, for the DS1305. It has an active-high **CE** (Chip Enable), rather than the more common active-low chip selects of other SPI devices. Therefore, the processor's I/O line driving **CE** must be low when the device is not selected and high when the device is selected.

The DS1305 has a special Power Fail (**$\overline{\text{PF}}$**) output that is asserted low when the primary power source **VCC1** falls below the secondary power source (**VCC2** or **VBAT**). This can be used to alert the processor of the power fail (by using it as an interrupt) or to stop the processor (by connecting it to the processor's **RESET**). This is used to prevent a failing processor from corrupting devices as the power dies. If you don't require a power-fail notification, **$\overline{\text{PF}}$** may be left unconnected.

The input **VCCif** (**VCC** for the interface logic) selects the output voltage levels of **SDO** and **$\overline{\text{PF}}$**. Since the DS1305 can be used in both 5 V and 3.3 V systems, this input allows the output levels of these pins to be set to the appropriate high voltage. **VCCif** is just connected to the system's power supply. Thus, for a 5 V system, **VCCif** is 5 V, and the outputs of the DS1305 are also 5 V. Similarly, for a 3.3 V system, **VCCif** is 3.3 V and so are the outputs.

Finally, the DS1305 has two interrupt outputs, **$\overline{\text{INT0}}$** and **$\overline{\text{INT1}}$**. These may be used to interrupt the processor when a DS1305 alarm function triggers. As the interrupt outputs are open-drain, they each require a 10k resistor to pull them high when they are inactive. If one or both of the interrupts are not required, just leave them unconnected. Only **$\overline{\text{INT0}}$** is used in our example, and so **$\overline{\text{INT1}}$** is safely ignored.

Finally, the DS1305 has two crystal inputs, **X1** and **X2**. A 32.768 kHz watch crystal is connected across these pins, providing the timing source for the internal clock.

So that is the DS1305, a versatile little chip that can provide timekeeping for your embedded system. It's easy to use, and the programming information for it is contained in the device's datasheet.

SPI-Based Digital Potentiometer

Let's look at another simple SPI example. This time, we will interface a digital potentiometer to a microprocessor. We looked at analog potentiometers in Chapter 4.

* Developed by National Semiconductor, three-wire, also known as *MicroWire*, is very similar to SPI and is found is some microcontrollers and DSP processors. Unlike SPI, which has separate data lines for reading and writing, three-wire uses a common bidirectional data line.

INTEL EXHIBIT 1008 Page 109
IPR2023-00787

Now, a standard pot is manually adjusted. It will either have a knob attached (as in a volume control or brightness adjustment), or it will have a small notch for screwdriver adjustment. Wouldn't it be great if your microprocessor could adjust the pots in your analog circuits, under software control? That way, your application software could adjust the brightness of the display or change the volume of the sound system. Well, by using a *digital potentiometer*, you can do just that. Televisions, computer monitors, and stereos with internal embedded controllers use digital pots to adjust settings such as volume. When you hit a volume button on a remote control, the TV or stereo adjusts the settings of digital pots, which are part of the amplifiers driving the speakers.

Figure 7-12 shows an Analog Devices AD5203 digital potentiometer with a SPI interface. This chip has four potentiometers, all of which may be adjusted under software control. Each pot has 64 possible positions, and versions of the chip are available with either 10 kΩ or 100 kΩ impedances. For higher resolution, the pin-compatible AD8403 has a possible 256 settings, also configurable through a SPI interface.

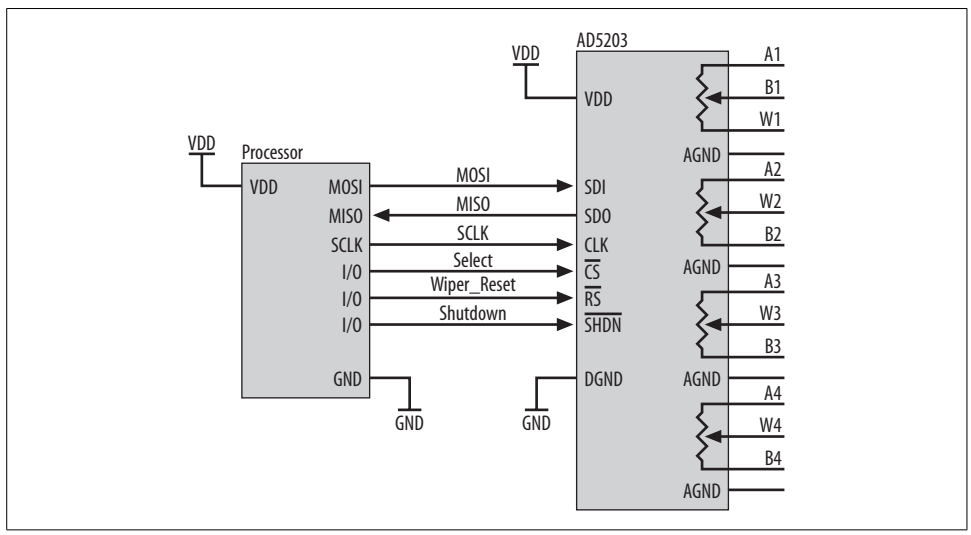


Figure 7-12. Interfacing a digital potentiometer to a processor using SPI

The AD5203 has a Serial Data Input (**SDI**), which is connected to the processor's **MOSI** output. Similarly, the device's Serial Data Output (**SDO**) is connected to **MISO**. The AD5203's clock input (**CLK**) is positive-edge triggered midway through each SPI cycle, which means that any processor communicating with it must use high clock polarity and clock phase one on **SCLK**. The Chip Select ($\overline{\textbf{CS}}$) of the AD5203 may be driven by a processor digital I/O line. The AD5203 has two other inputs, Shutdown ($\overline{\textbf{SHDN}}$) and Reset ($\overline{\textbf{RS}}$). $\overline{\textbf{SHDN}}$ places the device in low-power mode, and $\overline{\textbf{RS}}$ resets the potentiometer wipers to their midpoint. Both of these inputs

INTEL EXHIBIT 1008 Page 110
IPR2023-00787

may also be driven by a processor I/O line, or, if their functionality is not needed, they may be simply tied high using 10 kΩ pull-up resistors.

The potentiometers within the AD5203 are used as any other pots would be. The **A** and **B** terminals connect to either end of the internal resistors, and the position of the wiper (**W**) is adjusted under software control.

The AD5203 has several ground connections. **DGND** is the digital ground for the SPI interface and control logic of the chip. The **AGND**s are the analog grounds of the internal potentiometers, and they should all be connected to **DGND** at a single point.

The datasheet for the AD5203 provides the control codes needed to configure the chip, and its use is simple and straightforward.

Adding Nonvolatile Data Memory with SPI

The internal memory of microcontrollers is very small, and their data storage capabilities are severely limited. We're now going to look at how you can increase the storage capacity of your embedded system by adding an Atmel AT45DB161 2M serial DataFlash using SPI. These chips are commonly used in low-cost digital cameras and answering machines. You could also use this flash chip as a virtual disk drive in your embedded system.

Most other flash chips have a bus interface, but the AT45DB161 has a serial interface, making it well suited for use with small microcontrollers. The AT45DB161 is a 2M chip, but you can get similar chips in capacities ranging from 512K to 32M. They all use the same (or similar) SPI interface, so the same design works for all. (Note, however, that their pinouts and physical packages vary, so one chip will not mount onto a circuit board design for another.)

The chip consists of an array of flash memory, organized as individual pages of 528 bytes each, and two RAM buffers, also 528 bytes each (Figure 7-13). To write data into the main flash array, the processor must first write data into one of the buffers and then issue a command to write that buffer into the array. A processor can read the contents of either of the buffers, transfer a flash page to the buffers, or read from the flash array directly. The operation of the buffers is independent, and one buffer may be accessed by the processor (via SPI) while the contents of the other buffer are being written into the flash array.

The flash supports numerous commands for writing to and reading from the buffers, writing the buffers to the main array, and transferring an array page back to a buffer. The Atmel datasheet has full details of the software protocols and command set.

There are a few things to note about the internal architecture and the flash array. The first is that one 528-byte page of the flash array is not contiguous with the next. In

INTEL EXHIBIT 1008 Page 111
IPR2023-00787

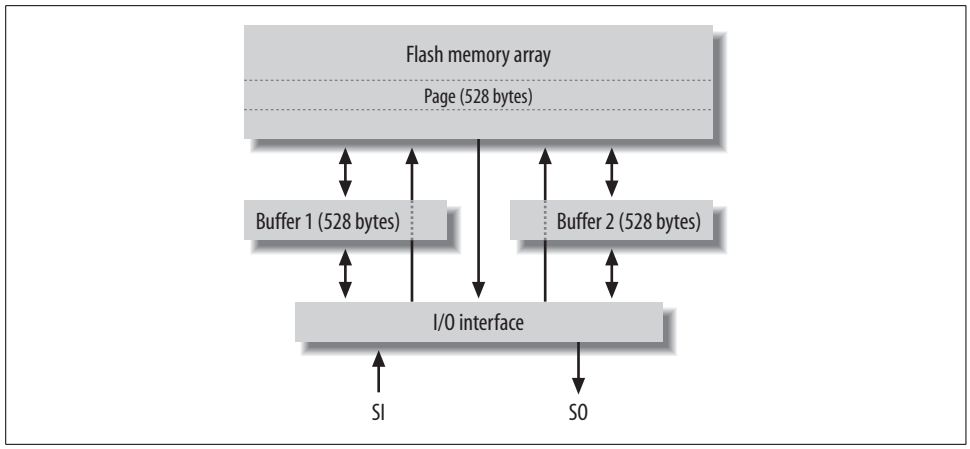


Figure 7-13. AT45DB161 internal architecture

other words, if you are using a pointer in your software to track the current location in the memory, you can't just increment it from the end of one page and expect it to be pointing to the next. Every 528 bytes (and it's a strange number), you have to leap forward to the next page. Think of it as pages of 528 bytes with big gaps in between.

The second catch with this memory is that it has a lifetime of only 1,000 write cycles per page. Most flash technologies (and there are several different types) support 100,000 write cycles or better, and you can normally exceed this limit and the device will keep working reliably for you. This isn't the case with the AT45DB161. Once the 1,000-write limit is exceeded, memory locations will start failing on you. The chip will read existing data back correctly, but new pages will not write successfully. Depending on the application, this limit may not be a problem. I've used this particular chip in my design for long-duration data loggers. These machines are deployed for yearlong deployments, collecting (and compressing) data and storing it away in the flash chip. The logger gradually builds a page image in one of the buffers before storing it to the array in a single write. Since during a deployment, a page will be written only once (and then the logger will move on to the next page), the 1,000-write limitation isn't a problem. It would take 1,000 deployments before the chip would fail. However, if you're using the chip for variable storage and are modifying the flash pages on a byte-by-byte basis, you're in trouble. Individually changing 528 bytes within a page counts as 528 writes. So do that twice to a page, and suddenly you're over the limit. Therefore, this flash is well suited to some applications and not others.

The basic design for using an AT45DB161 is shown in Figure 7-14.

On the left of the chip are the SPI interface connections, **MOSI**, **MISO**, **SCK**, and a chip select (**FLASH**). The chip will support SPI transfers at up to 20 MHz, so the SPI interface can be run very fast indeed. On the right of the chip is the power supply, **VDD**, which is decoupled to ground using a 100 nF capacitor. The AT45DB161

INTEL EXHIBIT 1008 Page 112
IPR2023-00787

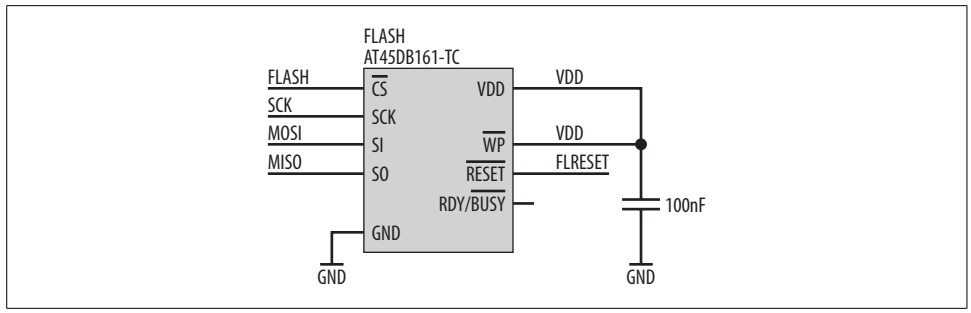


Figure 7-14. 2M Serial DataFlash

requires a power supply in the range 2.5 V to 3.6 V. However, its logic inputs are *5 V tolerant*, meaning this chip can be used in systems with mixed power supplies. In other words, while this chip requires a 3 V power supply, it can be directly interfaced to a processor with a 5 V supply (and 5 V logic levels). The AT45DB161 has a write-protect pin (**WP**), which, when driven low, prevents the contents of the flash from being modified. If you don't require write protection, simply tie this input high, as shown in the schematic. The flash also has a **RESET** input so that the chip can be manually reset under software control. The flash incorporates an inbuilt power-on reset that will put the device into a known state, and therefore a "manual" reset at power up should be unnecessary. However, I've found that the internal power-on reset generator is somewhat finicky and doesn't always kick in as it should. Under such circumstances, the flash fails to enter a known state and is unusable in the system. Therefore, I have found it good practice to give the processor control of the flash's reset. As part of the processor's initialization routines executed in its reset firmware, I get the processor to reset the flash, nudging it into reality. It's a simple thing, but it makes all the difference for a reliable system. Pin 1 is a status output (**RDY/BUSY**) indicating whether the device is ready or if it is still completing an internal operation.

The connections for interfacing this memory chip to an Atmel 90S4434 AVR processor are shown in Figure 7-15. The AVR portion of the schematic is no different from the examples we have seen previously. That's the nice thing about simple interfaces such as SPI. They form little subsystem modules that "bolt together" like building blocks. Start with the basic core design and just add peripherals as you need them. The schematic also shows decoupling capacitors for the power supplies, the crystal oscillator for the processor, and a pull-up resistor for **RESET**. Pin 41 (**PB1**) is used as a "manual" (processor-controlled) reset input to the flash.

Adding a Parameter Memory Using SPI

We saw in the previous section how to add a large-capacity serial flash for data storage. It is often useful to use nonvolatile memory to hold system parameters, a way of preserving important variables during periods of no power. But the AT45DB161

INTEL EXHIBIT 1008 Page 113
IPR2023-00787

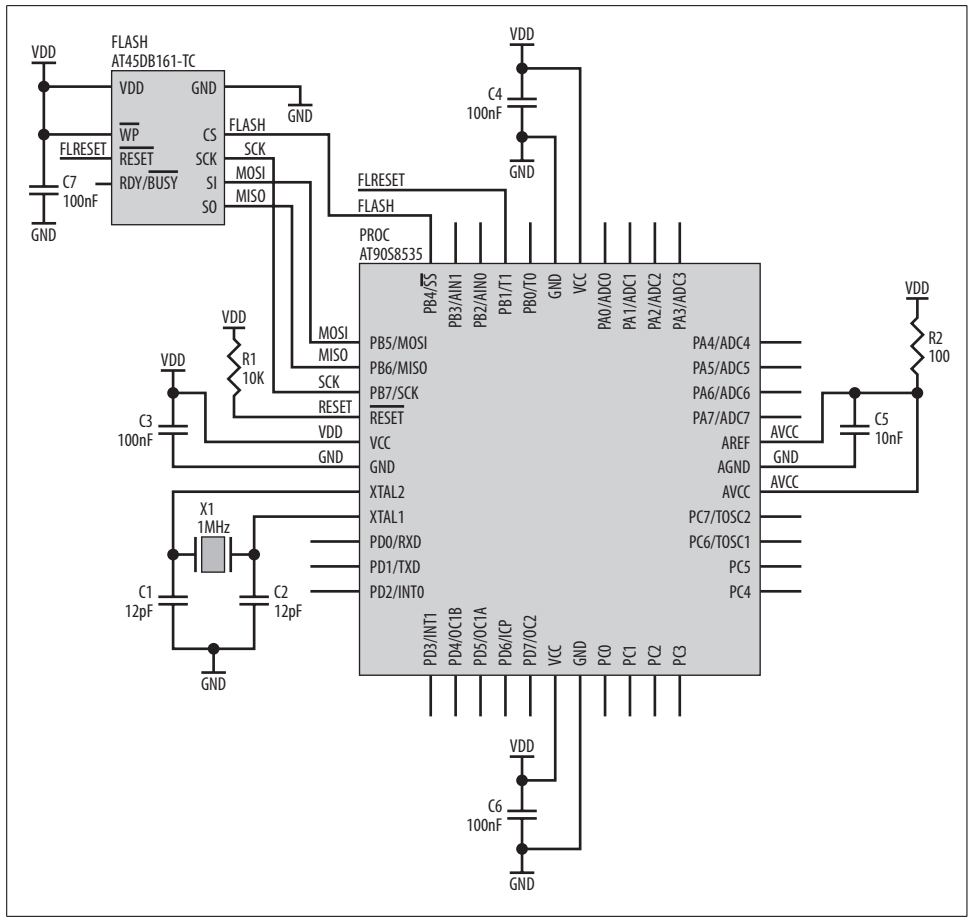


Figure 7-15. A 2M DataFlash interfaced to an AT90S4434

DataFlash is just not the device for that task. It is better suited to data recording, and its large capacity is overkill for parameter storage. So, now we're going to look at how you can use SPI to add a small parameter memory (in the form on an EEPROM) to your embedded system. The EEPROM I've chosen is the Atmel AT25640. This device will hold data for at least 100 years without power, and will endure more than one million write cycles (significantly more than an AT45DB161!). As such, your software can happily alter parameter variables without fear of limiting the lifespan of the chip. The AT25640 has only 8K of memory, which might not sound like much. But don't forget, that's 8192 `char` variables, which is more than enough storage space for most parameters. If 8K is too much, there are also versions of the chip with 1K (AT25080), 2K (AT25160), and 4K (AT25320) bytes of memory.

INTEL EXHIBIT 1008 Page 114
IPR2023-00787

The architecture and use of the AT25640 is much simpler than that of the AT45DB161. Full details of the required software protocol are in the Atmel datasheet for this chip.

The schematic for an AT25640 circuit is shown in Figure 7-16.

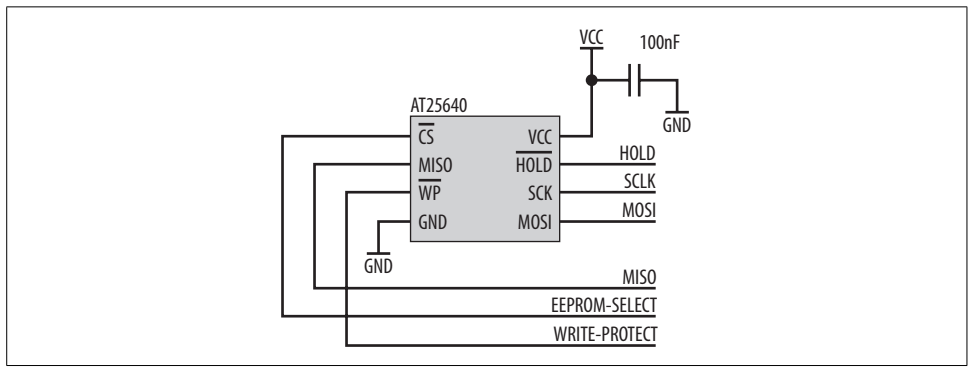


Figure 7-16. Using an AT25640 EEPROM

The interface is standard SPI, and the chip also has a write-protect input and a hold input. Asserting **HOLD** allows the processor to temporarily stall a serial transfer (while it performs other tasks) without terminating the access to the AT25640. And as you might expect, write-protect, when asserted, turns the chip into a read-only device. These control inputs may be driven by programmable I/O lines of the processor. The only other requirement is power (which is decoupled to ground using a 100 nF capacitor) and ground. The chip is available in two types. One will operate from a supply voltage of between 2.7 V and 5.5 V, while the other needs a supply voltage of between 1.8 V and 3.6 V.

INTEL EXHIBIT 1008 Page 115
IPR2023-00787

CHAPTER 15
The AVR Microcontrollers

A really useful engine...
—W.V. Awdry

In this chapter, we'll look at the Atmel AVR processor. Like the PIC, this processor family is a range of completely self-contained computers on chips. They are ideally suited to any sort of small control or monitoring application. They include a range of inbuilt peripherals and also have the capability of being expanded off-chip for additional functionality.

Like the PIC, the AVR is a RISC processor. Of the two architectures, the AVR is the faster in operation and arguably the easier for which to write code, in my personal experience. The PIC and AVR both approach single-cycle instruction execution. However, I find that the AVR has a more versatile internal architecture, and therefore you actually get more throughput with the AVR.

In this chapter, we'll look at the basics of creating computer hardware by designing a small computer based on the AVR *ATtiny15*. We'll also see how you can download code into an AVR-based computer and how it can be reprogrammed in-circuit. From there, we'll go on to look at some larger AVR processors, with a range of capabilities.

Later in the chapter, we're going to look at interfacing memory (and peripherals) to a processor using its address, data, and control buses. For most processors, this is the primary method of interfacing, and, therefore, the range of memory devices and peripherals available is enormous. You name it, it's available with a bus interface. So, knowing how to interface bus-based devices opens up a vast range of possibilities for your embedded computer. You can add RAM, ROM (or flash), serial controllers, parallel ports, disk controllers, audio chips, network interfaces, and a host of other devices.

Most small microcontrollers are completely self-contained and do not "bring out" the bus to the external world. In this chapter, we'll take a look at the Atmel AT90S8515 processor. It is the only processor of the AVR family that allows you access to the CPU's buses. But first, let's take a look at the AVR architecture in general.

INTEL EXHIBIT 1008 Page 116
IPR2023-00787

The AVR Architecture

The AVR was developed in Norway and is produced by the Atmel Corporation. It is a Harvard-architecture RISC processor designed for fast execution and low-power consumption. It has 32 general-purpose 8-bit registers (`r0` to `r31`), 6 of which can also act as 3 16-bit index registers (`X`, `Y`, and `Z`) (Figure 15-1). With 118 instructions, it has a versatile programming environment.

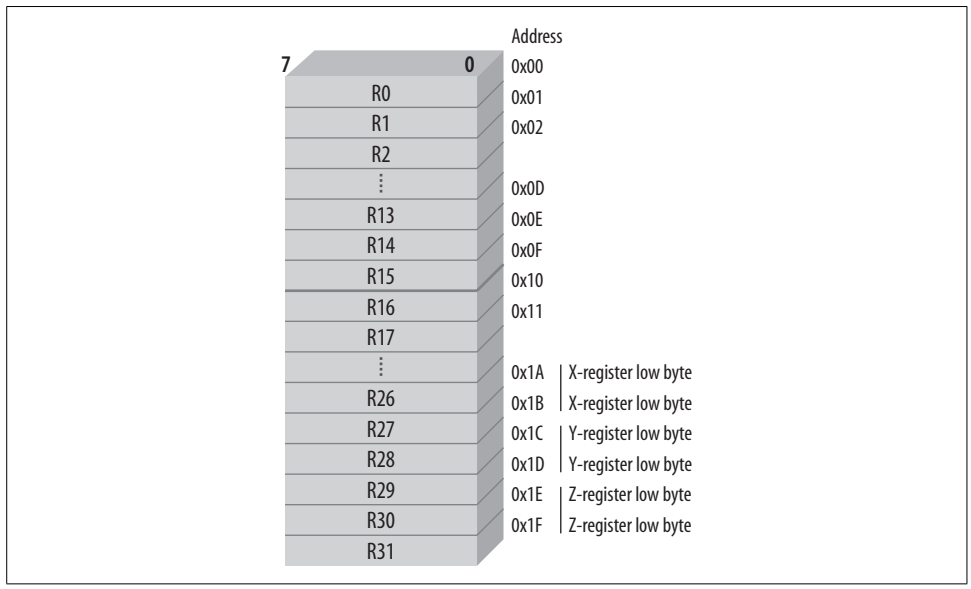


Figure 15-1. AVR registers

In most AVRs, the stack exists in the general memory space. It may therefore be manipulated by instructions and is not limited in size, as is the PIC's stack.

The AVR has separate program and data spaces and supports an address space of up to 8M. As an example, the memory map for an AT90S8515 AVR processor is shown in Figure 15-2.

Atmel provides the following sample C code, which it compiled and ran on several different processors:

```
int max(int *array)
{
  char a;
  int maximum = -32768;

  for (a = 0; a < 16; a++)
    if (array[a] > maximum)
      maximum = array[a];
  return (maximum);
}
```


INTEL EXHIBIT 1008 Page 117
IPR2023-00787

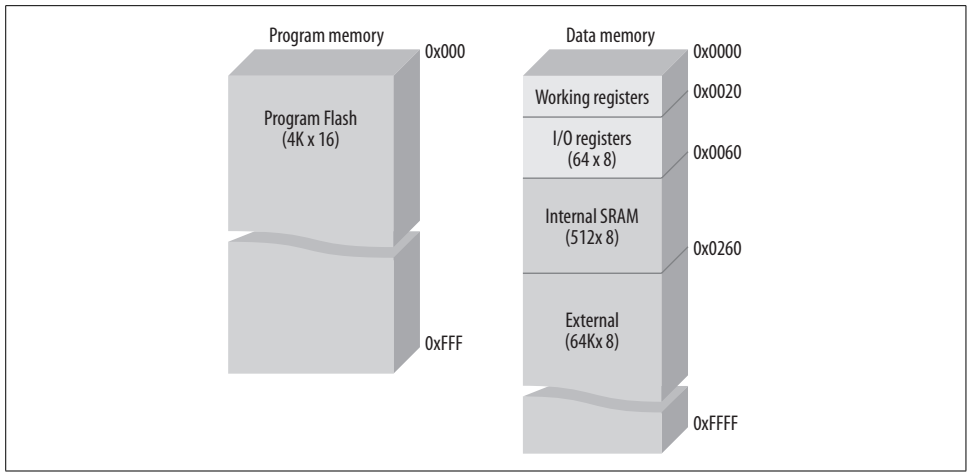


Figure 15-2. Atmel AT90S8515 memory map

The results are interesting (Table 15-1).

Table 15-1. Atmel's comparison of processor speed and efficiency

Processor	Compiled code size	Execution time (cycles)
AVR	46	335
8051	112	9,384
PIC16C74	87	2,492
68HC11	57	5,244

This indicates that, when running at the same clock speed, an AVR is 7 times faster than a PIC16, 15 times faster than a 68HC11, and a whopping 28 times faster than an 8051. Alternatively, you'd have to have an 8051 running at 224 MHz to match the speed of an 8 MHz AVR. Now, Atmel doesn't give specifics of which compiler(s) it used for the tests, and it is certainly possible to tweak results one way or the other with appropriately chosen source code. However, my personal experience is that with the AVR, you certainly do get significantly denser code and much faster execution.

There are three basic families within the AVR architecture. The original family is the *AT90xxxx*. For complex applications, there is the *ATmega* family, and for small-scale use, there's the *ATtiny* family. Atmel also produces large *FPGAs* (*Field-programmable Gate Arrays*), which incorporate an AVR core along with many tens of thousands of gates of programmable logic.

INTEL EXHIBIT 1008 Page 118
IPR2023-00787

For software development, a port of gcc is available for the AVR, and Atmel provides an assembler, simulator, and software to download programs into the processors. The Atmel software is freely available on its web site. The low-cost Atmel development system is a good way of getting started with the AVR. It provides you with the software and tools you need to begin AVR development.

The AVR processors at which we'll be looking are the small ATtiny15, the AT90S8535/AT90S4434, and the AT90S8515.

The ATtiny15 Processor

For many simple digital applications, a small microprocessor is a better choice than discrete logic, because it is able to execute software. It is therefore able to perform certain tasks with much less hardware complexity. You'll see just how easy it is to produce a small, embedded computer for integration into a larger system using an Atmel ATtiny15 AVR processor. This processor has 512 words of flash for program storage and no RAM! (Think of that the next time you have to install some 100 MB application on your desktop computer!) This tiny processor, unlike its bigger AVR siblings, relies solely on its 32 registers for working variable storage.

Since there is no RAM in which to allocate stack space, the ATtiny15 instead uses a dedicated hardware stack that is a mere *three* entries deep, and this is shared by subroutine calls and interrupts. (That fourth nested function call is a killer!) The program counter is 9 bits wide (addressing 512 words of program space), and therefore the stack is also 9 bits wide. Also, unlike the bigger AVRs, only two of the registers (R30 and R31) may be coupled as a 16-bit index register (called "Z").

The processor also has 64 bytes of EEPROM (for holding system parameters), up to five general-purpose I/O pins, eight internal and external interrupt sources, two 8-bit timer/counters, a four-channel 10-bit analog-to-digital converter, an analog comparator, and the ability to be reprogrammed in-circuit. It comes in a tiny 8-pin package, out of which you can get up to 8 MIPS performance. We're not going to worry about most of its features for the time being. That will all be covered in later chapters when we take a look at the I/O features of some larger AVRs. Instead, we're just going to concentrate on how you use one for simple digital control.

Using a small microcontroller such as the ATtiny15 is very easy. The basic processor needs very little external support for its own operation. Figure 15-3 shows just how simple it is.

Let's take a quick run-through of the design (what there is of it). **VCC** is the power supply. It can be as low as 2.7 V or as high as 5.5 V. **VCC** is decoupled to ground using a 0.1 uF capacitor. The five pins, **PB0** through **PB4**, can act as digital inputs or outputs. They can be used to read the state of switches, turn external devices on or

INTEL EXHIBIT 1008 Page 119
IPR2023-00787

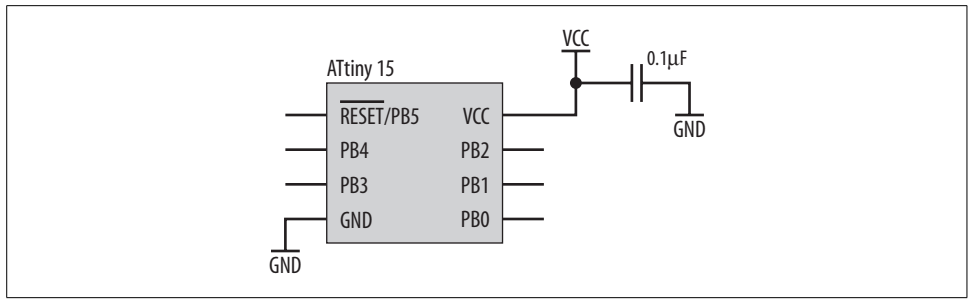


Figure 15-3. A simple AVR computer

off, generate waveforms to control small motors, or even synthesize an interface to simple peripheral chips. The digital I/O lines, **PB0** through **PB4**, get connected to whatever you're using the processor to monitor or control. We'll look at some examples of that later in the chapter.

Finally, one input, **$\overline{\text{RESET}}$**, is left unconnected. On just about any other processor, this would be fatal. Many processors require an external *power-on reset* (*POR*) circuit to bring them to a known state and to commence the execution of software. Some processors have an internal power-on reset circuit and require no external support. Such processors still have a reset input, allowing them to be manually reset by a user or external system. Normally, the reset input still requires a pull-up resistor to hold it inactive. But the ATtiny15 processor doesn't require this. It has an internal power-on reset *and* an internal pull-up resistor. So, unlike most (all) other processors, **$\overline{\text{RESET}}$** on the ATtiny15 may be left unconnected. In fact, on this particular processor, the **$\overline{\text{RESET}}$** pin may be utilized as a general-purpose input (**PB5**) when an external reset circuit is not required. One important point: the normal input protection against higher-than-normal voltage inputs is not present on **$\overline{\text{RESET}}$/PB5**, since it may be raised to +12 V during software download by the program burner. Therefore, if you are using **PB5**, you must take great care to ensure that the input never exceeds VCC by more than 1 V. Failing to do so may place the processor into software-download mode and thereby effectively crash your embedded computer.

The AVR processors (and PICs too) include an internal circuit known as a *brownout detector* (*BOD*). This detects minor fluctuations on the processor's power supply that may corrupt its operation, and if such a fluctuation is detected, it generates a reset and restarts the processor. There is also an additional reset generator, known as a *watchdog*, used to restart the computer in case of a software crash. It is a small timer whose purpose is to automatically reset the processor once it times out. Under normal operation, the software regularly restarts the watchdog. It's a case of "I'll reset you before you reset me." If the software crashes, the watchdog isn't cleared and thus times out, resetting the computer. Processors that incorporate watchdogs normally give software the ability to distinguish between a power-on reset and a

INTEL EXHIBIT 1008 Page 120
IPR2023-00787

watchdog reset. With a watchdog reset, it may be possible to recover the system's state from memory and resume operation without complete re-initialization.

Now the other curious aspect of the above design is that there is no clock circuit. The ATtiny15 can have an external crystal circuit. (On the ATtiny15, **PB3** and **PB4** function as the crystal inputs, **XTAL1** and **XTAL2**). But our design doesn't have a crystal, or even need one. The reason is that this little processor includes a complete internal oscillator (in this case, an RC oscillator) running at a frequency of 1.6 MHz and so requires no external components for its clock. The catch is that RC oscillators are not that stable and have the tendency to vary their frequency as the temperature changes. (The ATtiny15's oscillator can vary between 800 kHz and 1.6 MHz.) Generally, an RC oscillator is not really suitable for timing-critical applications (in which case, you'd use an external crystal instead). But if your ATtiny15 is just doing simple control functions, timing may not be an issue. You can therefore get by with using the internal RC oscillator and save on complexity. Atmel provides an 8-bit calibration register (OSCCAL) in the ATtiny15 that enables you to tune the internal oscillator, thus making it more accurate.

There we have the basic design for an ATtiny15 machine. In essence, it's a very cheap, very small, versatile computer that requires no work for the core design. The only design effort needed is to ensure that the computer will work correctly with the I/O devices to which it is interfaced. If you're going to power the system off a battery, then the capacitor is optional as well! The only component that *must* be there is the processor itself. (And you thought designing computer hardware was going to be hard!)

So, that's the basic AVR computer hardware with minimal components. We'll look at how you download code to it shortly.

That covers the basics of a ATtiny15 system, and it's not that much different from the corresponding PIC12C805 computer. The real differences lie in their internal architectures (and instruction sets) and in the subtleties of their operating voltages and interfacing capabilities. As you can see, there's not a lot of hard work involved in putting one of these little machines into your embedded system.

So far, our computer isn't interfaced to anything. Let's start with something simple by adding a LED to the AVR. The basic technique applies to all microcontrollers with programmable I/O lines as well.

Adding a Status LED

LEDs produce light when current flows through them. Being a diode, they conduct only if the current is flowing in the right direction, from *anode* (positive) to *cathode* (negative). The cathode end of a LED is denoted on a schematic by a horizontal bar. The anode is a triangle.

INTEL EXHIBIT 1008 Page 121
IPR2023-00787

The circuit for a status LED is shown in Figure 15-4. It uses an I/O line of the microcontroller to switch the LED on or off. Sending it low will turn on the LED. Sending it high will turn off the LED, as we'll soon see. The resistor (R) is used to limit the current sinking into the I/O line, as we shall also see shortly.

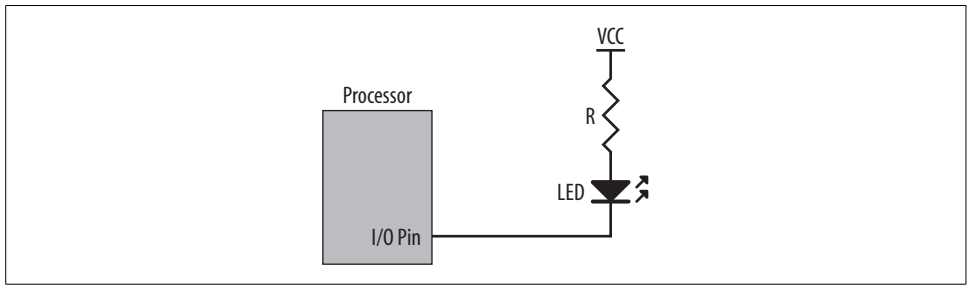


Figure 15-4. Status LED

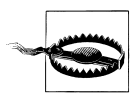
When conducting (and thereby producing light), LEDs have a *forward voltage drop*, meaning that the voltage present at the cathode will be less than that at the anode. The magnitude of this voltage drop varies between different LED types, so check the datasheet for the particular device you are using.

The output low voltage of an ATtiny15 I/O pin is 0.6 V when the processor is operating on a 5 V supply and 0.5 V when operating on a 3 V supply. Let's assume (for the sake of this example) that we are using a power supply (**VCC**) of 5 V, and the LED has a forward voltage drop of 1.6 V. Now, sending the output low places the LED's cathode at 0.6 V. This means the voltage difference between **VCC** (5 V) and the cathode is 4.4 V. If the LED has a voltage drop of 1.6 V, this means the voltage drop across the resistor is 2.8 V (5 V – 1.6 V – 0.6 V = 2.8 V).

Now, from the datasheet, the digital I/O pins of an AVR can sink up to 20 mA if the processor is running on a 5 V supply. We therefore have to limit the current flow to this amount, and this is the purpose of the resistor. If the resistor has a voltage difference across it of 2.8 V (as we calculated) and a current flow of 20 mA, then from Ohm's Law we can calculate what value resistor we need to use:

```
R = V / I
  = 2.8 V / 20 mA
  = 140 Ω
```

The closest available resistor value to this is 150 **Ω**, so that's what we'll use. (This will give us an actual current of 18.6 mA, which is fine.)



The AVR can sink 20 mA per pin when operating on a 5 V supply. However, the amount of current it can sink decreases with supply voltage. When running on a 2.7 V supply, the AVR can sink only 10 mA. As always, it's important to read the datasheets carefully.

INTEL EXHIBIT 1008 Page 122
IPR2023-00787

The next question is, how much power will the resistor have to dissipate? In other words, how much energy will it use in dropping the voltage by 2.9 V? This is important because if we try to pump too much current through the resistor, we'll burn it out. We thus need to choose a resistor with a power rating greater than that required. Power is calculated by multiplying voltage by current:

```
P = V * I
  = 2.8 V * 20 mA
  = 0.056 Watts = 56 mW
```

That's negligible, so the resistor value we need for R is 150 **Ω** and 0.0625 W. (0.0625 W is the lowest power rating commonly available in resistors.)

So, what happens when the I/O line is driven high? The AVR I/O pins output a *minimum* of 4.3 V when high (and using a 5 V supply). With the output high, the voltage at the LED's cathode will be at least 4.3 V, so the voltage difference between the cathode and **VCC** will be only 0.7 V (or less). But the forward voltage drop of the LED is 1.6 V. Thus, there is not enough voltage across the LED to turn it on.

In this way, we can turn the LED on or off using a simple digital output of the processor. We have also seen how to calculate voltages and currents. It is very important to do this with every aspect of a design. Ignoring it can result in a nonfunctioning machine or, worse, charred components and that wafting smell of burning silicon.

We've just seen how to use the digital outputs of the AVR to control a LED. This will work with any device that uses less than 20 mA. In fact, for low-power components, such as some sensors, it is possible to use the AVR's output to provide direct power control, just as we provided direct power control for the LED. In battery-powered applications, this can be a useful technique for reducing the system's overall power consumption.

Switching Analog Signals

We can also use the digital I/O lines of the processor to control the flow of analog signals within our system. For example, perhaps our embedded computer is integrated into an audio system and is used to switch between several audio sources. To do this, we use an analog switch such as the MAX4626, one for each signal path. This tiny component (about the size of a grain of rice in the surface-mount version) operates from a single supply voltage (as low as 1.8 V and as high as 5.5 V). It also incorporates inbuilt overload protection to prevent device damage during short circuits. The schematic showing a MAX4626 interfaced to an ATtiny15 AVR is shown in Figure 15-5. Driving the AVR's output (**PB2**) high turns on the MAX4626 and makes a connection between **NO** and **COM**. Sending **PB2** low breaks the connection. In this way, the MAX4626 can be used to connect an output to an input, under software control.

INTEL EXHIBIT 1008 Page 123
IPR2023-00787

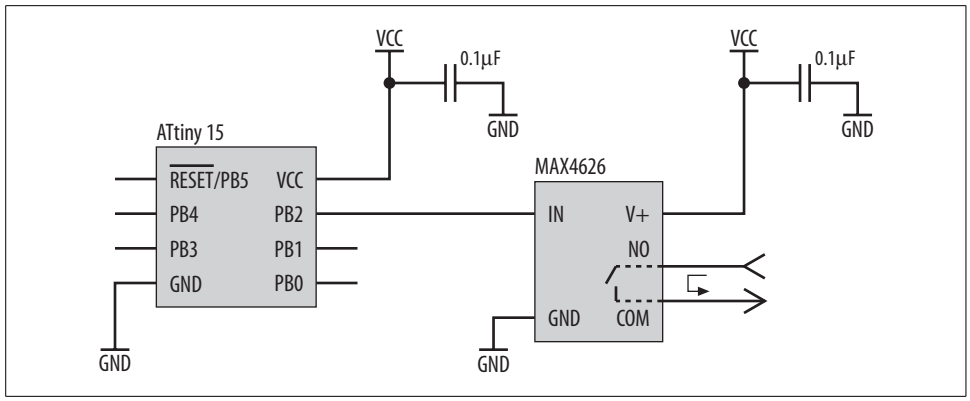


Figure 15-5. Switching an analog signal

The question is: will it work with an AVR? When operating on a 5 V supply, the input to the MAX4626 (pin 4, **IN**) requires a logic-low input of less than 0.8 V and a logic-high input of at least 2.4 V. The AVR's logic-low output is 0.6 V or less, and its logic-high output is a minimum of 4.3 V. So, the AVR's digital output voltages match the requirements of the MAX4626. As for current, the MAX4626 needs to sink or source only a miniscule 1 µA. For an AVR, this is not a problem.

If the MAX4626 doesn't meet your needs, MAXIM and other manufacturers produce a range of similar devices with varying characteristics. There's bound to be something that meets your needs.

The schematic in Figure 15-6 includes a push-button connected to **PB3**, where **PB3** is acting as a digital input. Now, there are a couple of interesting things to note about this simple input circuit. The first is that there is no external pull-up resistor attached to **PB3**. Normally, for such a circuit, an external pull-up resistor is required to place the input into a known state when the button is open (not being pressed). The pull-up resistor takes the input high, except when the button is closed and the input is connected directly to ground. The reason we can get away with not having an external pull-up resistor is that the AVR incorporates internal pull-up resistors, which may be enabled or disabled under software control.

The second interesting thing to note is that there is no *debounce* circuitry between the button and the input. Any sort of mechanical switch (and that includes a keyboard key) acts as a little inductor when pressed. The result is a rapid ringing oscillation on the signal line that quickly decays away (Figure 15-7).

So, instead of a single change of state, the resulting effect is as if the user had been rapidly hammering away on the button. Software written to respond to changes in this input will register the multiple pulses, rather than the single press the user intended. Removing these transients from the signal is therefore important and is known as *debouncing*. Now, there are several different circuits you could include that will cleanly remove the ringing. But here's the thing: you don't always need to!

INTEL EXHIBIT 1008 Page 124
IPR2023-00787

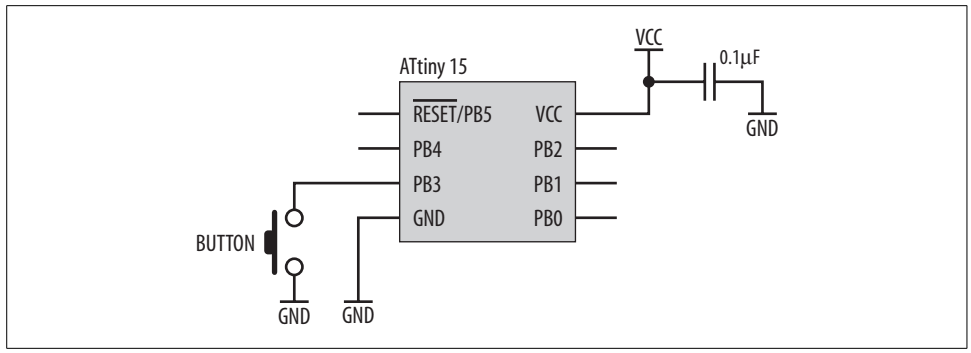


Figure 15-6. Push-button input

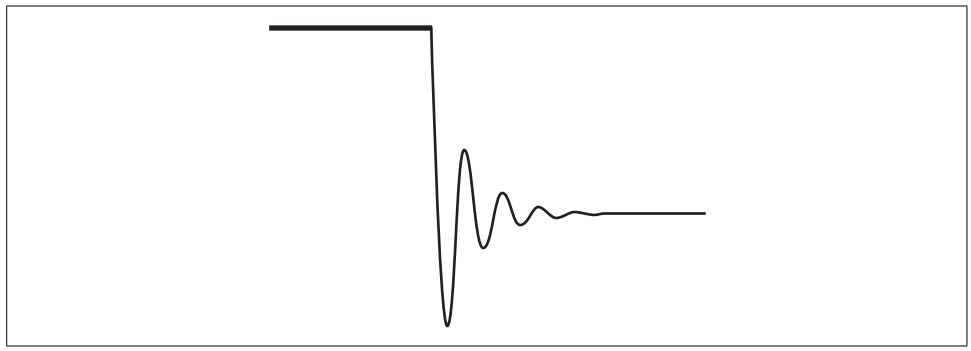


Figure 15-7. Signal bounce

When a user presses a button, he will usually hold that button closed for at least half a second, maybe more, by which time the ringing has died away. The problem can therefore be solved in software. The software, when it first registers a low on the input, waits for a few hundred milliseconds and then samples the input again (perhaps more than once). If it is still low, then it is a valid button press, and the software responds. The software then "re-arms" the input, awaiting the next press. Debouncing hardware does become important, however, if the button is connected to an interrupt line or reset.

So far, we have seen how to use the AVR to control digital outputs and read simple digital inputs. The astute among you may ask, "When looking at the previous two circuits, why do we need the processor?" After all, it is certainly possible to connect the button directly to the input of the MAX4626. Of what use can the processor be? Well, we've already seen one use. The processor can replace debounce circuitry on the input. Since it has internal memory and the ability to execute software, the processor can also keep track of system state (and mode), monitor various inputs in relation to each other, and provide complicated control sequencing on the outputs. In short, the inclusion of a microprocessor can reduce hardware complexity while increasing system functionality. They can be very useful tools. With more advanced

INTEL EXHIBIT 1008 Page 125
IPR2023-00787

processors, and with more diverse I/O, the functionality and usefulness of an embedded computer can be significant.

Downloading Code

The AVR processors use internal flash memory for program storage, and this may be programmed in-circuit or, in the case of socketed components, out of circuit as well. The AVR processors are reprogrammed via a SPI port on the chip. Even AVR processors such as the ATtiny15, which do not have a SPI interface for their own use, still incorporate a SPI port for reprogramming. The pins **PB0**, **PB1**, and **PB2** take on SPI functions (**MOSI**, **MISO**, and **SCK**) during programming.

VCC can be supplied by the external programmer downloading the code. For programming, **VCC** *must* be 5 V. If the embedded system's local supply will provide 5 V, then the connection to the programmer's **VCC** may be left unmade. However, if the embedded system's supply voltage is something other than 5 V, the programmer's **VCC** must be used, and any local power source within the embedded system should be disabled. **$\overline{\text{RESET}}$** plays an important role in downloading code. Programming begins with **$\overline{\text{RESET}}$** being asserted (driven low). This disables the CPU within the processor and thus allows access to the internal memory. It also changes the functionality of **PB0**, **PB1**, and **PB2** to a SPI interface. The development software then sends, via the SPI interface, a sequence of codes to "unlock" the program memory and enable software to be downloaded. Once programming is enabled, sequences of write commands are performed, and the software (and other settings) are downloaded byte by byte. The Atmel software takes care of this, so normally you don't need to worry about the specifics. If you need to do it "manually," perhaps from some other type of host computer, the Atmel datasheets give full details of the protocol.

The Atmel development system comes with a special adaptor cable that plugs into the company's development board and allows you to reprogram microprocessors via a PC's parallel port. By including the right connector (with the appropriate connections) in your circuit, it's possible to use the same programming cable on your own embedded system. Depending on the particular development board, there is one of two possible connectors for in-circuit programming. The pinouts for these are shown in Figure 15-8. (**VTG** is the voltage supply for the target system. If the target has its own power source, of the appropriate voltage level for programming (+5 V), then **VTG** may be left unconnected.) Pin 3 is labeled as a nonconnect on some Atmel application notes; however, some development systems use this to drive a LED indicating that a programming cycle is underway.

The schematic showing how to make your computer support this is shown in Figure 15-9. Note that **MOSI** on the connector goes to **MISO** on the processor, and, similarly, **MISO** goes to **MOSI** on the processor. This is because during programming, the processor is a *slave* and not a master.

INTEL EXHIBIT 1008 Page 126
IPR2023-00787

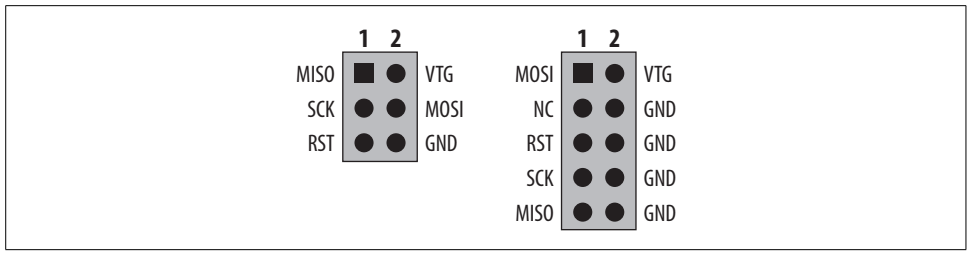


Figure 15-8. In-circuit programming connectors

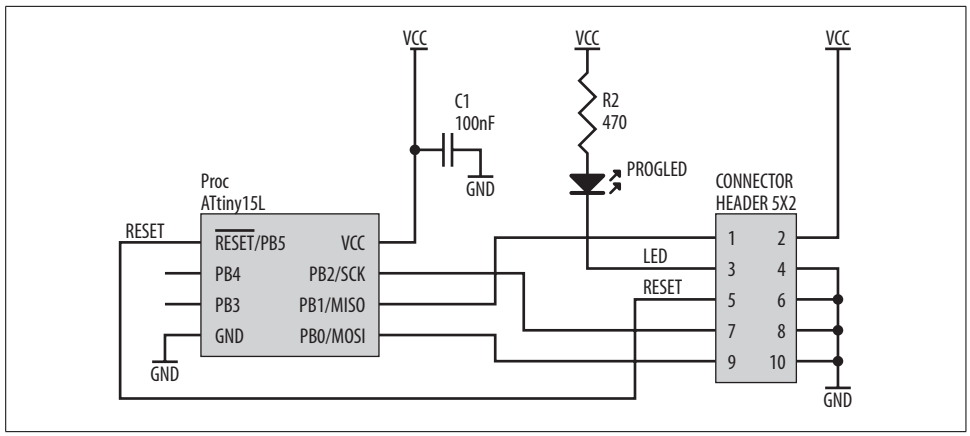


Figure 15-9. In-circuit programming

The connector type is an *IDC* header, and the cable provides all the signals necessary for programming, including one to drive a programming indicator LED. When not being used for programming, the connector may also double as a simple I/O connector for the embedded computer, allowing access to the digital signals. Thus, one piece of hardware can assume dual roles.

There is something important to note, however. If you use **PB0**, **PB1**, or **PB2** to interface to other components within your computer, care must be taken that the activity of programming does not adversely affect them. For example, our circuit with the MAX4626 used **PB2** as the control input. During programming, **PB2** acts as **SCK**, a clock signal. Therefore, the MAX4626 would be rapidly turning on and off as code was downloaded to the processor. If the MAX4626 was controlling something, that device would also rapidly turn on and off, with potentially disastrous effects. Conversely, if there are other components in your system, these must not attempt to drive a signal onto **PB0**, **PB1**, or **PB2** during the programming sequence. To do so would, at the very least, result in a failed download and, at worst, damage both the embedded system *and* the programmer. It's therefore vitally important to consider the implications of in-circuit programming on other components within the system.

INTEL EXHIBIT 1008 Page 127
IPR2023-00787

So, what's the answer? Well, we could use **PB3** to control the MAX4626 instead, since it doesn't take part in the programming process. Alternatively, if we needed to use **PB2**, we could provide a *buffer* between the processor and the MAX4626, perhaps controlled by **RESET**. When **RESET** is low (during programming), the buffer is disabled and the MAX4626 is isolated. Another solution may simply be to use a DIP version of the processor, mounted via a socket, and physically remove it for reprogramming. If you're using a surface-mount version of the processor, perhaps the processor could be mounted on a small PCB that plugs into the embedded computer (much like a memory SIMM on a desktop computer) and may be removed for programming. There are plenty of alternatives, and the best one really depends on your application.

Some AVRs (not the ATtiny15) have the capability of modifying their own program memory with the `SPM` (Store Program Memory) instruction. With such processors, it is possible for your software to download new code via the processor's serial port and write this into the program memory. To do so, you need to have your processor preprogrammed with a *bootloader* program. Normally, you would load all your processors with the bootloader (and Version 1.0 of your application software) during construction. The self-programming can then be used to update the application software when the systems are out in the field. To facilitate this, the program memory is divided into two separate sections: a *boot section* and an *application section*. The memory space is divided into *pages* of either 128 or 256 bytes (depending on the particular processor). Memory must be erased and reprogrammed one page at a time. During programming, the `Z` register is used as a pointer for the page address, and the `r1` and `r0` registers together hold the data word to be programmed. The Atmel application note (*AVR109: Self-programming*), available on its web site, gives sample source code for the bootloader and explains the process in detail.

No matter what processor you are using, the technical data from the chip manufacturer will tell you how to go about putting your code into the processor.

A Bigger AVR

So far, we have looked at a small AVR with very limited capabilities. In later chapters, we will look at various forms of input and output commonly found in embedded systems. For this, we will need processors with more functionality. We have exhausted the ATtiny15, so now we need to move on to processors with a bit more "grunt." Before getting into the details of I/O in the later chapters, you'll be introduced to these processors and learn what you need to do to include them in your design.

The first processor is the Atmel AT90S8535. This is a mid-range AVR with lots of inbuilt I/O, such as a UART, SPI, timers, eight channels of analog input, an analog comparator, and internal EEPROM for parameter storage. The processor has 512

INTEL EXHIBIT 1008 Page 128
IPR2023-00787

bytes of internal RAM and 8K of flash memory for program storage. Its smaller sibling, the AT90S4434, is identical in every other way except that it has smaller memory spaces of 4K for program storage and 256 bytes of RAM. But from a hardware point of view, the AT90S8535 and the AT90S4434 are the same.

The basic schematic for an AT90S8535-based computer, without any extras, is shown in Figure 15-10. It is not that different from the ATtiny15, save that it has a lot more pins. **RESET** has an external pull-up 10k resistor. The processor has an external crystal (X1), and this requires two small decoupling capacitors, C1 and C2. There are four power pins for this processor, and each is decoupled with a 100 nF ceramic capacitor. One of the power inputs (**AVCC**) is the power supply for the analog section of the chip, and this is isolated from the digital power supply by a 100 Ω resistor, R2. This is to provide a small barrier between the analog section and any switching noise that may be present from the digital circuits. The remaining pins are general-purpose digital I/O, as with the ATtiny15. However, unlike the ATtiny15, these pins have dual functionality. They may be configured, under software control, for alternative I/O functions. The processor's datasheet gives full details for configuring the functionality of the processor under software control.

This basic AVR design is applicable to most AVRs that you will find. The pinouts may be different, but the basic support required is the same. As with everything, grab the appropriate datasheet, and it will tell you the specifics for the particular processor that you are using.

AVR-Based Datalogger

In the previous chapter, we saw how to design a datalogger based on a PIC16F873A. A datalogger based on an AVR is not too dissimilar. Figure 15-11 shows the basic schematic.

The connections for interfacing a serial dataflash memory chip to an Atmel 90S4434 AVR processor are simply SPI, as with the PIC processor. The AVR portion of the schematic is no different from the examples we have seen previously. That's the nice thing about simple interfaces such as SPI. They form little subsystem modules that "bolt together" like building blocks. Start with the basic core design and just add peripherals as you need them. The schematic also shows decoupling capacitors for the power supplies, the crystal oscillator for the processor, and a pull-up resistor for **RESET**. Pin 41 (**PB1**) is used as a "manual" (processor-controlled) reset input to the flash.

The analog inputs, **ADC0:ADC7**, can be connected to an IDC header allowing for external sampling, or they can be interfaced directly to sensors, as we saw with the PIC datalogger. The serial port signals, RXD and TXD, connect to a MAX3233 in the same way as we saw in the PIC design.

INTEL EXHIBIT 1008 Page 129
IPR2023-00787

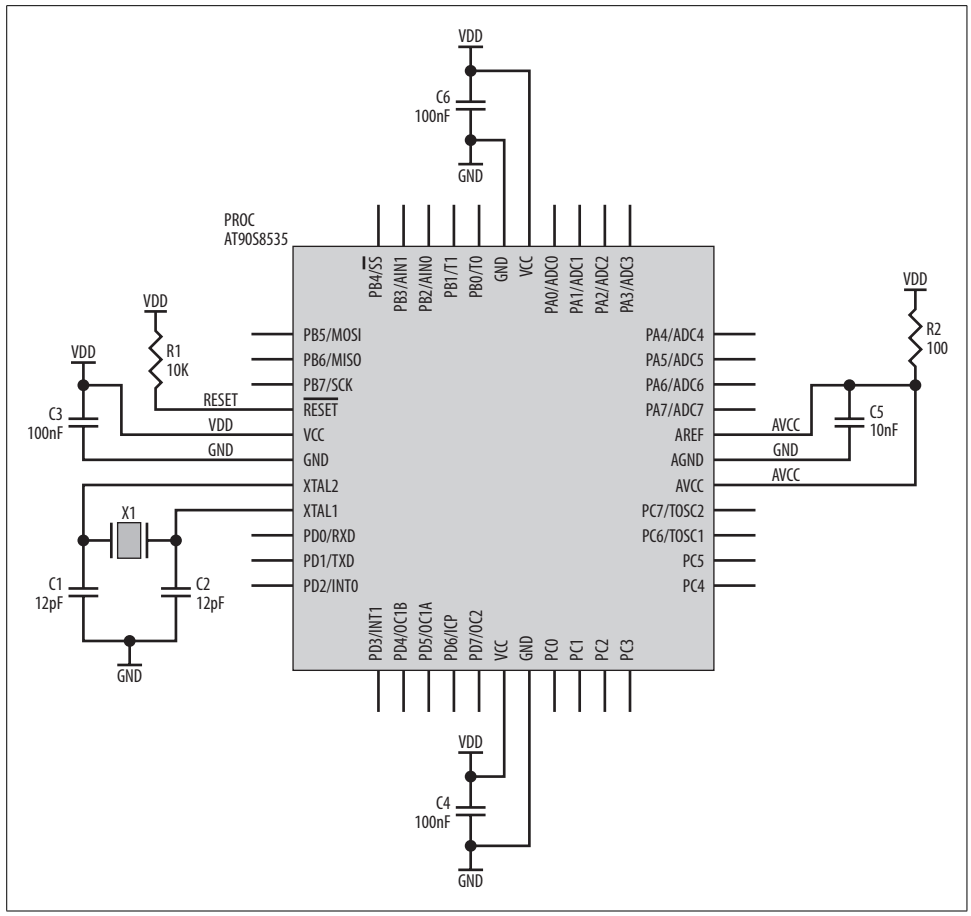


Figure 15-10. AT90S8535 processor and support components

Bus Interfacing

In this section, I'll show you how to expand the capabilities of your processor by interfacing it to bus-based memories and peripherals. Different processor architectures have different signals and different timing, but once you understand one, the basic principles can be applied to all. Since most small microcontrollers don't have external buses, the choice is very limited. We'll look at the one, and only, AVR with an external bus—the AT90S8515. In the PIC world, the PIC17C44 is capable of bus-based interfacing.

INTEL EXHIBIT 1008 Page 130
IPR2023-00787

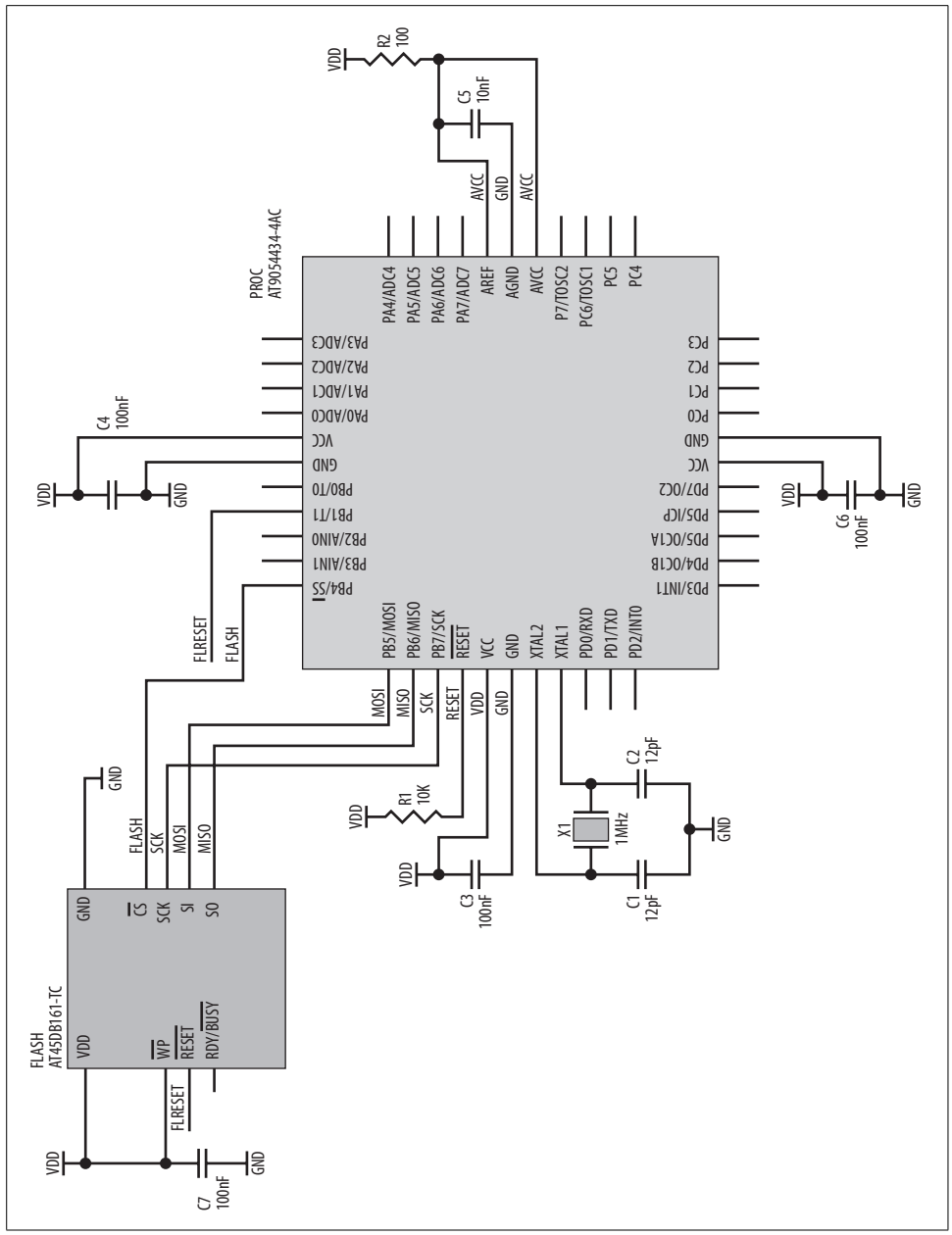


Figure 15-11. A 2M DataFlash interfaced to an AT90S4434

INTEL EXHIBIT 1008 Page 131
IPR2023-00787

AT90S8515 Memory Cycle

A *memory cycle* (also known as a *machine cycle* or *processor cycle*) is defined as the period of time it takes for a processor to initiate an access to memory (or peripheral), perform the transfer, and terminate the access. The memory cycle generated by a processor is usually of a fixed period of time (or multiples of a given period) and may take several (processor) clock cycles to complete.

Memory cycles usually fall into two categories: the *read cycle* and the *write cycle*. The memory or device that is being accessed requires that the data is held valid for a given period after it has been selected and after a read or write cycle has been identified. This places constraints on the system designer. There is a *limited* amount of time in which any *glue logic* (interface logic between the processor and other devices) must perform its function, such as selecting which external device is being accessed. The setup times *must* be met. If they are not, the computer will not function. The glue logic that monitors the address from the processor and uniquely selects a device is known as an *address decoder*. We'll take a closer look at address decoders shortly.

Timing is probably the most critical aspect of computer design. For example, if a given processor has a 150 ns cycle time and a memory device requires 120 ns from when it is selected until when it has completed the transfer, this leaves only 30 ns at the start of the cycle in which the glue logic can manipulate the processor signals. A 74LS series TTL gate has a typical propagation delay of 10 ns. So, in this example, an address decoder implemented using any more than two 74LS gates (in sequence) is cutting it very fine.

A *synchronous* processor has memory cycles of a fixed duration, and all processor timing is directly related to the clock. It is assumed that all devices in the system are capable of being accessed and responding within the set time of the memory cycle. If a device in the system is slower than that allowed by the memory cycle time, logic is required to pause the processor's access, thus giving the slow device time to respond. Each clock cycle within this pause is known as a *wait state*. Once sufficient time has elapsed (and the device is ready), the processor is released by the logic and continues with the memory cycle. Pausing the processor for slower devices is known as *inserting wait states*. The circuitry that causes a processor to hold is known as a *wait-state generator*. A wait-state generator is easily achieved using a series of flip-flops acting as a simple counter. The generator is enabled by a processor output indicating that a memory cycle is beginning and is normally reset at the end of the memory cycle to return it to a known state. (Some processors come with internal, programmable wait-state generators.)

An *asynchronous* processor (such as a 68000) does not terminate its memory cycle within a given number of clock cycles. Instead, it waits for a transfer acknowledge assertion from the device or support logic to indicate that the device being accessed has had sufficient time to complete its part in the memory cycle. In other words, the processor *automatically* inserts wait states in its memory cycle until the device being

INTEL EXHIBIT 1008 Page 132
IPR2023-00787

accessed is ready. If the processor does not receive an acknowledge, it will wait indefinitely. Many computer systems using asynchronous processors have additional logic to cause the processor to restart if it waits too long for a memory cycle to terminate. An asynchronous processor can be made into a synchronous processor by tying the acknowledge line to its active state. It then assumes that all devices are capable of keeping up with it. This is known as *running with no wait states*.

Most microcontrollers are synchronous, whereas most larger processors are asynchronous. The AT90S8515 is a synchronous processor, and it has an internal wait-state generator capable of inserting a single wait state.

Bus Signals

Figure 15-12 shows an AT90S8515 processor with minimal support components. The AT90S8515 has an address bus, a data bus, and a control bus that it brings to the outside world for interfacing. Since this processor has a limited number of pins, these buses share pins with the digital I/O ports ("port A" and "port C") of the processor. A bit in a control register determines whether these pins are I/O or bus pins. Now, a 16-bit address bus and an 8-bit data bus add up to 24 bits, but ports A and B have only 16 bits between them. So how does the processor fit 24 bits into 16? It multiplexes the lower half of the address bus with the data bus. At the start of a memory access, port A outputs address bits A0:A7. The processor provides a control line, **ALE** (Address Latch Enable), which is used to control a latch, such as a 74HCT573 (shown on the right in Figure 15-12). As **ALE** falls, the latch grabs and holds the lower address bits. At the same time, port B outputs the upper address bits, A8:A15. These are valid for the entire duration of the memory access. Once the latch has acquired the lower address bits, port A then becomes the data bus for information transfer between the processor and an external device. Also shown in Figure 15-12 are the crystal circuit, the In-System Programming port, decoupling capacitors for the processor's power supply, and net labels for other important signals.

The *timing diagrams* for an AT90S8515 are shown in Figure 15-13. The cycle "T3" exists only when the processor's wait-state generator is enabled.

Now, let's look at these signals in more detail. (We'll see later how you actually work with this information. For the moment, we're just going to "take a tour" of the timing diagrams.) The numbers for the timing information can be found in the datasheet, available from Atmel's web site. Figure 15-14 shows the timing information as presented in the Atmel datasheet, complete with timing references.

INTEL EXHIBIT 1008 Page 133
IPR2023-00787

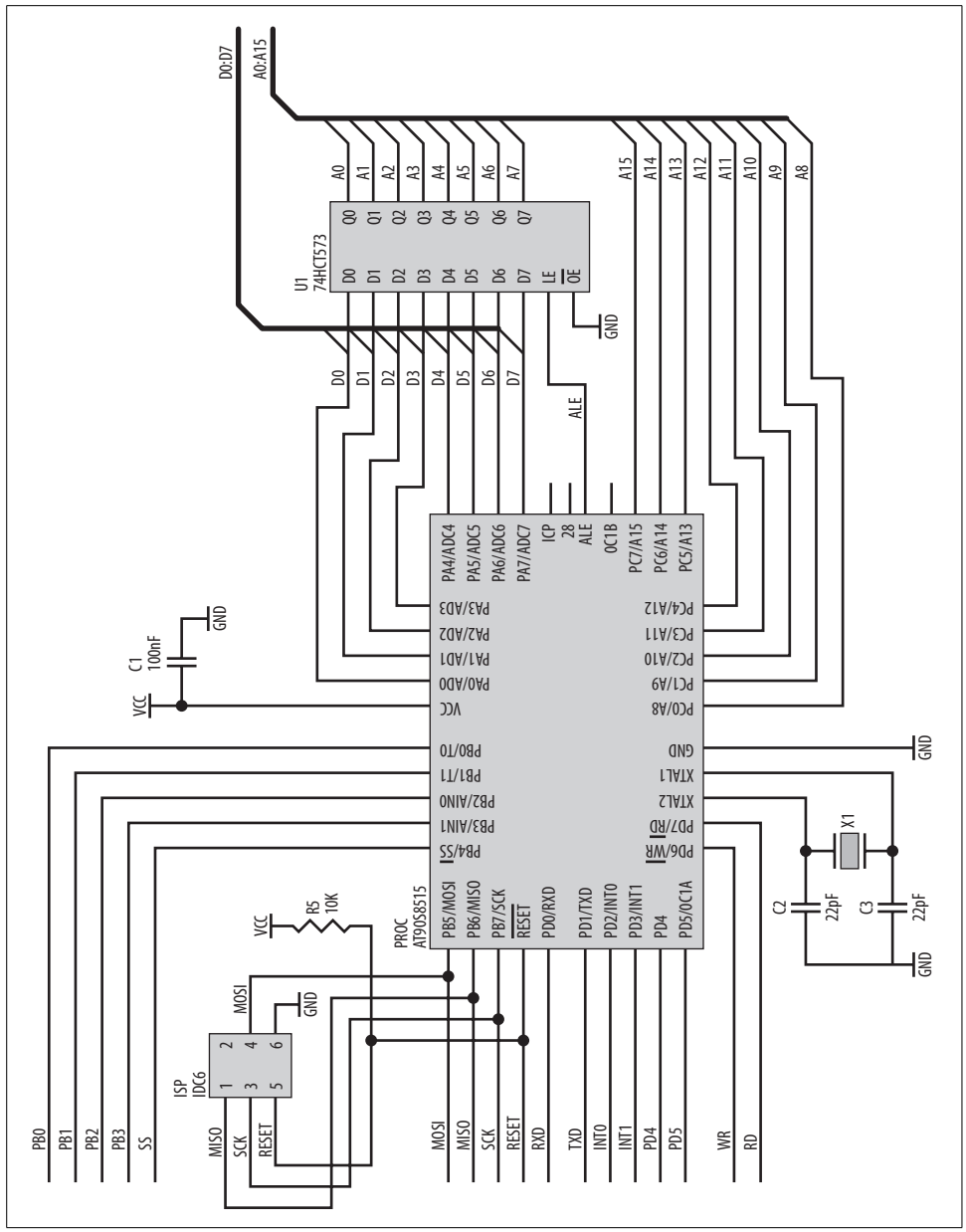


Figure 15-12. Address bus demultiplexing

INTEL EXHIBIT 1008 Page 134
IPR2023-00787

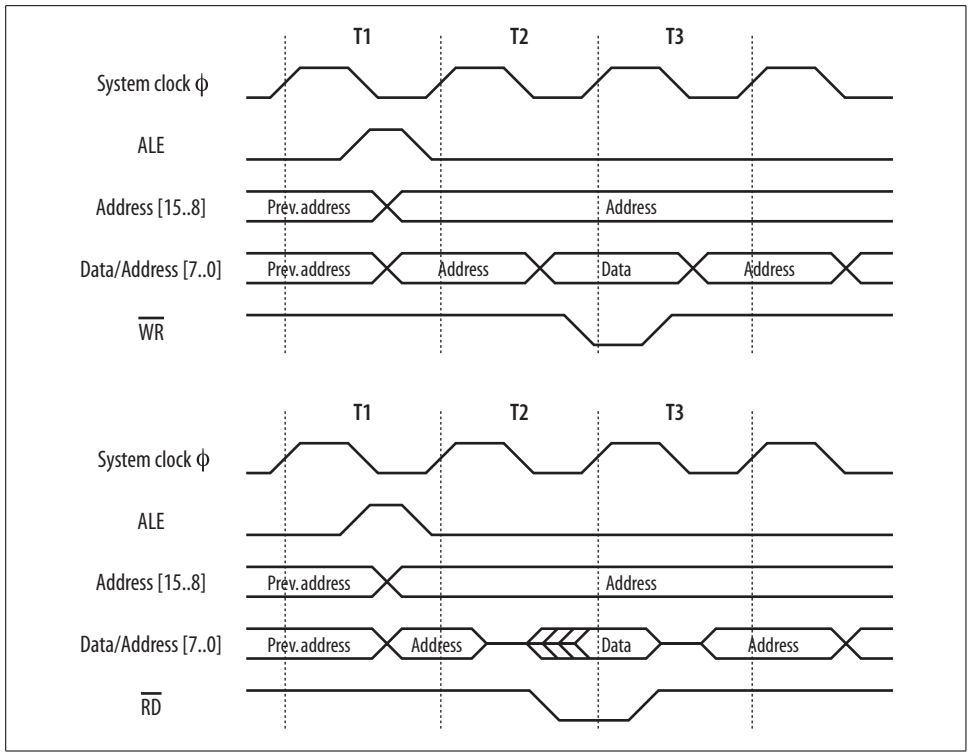


Figure 15-13. AT90S8515 memory cycles

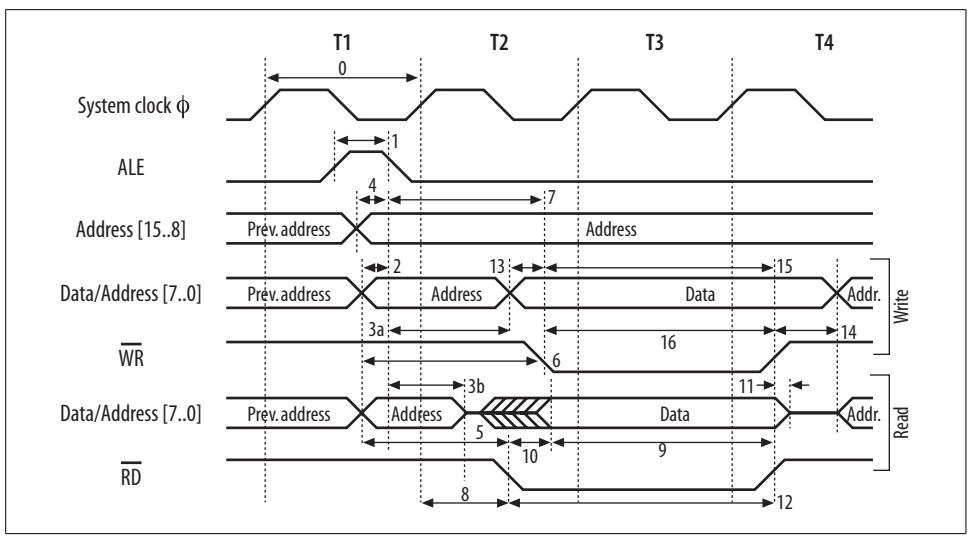


Figure 15-14. AT90S8515 memory cycles with timing parameters

INTEL EXHIBIT 1008 Page 135
IPR2023-00787

The references are looked up in the appropriate table in the processor's datasheet (Table 15-2).

Table 15-2. Timing parameters

Ref. #	Symbol	Parameter	8 MHz oscillator Min	8 MHz oscillator Max	Variable oscillator Min	Variable oscillator Max	Unit
0	$^1/t_{CLCL}$	Oscillator Frequency			0.0	8.0	MHz
1	t_{LHLL}	ALE Pulse Width	32.5		$0.5\,t_{CLCL}-30.0$		ns
2	t_{AVLL}	Address Valid A to ALE Low	22.5		$0.5\,t_{CLCL}-40.0$		ns
3a	$t_{LLAX...ST}$	Address Hold after ALE Low, ST/STD/STS Instructions	67.5		$0.5\,t_{CLCL}+5.0$		ns
3b	$t_{LLAX...LD}$	Address Hold after ALE Low, LD/LDD/LDS Instructions	15.0		15.0		ns
4	t_{AVLLC}	Address Valid C to ALE Low	22.5		$0.5\,t_{CLCL}-40.0$		ns
5	t_{AVRL}	Address Valid to RD Low	95.0		$1.0\,t_{CLCL}-30.0$		ns
6	t_{AVWL}	Address Valid to WR Low	157.5		$1.5\,t_{CLCL}-30.0$		ns
7	t_{LLWL}	ALE Low to WR Low	105.0	145.0	$1.0\,t_{CLCL}-20.0$	$1.0\,t_{CLCL}+20.0$	ns
8	t_{LLRL}	ALE Low to RD Low	42.5	82.5	$0.5\,t_{CLCL}-20.0$	$0.5\,t_{CLCL}+20.0$	ns
9	t_{DVRH}	Data Setup to RD High	60.0		60.0		ns
10	t_{RLDV}	Read Low to Data Valid		70.0		$1.0\,t_{CLCL}-55.0$	ns
11	t_{RHDX}	Data Hold after RD High	0.0		0.0		ns
12	t_{RLRH}	RD Pulse Width	105.0		$1.0\,t_{CLCL}-20.0$		ns
13	t_{DVWL}	Data Setup to WR Low	27.5		$0.5\,t_{CLCL}-35.0$		ns
14	t_{WHDX}	Data Hold after WR High	0.0		0.0		ns
15	t_{DVWH}	Data Valid to WR High	95.0		$1.0\,t_{CLCL}-30.0$		ns
16	t_{WLWH}	WR Pulse Width	42.5		$0.5\,t_{CLCL}-20.0$		ns

The system clock, ϕ, is shown at the top of both diagrams for reference, since all processor activity relates to this clock. The period of the clock is designated in the Atmel datasheet as "t_{CLCL}"* and is equal to 1/frequency. For an 8 MHz clock, this is 125 ns. The width of T1, T2, and T3 are each t_{CLCL}.

No processor cycle exists in isolation. There is always† a preceding cycle and following cycle. We can see this in the timing diagrams. At the start of the cycles, the

* Datasheet nomenclature can often be very cryptic. The "CL" comes from "clock." Since Atmel is using four-character subscripts for its timing references, it pads by putting "CL" twice. You don't really need to know what the subscripts actually mean; you just need to know the signals they refer to and the actual numbers involved.

† I'm ignoring coming out of reset, or just before power-off!

INTEL EXHIBIT 1008 Page 136
IPR2023-00787

address from the previous access is still present on the address bus. On the falling edge of the clock, in cycle T1, the address bus changes to become the valid address required for this cycle. Port A presents address bits A0:A7, and port B presents A8:A15. At the same time, **ALE** goes high, releasing the external address latch in preparation for acquiring the new address from port A. **ALE** stays high for $0.5 \times t_{CLCL} - 30$ ns. So, for example, with an AT90S8515 running at 8 MHz, **ALE** stays high for 32.5 ns. **ALE** falls, causing the external latch to acquire and hold the lower address bits. Prior to **ALE** falling, the address bits will have been valid for $0.5 \times t_{CLCL} - 40$ ns or, in other words, 40 ns before the system clock rises at the end of the T1 period. After ALE falls, the lower address bits will be held on port A for $0.5 \times t_{CLCL} + 5$ ns for a write cycle before changing to data bits. For a read cycle, they are held for a minimum of 15 ns only. The reason this is so much shorter for a read cycle is that the processor wishes to free those signal pins as soon as possible. Since this is a read cycle, an external device is about to respond, which means the processor needs to "get out of the way" as soon as it can.

For a write cycle, $t_{CLCL} - 20$ ns after **ALE** goes low, the write strobe, **$\overline{\text{WR}}$**, goes low. This indicates to external devices that the processor has output valid data on the data bus. **$\overline{\text{WR}}$** will be low for $0.5 \times t_{CLCL} - 20$ ns. This time allows the external device to prepare to read in (latch) the data. On the rising edge of **$\overline{\text{WR}}$**, the external device is expected to latch the data presented on the data bus. At this point, the cycle completes, and the next cycle is about to begin.

For a read cycle, the read strobe, **$\overline{\text{RD}}$**, goes low $0.5 \times tCLCL - 20$ ns after **ALE** is low. **$\overline{\text{RD}}$** will be low for $tCLCL - 20$ ns. During this time, the external device is expected to drive valid data onto the data bus. It can present data anytime after **$\overline{\text{RD}}$** goes low, so long as data is present and stable at least 60 ns before **$\overline{\text{RD}}$** goes high again. At this point, the processor latches the data from the external device, and the read cycle terminates. Note that many processors do not have a separate read-enable signal, so this must be generated by external logic, based on the premise that if the cycle is not a write cycle, it must be a read cycle.

So, that is how an AT90S8515 expects to access any external device attached to its buses, whether those devices are memory chips or peripherals. But how does it work in practice? Let's look at designing* a computer based on an AT90S8515 with some external devices. For this example, we will interface the processor to a static RAM and some simple latches that we could use to drive banks of LEDs.

* Since we've covered oscillators and in-circuit programming previously, I'll ignore those in this discussion. That doesn't mean you should leave them out of your design!

INTEL EXHIBIT 1008 Page 137
IPR2023-00787

Memory Maps and Address Decoding

To the processor, its address space is one big, linear region. Although there may be numerous devices within that space, both internal to the processor and external, it makes no distinction between devices. The processor simply performs memory accesses in the address space. It is up to the system designer (that's you) to allocate regions of memory to each device and then provide address-decode logic. The address decoder takes the address provided by the processor during an external access and uniquely selects the appropriate device (Figure 15-15). For example, if we have a RAM occupying a region of memory, any address from the processor corresponding to within that region should select the RAM and *not* select any other device. Similarly, any address outside that region should leave the RAM unselected.

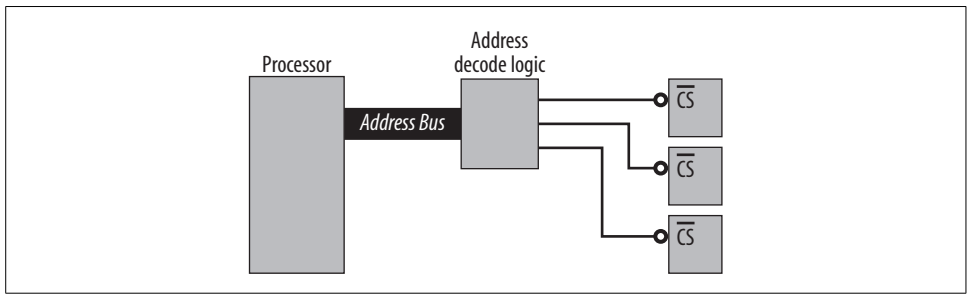


Figure 15-15. An address decoder uses the address to select one of several devices

The allocation of devices within an address space is known as a *memory map* or *address map*. The address spaces for an AT90S8515 processor are shown in Figure 15-16. Any device we interface to the processor must be within the data memory space. Thus, we can ignore the processor's internal program memory. As the processor has Harvard architecture, the program space is a completely separate address space. Within the 64K data space lie the processor's internal resources: the working registers, the I/O registers, and the internal 512 bytes of SRAM. These occupy the lowest addresses within the space. Any address above `0x0260` is ours to play with. (Not all processors have resources that are memory-mapped, and, in those cases, the entire memory space is usable by external devices.)

Now, our first task is to allocate the remaining space to the external devices. Since the RAM is 32K in size, it makes sense to place it within the upper half of the address space (`0x8000`–`0xFFFF`). Address decoding becomes much easier if devices are placed on neat boundaries. Placing the RAM between addresses `0x8000` and `0xFFFF` leaves the lower half of the address space to be allocated to the latches and the processor's internal resources. Now a latch need only occupy a single byte of memory within the address space. So, if we have three latches, we need only three bytes of the address space to be allocated. This is known as *explicit address decoding*. However, there's a good reason not to be so efficient with our address allocation. Decoding the address

INTEL EXHIBIT 1008 Page 138
IPR2023-00787

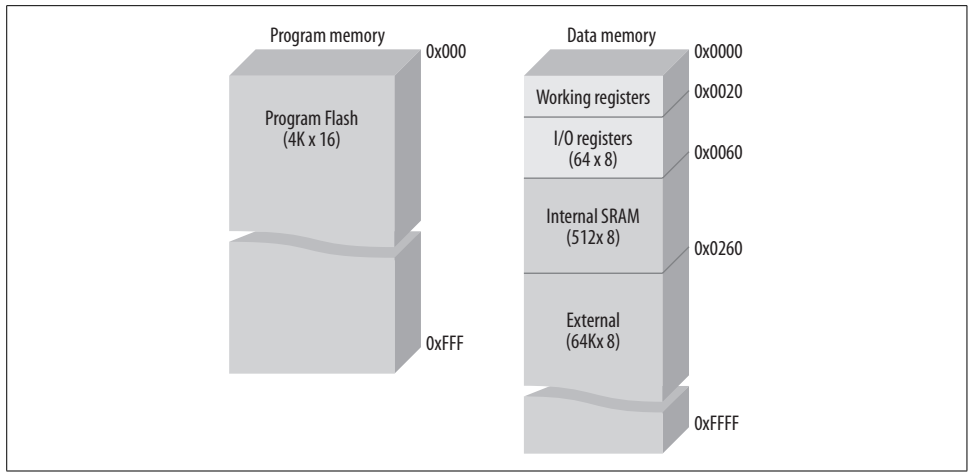


Figure 15-16. Atmel AT90S8515 memory map

down to three bytes would require an address decoder to use 14 bits of the address. That's a lot of (unnecessary) logic to select just three devices. A better way is simply to divide the remaining address space into four, allocating three regions for the latches and leaving the fourth unused (for the processor's internal resources). This is known as *partial address decoding* and is much more efficient. The trick is to use the minimal amount of address information to decode for your devices.

Our address map allocated to our static RAM and three latches is shown in Figure 15-17. Note that the lowest region leaves the addresses in the range `0x0260` to `0x1FFF` unused.

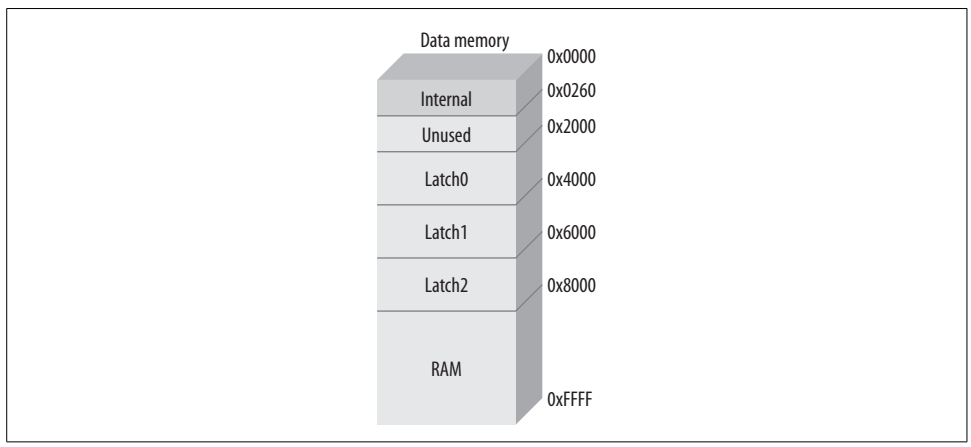


Figure 15-17. Allocated memory map

INTEL EXHIBIT 1008 Page 139
IPR2023-00787

Any address within the region 0x2000 to 0x3FFF will select Latch0, even though that latch needs only one byte of space. Thus, the device is said to be *mirrored* within that space. For simplicity in programming, you normally just choose an address (0x2000 say) and use that within your code. But you could just as easily use address 0x290F, and that would work too.

We now have our memory map, and we need to design an address decoder. We start by tabling the devices along with their addresses (Table 15-3). We need to look for which address bits are different between the devices, and which address bits are common within a given device's region.

Table 15-3. Address table

Device	Address range	A15 .. A0
Unused	0x0000–0x1FFF	0000 0000 0000 0000 0000 0001 1111 1111 1111 1111
Latch0	0x2000–0x3FFF	0010 0000 0000 0000 0000 0011 1111 1111 1111 1111
Latch1	0x4000–0x5FFF	0100 0000 0000 0000 0000 0101 1111 1111 1111 1111
Latch2	0x6000–0x7FFF	0110 0000 0000 0000 0000 0111 1111 1111 1111 1111
RAM	0x8000–0xFFFF	1000 0000 0000 0000 0000 1111 1111 1111 1111 1111

So, what constitutes a unique address combination for each device? Looking at the table, we can see that for the RAM, address bit (and address signal) **A15** is high, while for every other device it is low. We can therefore use **A15** as the trigger to select the RAM. For the latches, address bits **A15**, **A14**, and **A13** are critical. So we can redraw our table to make it clearer. This is the more common way of doing an address table, as shown in Table 15-4. An "x" means a "don't care" bit.

Table 15-4. Simplified address table

Device	Address range	A15 .. A0
Unused	0x0000–0x1FFF	000x xxxx xxxx xxxx xxxx
Latch0	0x2000–0x3FFF	001x xxxx xxxx xxxx xxxx
Latch1	0x4000–0x5FFF	010x xxxx xxxx xxxx xxxx
Latch2	0x6000–0x7FFF	011x xxxx xxxx xxxx xxxx
RAM	0x8000–0xFFFF	1xxx xxxx xxxx xxxx xxxx

Therefore, to decode the address for the RAM, we simply need to use **A15**. If **A15** is high, the RAM is selected. If **A15** is low, then one of the other devices is selected and the RAM is not. Now, the RAM has a chip select ($\overline{\mathbf{CS}}$) that is active low. So when

INTEL EXHIBIT 1008 Page 140
IPR2023-00787

A15 is high, **$\overline{\text{CS}}$** should go low. So, our address decoder for the RAM is simply to invert **A15** using an inverter chip such as a 74HCT04 (Figure 15-18). It is common practice to label the chip-select signal after the device it is selecting. Hence, our chip select to the RAM is labeled **$\overline{\text{RAM}}$**.

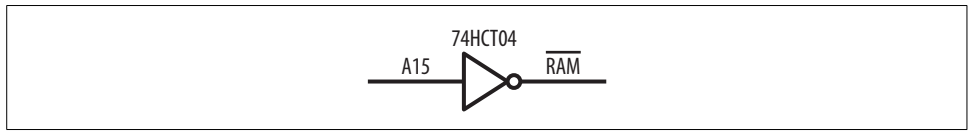


Figure 15-18. Address decode for the RAM

Note that for the RAM to respond, it needs both a chip select and either a read or write strobe from the processor. All other address lines from the processor are connected directly to the corresponding address inputs of the RAM (Figure 15-19).

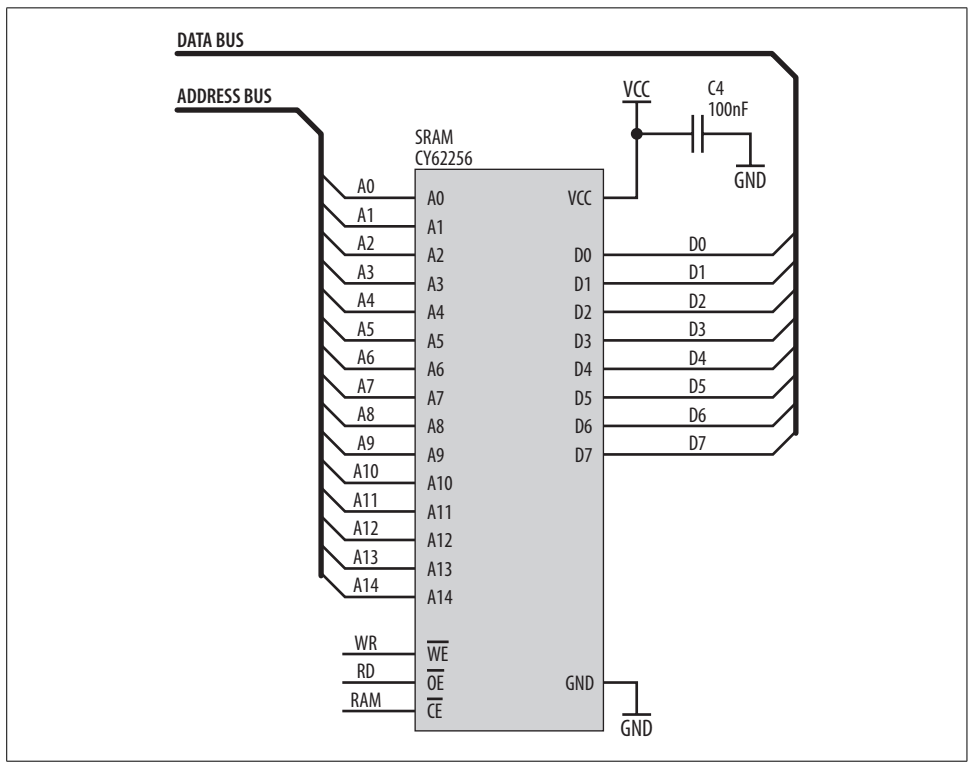


Figure 15-19. Connections to the SRAM

Now for the other four regions, **A15** must be low, and **A14** and **A13** are sufficient to distinguish between the devices. Having our address decoder use discrete logic would require several gates and would be "messy." There's a simpler way. We can

INTEL EXHIBIT 1008 Page 141
IPR2023-00787

use a 74HCT139* decoder, which takes two address inputs (A and B) and gives us four unique, active-low, chip-select outputs (labeled Y0:Y3). Since the latches require active-high enables, we use inverters on the outputs of the 7HCT139. So our complete address decoder for the computer is shown in Figure 15-20.

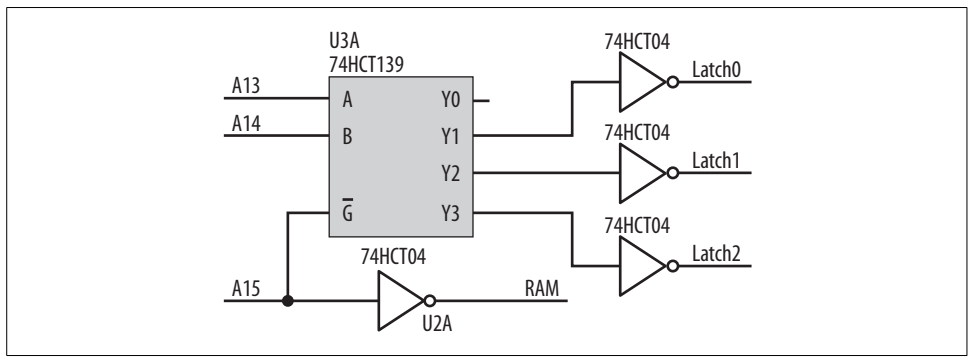


Figure 15-20. Complete address decoder

The 74HCT139 uses **A15** (low) as an enable (input $\overline{G}$), and, in this way, **A15** is included as part of the address decode. If we needed to decode for eight regions instead of four, we could have used a 74HCT138 decoder, which takes three address inputs and gives us eight chip selects. The interface between the processor and an output latch is simple. We can use the same type of latch (a 74HCT573) that we used to demultiplex the address. Such an output latch could be used in any situation in which we need some extra digital outputs. In the sample circuit shown in Figure 15-21, I'm using the latch to control a bank of eight LEDs.

The output from our 74HCT139 address decoder is used to drive the LE (Latch Enable) input of the 74HCT573. Whenever the processor accesses the region of memory space allocated to this device, the address decoder triggers the latch to acquire whatever is on the data bus. And so, the processor simply writes a byte to any address in this latch's address region, and that byte is acquired and output to the LEDs. (Writing a "0" to a given bit location will turn on a LED; writing a "1" will turn it off.)

Note that the latch's output enable ($\overline{\mathbf{OE}}$) is permanently tied to ground. This means that the latch is always displaying the byte that was last written to it. This is important, as we always want the LEDs to display, and not just transitorily blink on, while the processor is accessing them.

Using the 74HCT139 in preference to discrete logic gates makes our design much simpler, but there's an even better way to implement system glue.

* There are actually two separate decoders in each 74HCT139 chip. We'll need only one.

INTEL EXHIBIT 1008 Page 142
IPR2023-00787

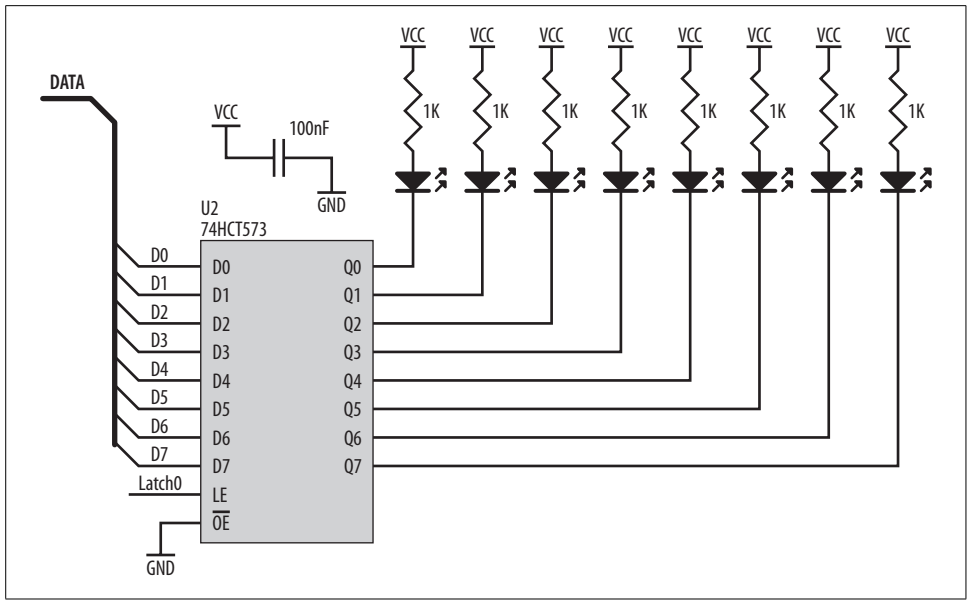


Figure 15-21. Using a 74HCT573 latch to control a bank of LEDs

PALs

It is now rare to see support logic implemented using individual gates. It is more common to use programmable logic (PALs, LCAs, or PLDs)* to implement the miscellaneous "glue" functions that a computer system requires. Such devices are fast, take up relatively little space, have low power consumption, and, as they are *reprogrammable*, make system design much easier and more versatile.

There is a wide range of devices available, from simple chips that can be used to implement glue logic (just as we are about to do) to massive devices with hundreds of thousands of gates. Altera (*http://www.altera.com*), Xilinx (*http://www.xilinx.com*), Lattice Semiconductor (*http://www.latticesemi.com*), and Atmel are some manufacturers to investigate for large-scale programmable logic. These big chips are sophisticated enough to contain entire computer systems. *Soft cores* are processor designs implemented in gates, suitable for incorporating into these logic devices. You can also get serial interfaces, disk controllers, network interfaces, and a range of other peripherals, all for integration into one of these massive devices. Of course, it's also fun to experiment and design your own processor from the ground up.

Each chip family requires its own suite of development tools. These allow you to create your design (either using schematics or some programming language such as

* Programmable Array Logic, Logic Cell Arrays, and Programmable Logic Devices, respectively.

INTEL EXHIBIT 1008 Page 143
IPR2023-00787

VHDL), simulate the system, and finally download your creation into the chip. You can even get C compilers for these chips that will take an algorithm and convert it, not into machine code, but into *gates*. What was software now runs not *on* hardware, but *as* hardware. Sounds cool, but the tools required to play with this stuff can be expensive. If you just want to throw together a small, embedded system, they are probably out of your price range. For what we need to do for our glue logic, such chips are overkill. Since our required logic is simple, we will use a simple (and cheap) PAL that can be programmed using freely available, public-domain software.

PALs are configured using equations to represent the internal logic. "+" represents OR, "*" represents AND, and "/" represents NOT. (These symbols are the original operator symbols that were used in Boolean logic. If you come from a programming background, these symbols may seem strange to you. You will be used to seeing "|", "&", and "!".) The equations are compiled using software such as PALASM, ABEL, or CUPL to produce a *JED* file. This is used by a device known as a *PAL burner* to configure the PAL. In many cases, standard EPROM burners will also program PALs.

PALs have pins for input, pins for output, and pins that can be configured as either input or output. Most of the PAL's pins are available for your use. In your PAL source code file (PDS file), you declare which pins you are using and label them. This is not unlike declaring variables in program source code, except that instead of allocating bytes of RAM, you're allocating physical pins of a chip. You then use those pin labels within equations to specify the internal logic. Our address decoder, implemented in a PAL, would have the following equations to specify the decode logic:

```
RAM = /A15
LATCH0 = (/A15 * /A14 * A13)
LATCH1 = (/A15 * A14 * /A13)
LATCH2 = (/A15 * A14 * A13)
```

I have (deliberately) written the above equations in a form that makes it easier to compare them with the address tables listed previously. You could simplify these equations, but there is no need. Just as an optimizing C compiler will simplify (and speed up) your program code, so too will PALASM rework your equations to optimize them for a PAL.

A PDS file to program a 22V10 PAL for the above address decode might look something like:

```
TITLE decoder.pds          ; name of this file
PATTERN
REVISION 1.0
AUTHOR John Catsoulis
DATE January 2005

CHIP decoder PAL22V10      ; specify which PAL device you
                           ; are using and give it a name ("decoder")

PIN   2   A15              ; pin declarations and allocations
```


INTEL EXHIBIT 1008 Page 144
IPR2023-00787

```
PIN   3   A14
PIN   12  LATCH0
PIN   13  LATCH1
PIN   14  LATCH2
PIN   15  RAM

EQUATIONS                  ; equations start here

RAM = /A15
LATCH0 = (/A15 * /A14 * A13)
LATCH1 = (/A15 * A14 * /A13)
LATCH2 = (/A15 * A14 * A13)
```

The advantages of using a PAL for system logic are twofold. The PAL equations may be changed to correct for bugs or design changes. The propagation delays through the PAL are of a fixed and small duration (no matter what the equations), which makes analyzing the overall system's timing far simpler. For very simple designs, it probably doesn't make a lot of difference whether you use PALs or individual chips. However, for more complicated designs, programmable logic is the only option. If you can use programmable logic devices instead of discrete logic chips, please do so. They make life much easier.

Timing Analysis

Now that we have finished our logic design, the question is: will it actually work? It's time (pardon the pun) to work through the numbers and analyze the timing. This is the least fun, and most important, part of designing a computer.

We start with the signals (and timing) of the processor, add in the effects of our glue logic, and finally see if this falls within the requirements of the device to which we are interfacing. We'll work through the example for the SRAM. For the other devices, the analysis follows the same method. The timing diagram for a read cycle for the SRAM is shown in Figure 15-22. The RAM I have chosen is a CY62256-70 (32K) SRAM made by Cypress Semiconductor. Most 32K SRAMs follow the JEDEC standard, which means their pinouts and signals are all compatible. So, what works for one 32K SRAM *should* work for them all. But the emphasis is on *should*, and, as always, *check the datasheet* for the individual device you are using.

The "-70" in the part number means that this is a "70 ns SRAM," or, put simply, the access time for the chip is 70 ns. Now, from the CY62256-70 datasheet (available from *http://www.cypress.com*), t_{RC} is a minimum of 70 ns. This means that the chip enable, $\overline{\mathbf{CE}}$, can be low for no less than 70 ns. $\overline{\mathbf{CE}}$ is just our chip select ($\overline{\mathbf{RAM}}$) from our address decoder, so we need to ensure that the address decoder will hold $\overline{\mathbf{RAM}}$ low for at least this amount of time. For the SRAM to output data during a read cycle, it needs a valid address, an active chip enable, and an active output enable ($\overline{\mathbf{OE}}$). The output enable is just the read strobe ($\overline{\mathbf{RD}}$) from the processor. These three conditions must be met before the chip will respond with data. It will take 70 ns

INTEL EXHIBIT 1008 Page 145
IPR2023-00787

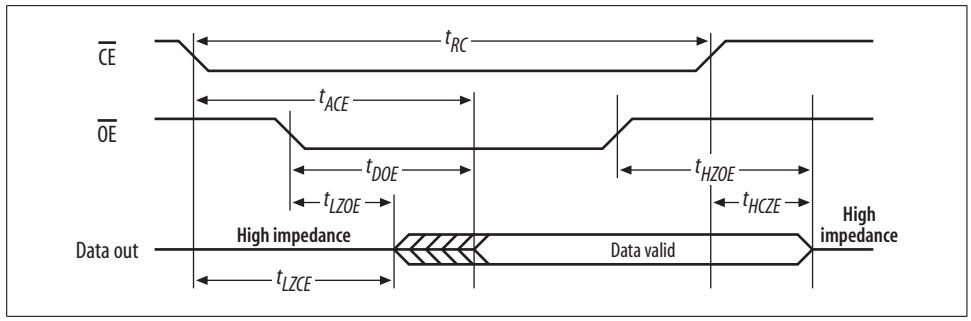


Figure 15-22. Timing for a read cycle to the RAM

from **CE** low (t_{ACE}) *or* 35 ns from **OE** low (t_{DOE}), whichever is the latter, until data is output. Now, **CE** is generated by our address decoder (which in turn uses address information from the processor), and **OE** (**RD**) comes from the processor. During a read cycle, the processor will output a read strobe and an address, which in turn will trigger the address decoder. Some time later in the cycle, the processor will expect data from the RAM to be present on the data bus. It is critical that the signals that cause the RAM to output data will do so such that there will be valid data when the processor expects it. Meet this requirement, and you have a processor that can read from external memory. Fail this requirement, and you'll have an intriguing paperweight and a talking piece at parties.

We start with the processor. I'm assuming that the processor's wait-state generator is disabled. For an AT90S8515 processor, everything is referenced to the falling edge of **ALE**. The high-order address bits, which feed our address decoder, become valid 22.5 ns prior to **ALE** going low on an 8 MHz AT90S8515. If we're using a 74HCT139 as an address decoder,* this takes 40 ns to respond to a change in inputs. So, our chip select for the RAM will become valid 17.5 ns after **ALE** has fallen (Figure 15-23).

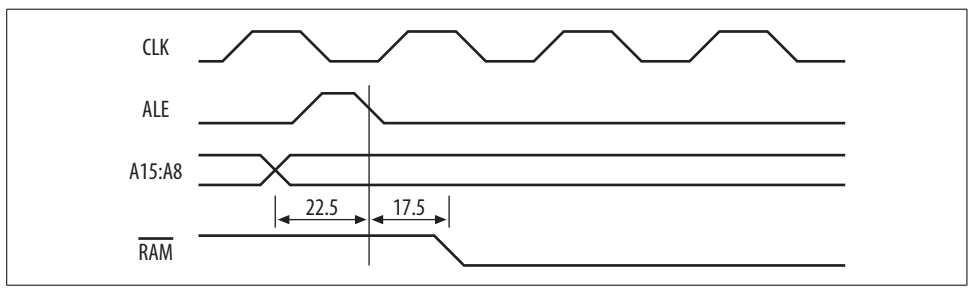


Figure 15-23. Timing for RAM chip select

* PALs may respond in 15 ns or less. This is another reason why PALs are a better choice than discrete logic.

INTEL EXHIBIT 1008 Page 146
IPR2023-00787

Now, **RD** will go low between 42.5 ns and 82.5 ns after **ALE** falls. Since the RAM will not output data until **RD** (**OE**) is low, we take the worst case of 82.5 ns (Figure 15-24).

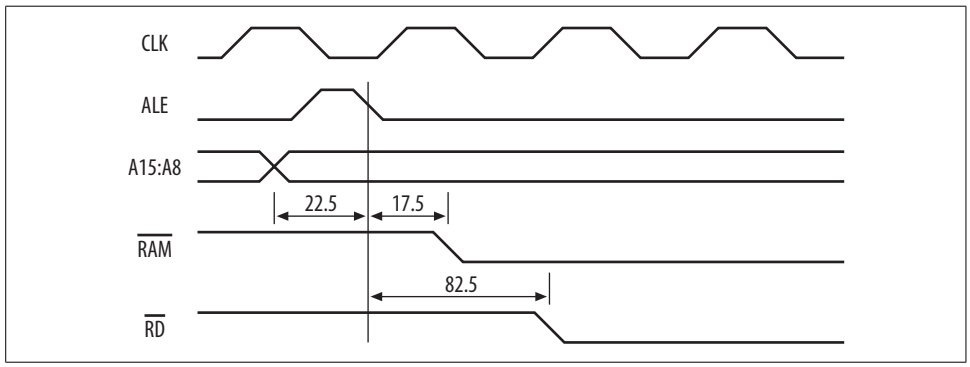


Figure 15-24. Read strobe and chip select for RAM

The RAM will respond 70 ns after **RAM** and 35 ns after **RD**, whichever comes last. So, 70 ns from **RAM** low is 87.5 ns after **ALE**, and 35 ns after **RD** is 117.5 ns after **ALE**. Therefore, **RD** is the determining control signal in this case. This means that the SRAM will output valid data 117.5 ns after **ALE** falls (Figure 15-25).

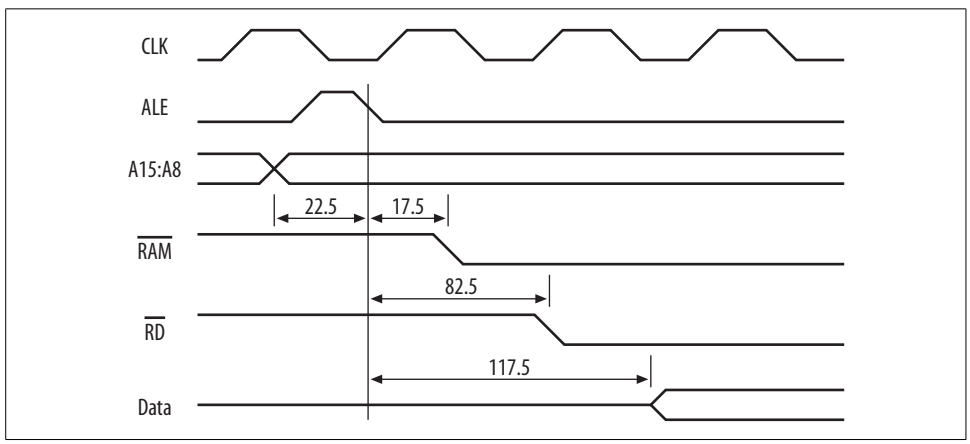


Figure 15-25. Valid data from the SRAM

Now, an 8 MHz processor expects to latch valid data during a read cycle at 147.5 ns after **ALE**. So our SRAM will have valid data ready with 30 ns to spare. So far, so good. But what about at the end of the cycle? Now, the processor expects the data bus to be released and available for the next access at 200 ns after **ALE** falls. The RAM takes 25 ns from when it is released by **CS** until it stops outputting data onto the data bus. This means that the data bus will be released by the RAM at 142.5 ns. So that will work too.

INTEL EXHIBIT 1008 Page 147
IPR2023-00787

The analysis for a write cycle is done in a similar manner. It is important to do this type of analysis for every device interfaced to your processor, for every type of memory cycle. It can be difficult, because datasheets are notorious for leaving out information or presenting necessary data in a roundabout way. Working through it all can be time-consuming and frustrating, and it's far too easy to make a mistake. However, it is very necessary. Without it, you're relying on blind luck to make your computers run, and that's not good engineering.

Memory Management

In most small-scale embedded applications, the connections between a processor and an external memory chip are straightforward. Sometimes, though, it's advantageous to play with the natural order of things. This is the realm of memory management.

Memory management deals with the translation of logical addresses to physical addresses and vice versa. A *logical address* is the address output by the processor. A *physical address* is the actual address being accessed in memory. In small computer systems, these are often the same. In other words, no address translation takes place, as illustrated in Figure 15-26.

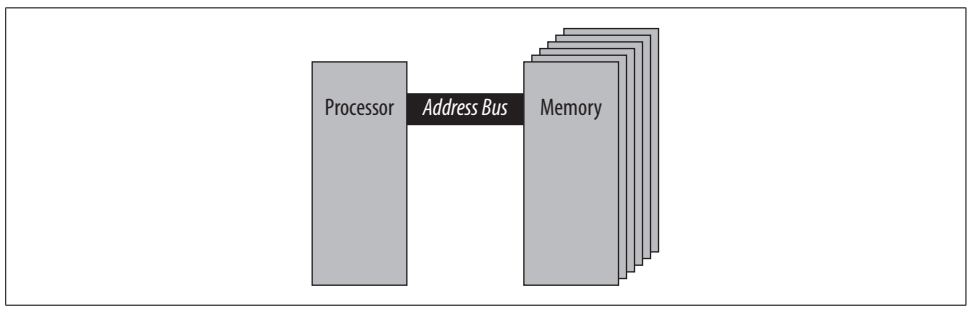


Figure 15-26. No address translation

For small computer systems, this absence of memory management is satisfactory. However, in systems that are more complex, some form of memory management may become necessary. There are four cases where this might be so:

Physical memory > logical memory

When the logical address space of the processor (determined by the number of address lines) is smaller than the actual physical memory attached to the system, it becomes necessary to *map* the logical space of the processor into the physical memory space of the system. This is sometimes known as *banked memory*. This is not as strange or uncommon as it may sound. Often, it is necessary to choose a particular processor for a given attribute, yet that processor may have a limited address space—too small for the application. By implementing

INTEL EXHIBIT 1008 Page 148
IPR2023-00787

banked memory, the address space of the processor is expanded beyond the limitation of the logical address range.

Logical memory > physical memory

When the logical address space of the processor is very large, it is not always practical to fill this address space with physical memory. It is possible to use some space on disk as *virtual memory*, thus making it appear that the processor has more physical memory than exists within the chips. Memory management is used to identify whether a memory access is to physical memory or virtual memory and must be capable of swapping the virtual memory on disk with real memory and performing the appropriate address translation.

Memory protection

It is often desirable to prevent some programs from accessing certain sections of memory. Protection can prevent a crashing program from corrupting the operating system and bringing down the computer. It is also a way of channeling all I/O access via the operating system, since protection can be used to prevent all software (save the OS) from accessing the I/O space.

Task isolation

In a multitasking system, tasks should not be able to corrupt each other (by stomping on each other's memory space, for example). In addition, two separate tasks should be able to use the same logical address in memory, with memory management performing the translation to *separate*, physical addresses.

The basic idea behind memory management is quite simple, but the implementation can be complicated, and there are nearly as many memory-management techniques as there are computer systems that employ memory management. Memory management is performed by a *Memory Management Unit* (*MMU*). The basic form of this is shown in Figure 15-27. An MMU may be a commercial chip, a custom-designed chip (or logic), or an integrated module within the processor. Most modern, fast processors incorporate MMUs on the same chip as the CPU.

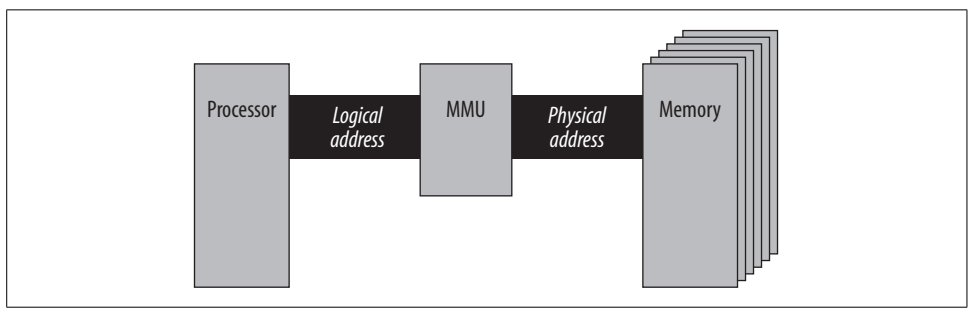


Figure 15-27. Address translation using an MMU

INTEL EXHIBIT 1008 Page 149
IPR2023-00787

Page mapping

In all practical memory-management systems, words of memory are grouped together to form *pages*, and an address can be considered to consist of a page number and the number of a word within that page. The MMU translates the logical page to a physical page, while the word number is left unchanged (Figure 15-28). In practice, the overall address is just a concatenation of the page number and the word number.

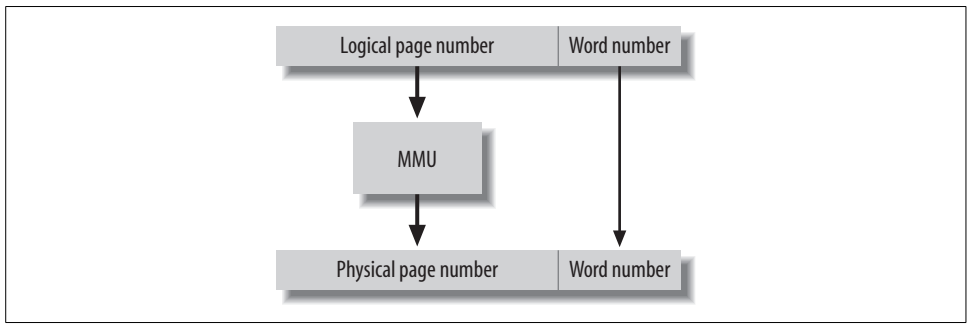


Figure 15-28. Address translation

The logical address from the processor is divided into a page number and a word number. The page number is translated by the MMU and recombined with the word number to form the physical address presented to memory (Figure 15-29).

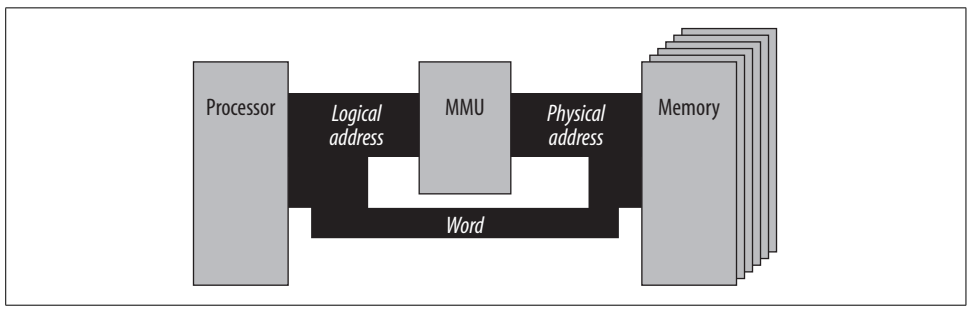


Figure 15-29. System using page address translation

Banked memory

The simplest form of memory management is when the logical address space is smaller than the physical address space. If the system is designed such that the size of a page is equal to the logical address space, then the MMU provides the page number, thus mapping the logical address into the physical address (Figure 15-30).

INTEL EXHIBIT 1008 Page 150
IPR2023-00787

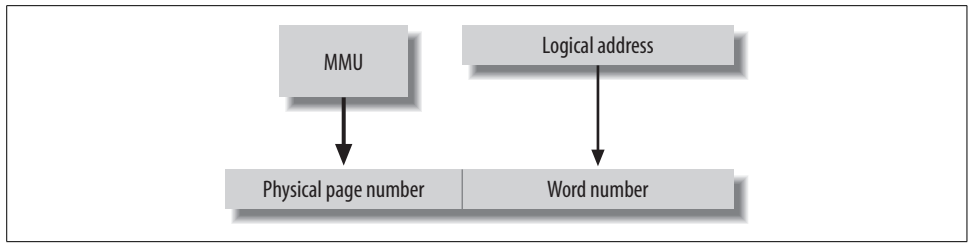


Figure 15-30. MMU generation of page number

The effective address space from this implementation is shown in Figure 15-31. The logical address space can be mapped (and *remapped*) to anywhere in the physical address space.

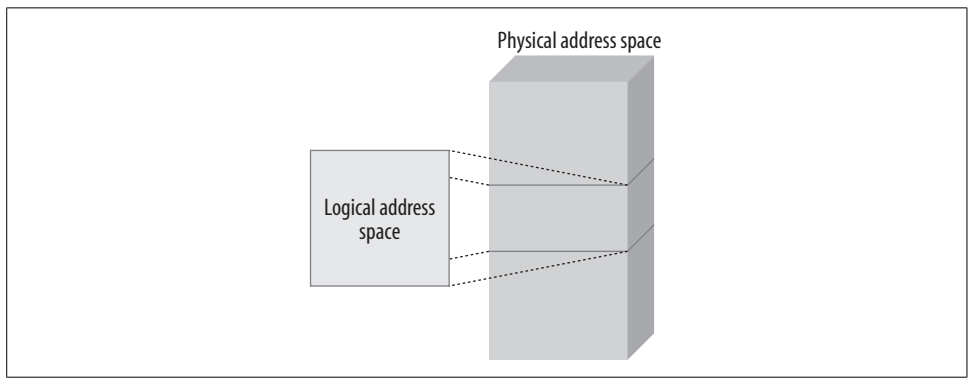


Figure 15-31. Mapping a smaller logical address space into a larger physical address

The system configuration for this is shown in Figure 15-32. This technique is often used in processors with 16-bit addresses (64K logical space) to give them access to larger memory spaces.

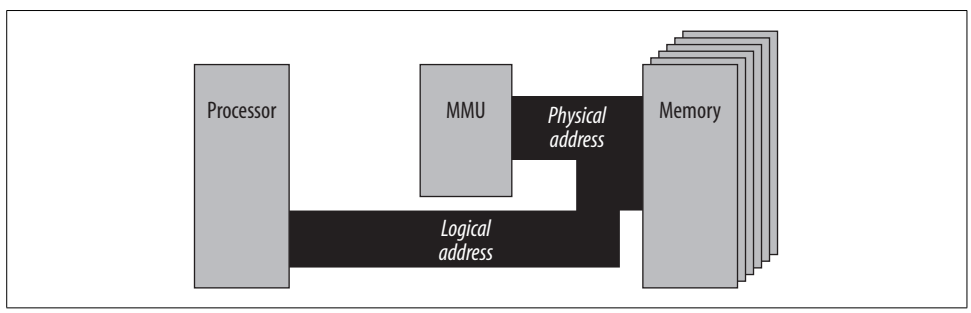


Figure 15-32. Generating a larger physical address

For many small systems, banked memory may be implemented simply by latching (acquiring and holding) the data bus and using this as the additional address bits for

INTEL EXHIBIT 1008 Page 151
IPR2023-00787

the physical memory (Figure 15-33). The latch appears in the processor's logical space as just another I/O device. To select the appropriate bank of memory, the processor stores the bank bits to the latch, where they are held. All subsequent memory accesses in the logical address space take place within the selected bank. In this example, the processor's address space acts as a 64K window into the larger RAM chip. As you can see, while memory management may seem complex, its actual implementation can be quite simple.

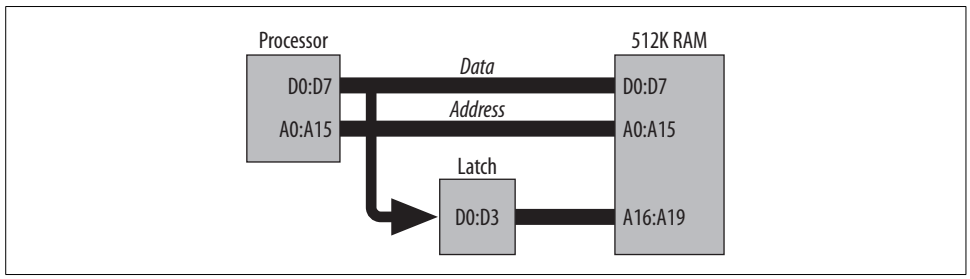


Figure 15-33. Simple banked-memory implementation



This technique has also been used in desktop systems. The old Apple /// computer came with up to 256K of memory, yet the address space of its 6502 processor was only 64K.

Figure 15-34 shows the actual wiring required for a banked-memory implementation for our AT90S8515 AVR system, replacing the 32K RAM with a 512K RAM.

The RAM used is an HM628511H made by Hitachi. In this implementation, we still have the RAM allocated into the upper 32K of the processor's address space as before. In other words, the upper 32K of the processor's address space is a window into the 512K RAM. The lower 32K of the processor's address space is used for I/O devices, as before. Address bits **A0** to **A14** connect to the RAM as before, and the data bus (**D0** to **D7**) connects to the data pins (**IO1** to **IO8***) of the SRAM.

Now, we also have a 74HCT573 latch, which is mapped into the processor's address space, just as we did with the LEDs latch. The processor can write to this latch, and it will hold the written data on its outputs. The lower nybble of this latch is used to provide the high-order address bits for the RAM.

Let's say the processor wants to access address `0x1C000`. In binary, this is `%001 1100 0000 0000 0000`. The lower 15 address bits (**A0** to **A14**) are provided directly by the processor. The remaining address bits must be latched. So, the processor first stores the byte `0x03` to the latch, and the RAM's address pins **A18**, **A17**, **A16**, and **A15** see

* Memory chip manufacturers often label data pins as "IO" pins, since they perform data input and output for the device.

INTEL EXHIBIT 1008 Page 152
IPR2023-00787

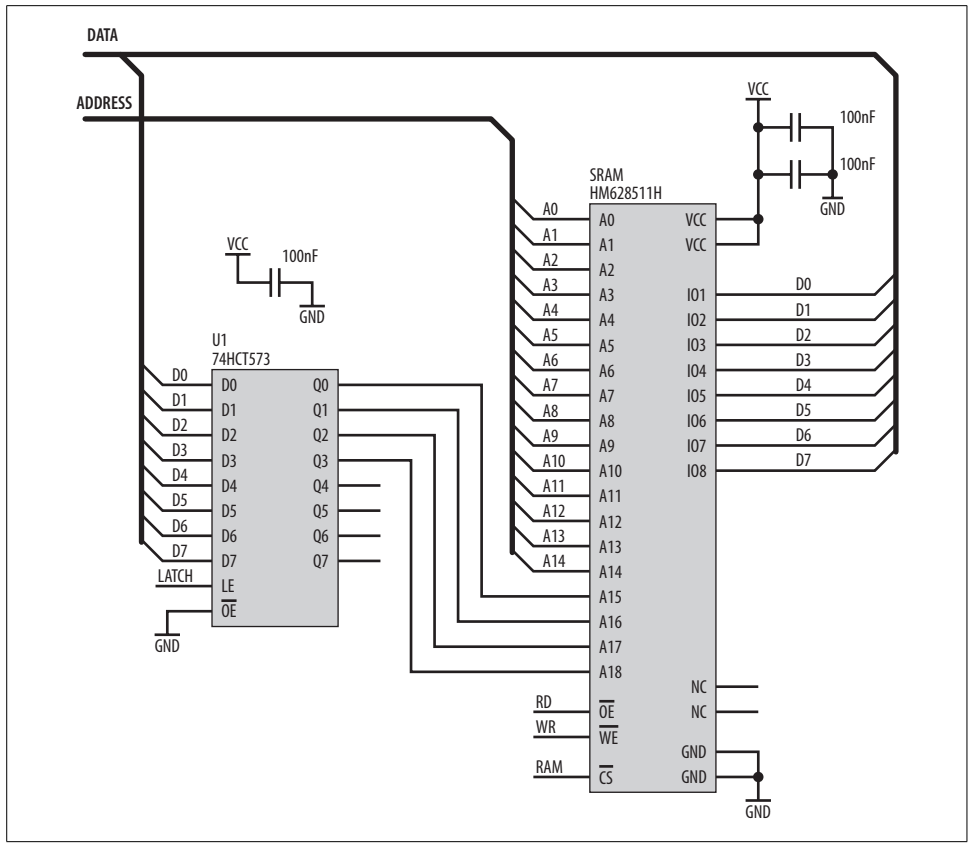


Figure 15-34. Banked memory for an AVR computer

`%0011` (`0x03`), respectively. That region of the RAM is now banked to the processor's 32K window. When the processor accesses address `0xC000`, the high-order address bit (**A15**) from the processor is used by the address decoder to select the RAM by sending its $\overline{\textbf{CS}}$ input low. The remaining 15 address bits (**A0** to **A14**) combine with the outputs of the latch to select address `0x1C000`.

The **NC** pins are "No Connection" and are left unwired.

Address translation

For processors with larger address spaces, the MMU can provide translation of the upper part of the address bus (Figure 15-35).

The MMU contains a *translation table*, which remaps the input addresses to different output addresses. To change the translation table, the processor must be able to access the MMU. (There is little point in having an MMU if the translation table is unalterable.) Some processors are specifically designed to work with an MMU, while

INTEL EXHIBIT 1008 Page 153
IPR2023-00787

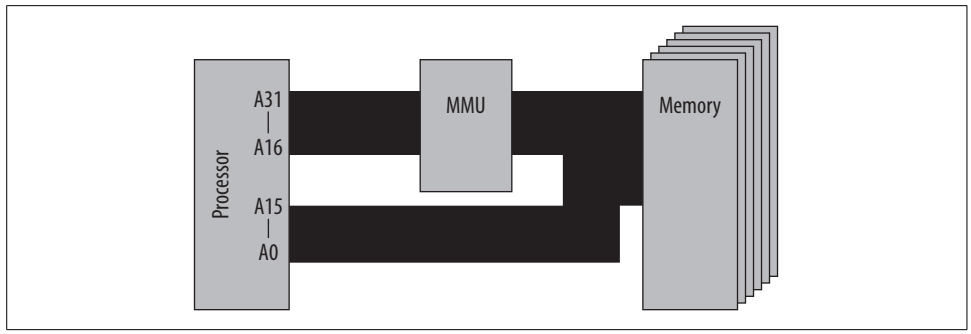


Figure 15-35. Logical page-number translation

other processors have MMUs incorporated. However, if the processor being used was not designed for use with an MMU, it will have no special support. The processor must therefore communicate with the MMU as though it were any other peripheral device using standard read/write cycles. This means the MMU must appear in the processor's address. It may seem that the simplest solution is to map the MMU into the physical address space of the system. In real terms, this is not practical. If the MMU is ever (intentionally or accidentally) mapped out of the current logical address space (such that the physical page on which the MMU is located is not part of the current logical address space), it becomes impossible to access the MMU ever again. This may also happen when the system powers up, because the contents of the MMU's translation table may be unknown.

The solution is to decode the chip select for the MMU directly from the logical address bus of the processor. Hence, the MMU will lie at a constant address in the logical space. This removes the possibility of "losing" the MMU, but it introduces another problem. Since the MMU now lies directly in the logical address space, it is no longer protected from accidental tampering (by a crashing program) or illegal and deliberate tampering in a multitasking system. To solve this problem, many larger processors have two states of operation—*supervisor state* and *user state*—with separate stack pointers for each mode. This provides a barrier between the operating system (and its privileges) and the other tasks running on the system. The state the processor is in is made available to the MMU through special status pins on the processor. The MMU may be modified only when the processor is in supervisor state, thereby preventing modification by user programs. The MMU uses a different logical-to-physical translation table for each state. The supervisor translation table is usually configured on system initialization, then remains unchanged. User tasks (user programs) normally run in user state, whereas the operating system (which performs task swapping and handles I/O) runs in supervisor state. Interrupts also place the processor in supervisor state, so that the vector table and service routines do not have to be part of the user's logical address space. While in user state, tasks may be denied access to particular pages of physical memory by the operating system.

INTEL EXHIBIT 1008 Page 154
IPR2023-00787

CHAPTER 18
68000-Series Computers

All is flux, nothing stays still.
—Heraclitus
Diogenes Laertius' *Lives of Eminent Philosophers*

In this chapter, we'll take a look at a 32-bit processor that has been around for quite some time and has evolved into a plethora of controllers and embedded processors. The 68000 (also known as the "68k") is produced by Freescale Semiconductor and is licensed by several other manufacturers. The range of 68000-based processors is large (check out the manufacturers' web sites for a list of processors and their features). The number of applications that the 68000 has found its way into is enormous. You can even get 68000s as soft cores for FPGAs, which means you place a 68000 CPU in the midst of your programmable logic, all on the one chip.

The Motorola* MC68000 was introduced in 1979 as the successor to its 8-bit 6800 family. It featured a large address space, 32-bit registers, a large number of addressing modes, and an enlarged instruction set with over 1,000 opcodes. It was designed with the intention of running multitasking operating systems, specifically Unix. Its use in Unix machines has now long since passed, having been usurped by more advanced RISC processors. The 68000 processor was also used in the original Macintosh computers, as well as in the Atari ST, the Commodore Amiga, and Jef Raskin's CAT computer, all long extinct.



For an interesting overview of the CAT, read Jef Raskin's *The Humane Interface* (Addison-Wesley Professional). He discusses the CAT's unique design and has some interesting ideas on user interface design.

The processor's wide range of software and reasonable computing power are now encouraging its extensive use in embedded systems. It now forms the basis of a

* Motorola Semiconductor is now known as Freescale Semiconductor.

INTEL EXHIBIT 1008 Page 155
IPR2023-00787

family of microcontrollers designed for embedded systems, industrial control, networking, and PDAs. The 683xx series is the primary family of microcontrollers, specifically tailored to embedded applications. These processors combine a *CPU32* core (68020-based) with various integrated functions (such as UARTs, SPI, ADCs, etc.). Additional 68000 processors have been developed for specialized applications. The original Palm PDA has a 68EZ328 *DragonBall* processor, also based on a CPU32 core, that incorporates an LCD controller along with many of the common functions found in PDAs. The DragonBall is essentially a PDA on a chip—just add memory. The ucLinux fraternity uses a DragonBall processor in its small embedded controller board.

The 68000 architecture was upgraded to RISC with the *ColdFire* series of processors. These see extensive use in industrial control and network interfaces.

The 68000 series of processors are good general-purpose processors. They have a nice instruction set, are easy (and fun) to write code for, and are relatively easy to build computers around. They have large address spaces and asynchronous operation, allowing them to be interfaced to a wide variety of memory and peripherals of varying operating speeds. They are used in industrial control and monitoring, and also in consumer electronics.

In this chapter, we'll look at the standard 68000 processor. More than likely, this is not the processor you will use in a design. Rather, you will choose a 68000-based integrated controller that better suits your needs. So, why look at a standard 68000 and not one of the derivatives? First, there are far too many diverse 68000-based processors to cover. Second, since these processors are all based on the 68000, understanding the basic 68000 is a great starting point. Finally, all the derivatives are generally easier to use than the original, so if you can design around a standard 68000, then you can design for a derivative processor as well.

Understanding the 68000 gives you access to a wide range of available processors. There are dozens of commercial C compilers and assemblers available for the 68000 family, as well as a number of public-domain compilers as well. The 68000 is fully supported by the gnu development suite. Both Linux and BSD are also available for the 68000, as well as for numerous commercial operating systems.

The 68000 Architecture

The 68000 has eight 32-bit data registers (D0–D7), eight 32-bit address registers (A0–A7), a 32-bit program counter, two 32-bit stack pointers, and a 16-bit status register (Figure 18-1). The processor is capable of handling data as 32-bit long words, 16-bit words, bytes, or bits.

The processor has two modes of operation: *supervisor mode* (operating system) and *user mode* (applications). The mode of operation is made available to external

INTEL EXHIBIT 1008 Page 156
IPR2023-00787

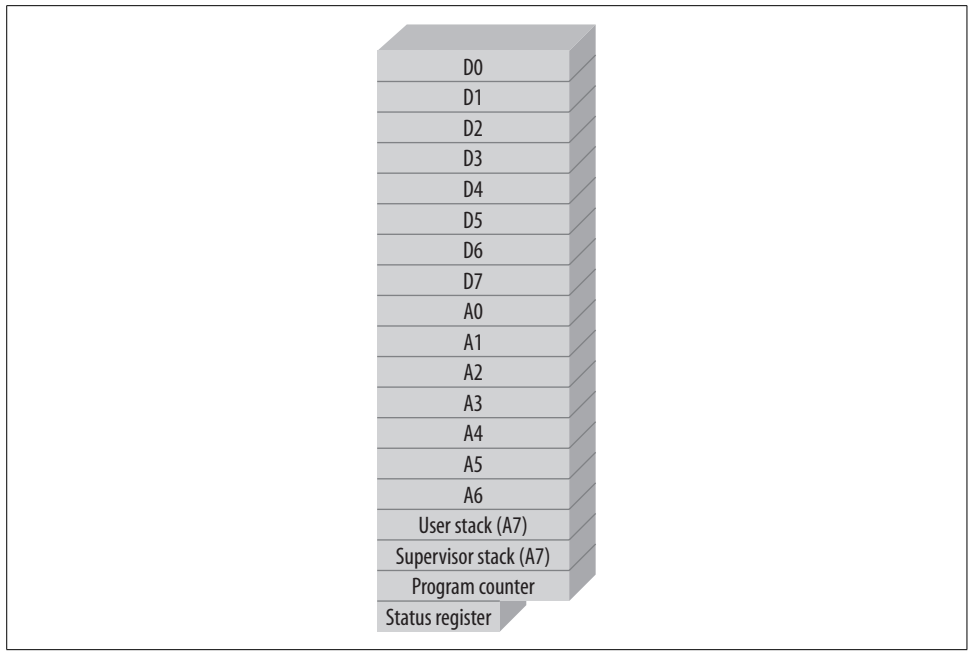


Figure 18-1. 68000 programmer's model

hardware, thereby giving the address decoder the ability to have separate supervisor and user spaces.

The standard 68000 is just a conventional, bus-based processor. A block diagram of a generic 68000-series processor is shown in Figure 18-2. The figure also shows the pins for a sample 68000-series processor. The pins and signals of 68000s can vary from one device to another, but they all have the same core functionality. The embedded controllers add to this basic functionality with additional I/O capability. We'll look at the pins for the MC68EC000 shortly. You can download datasheets, programmers' manuals, and technical references for 68000-series processors from the Freescale Semiconductor web site, *http://www.freescale.com*. While you're there, check out the other Freescale processor families, which range from 8-bit controllers (such as the 68HC05 and 68HC11) to 64-bit PowerPC RISC processors, and everything in between.

The original 68000 has a 23-bit address bus (A1 to A23), giving it access to a memory space of 16M, and a 16-bit data bus. Most other processors based on the 68000 architecture have address and data buses of 32 bits and can therefore access up to 4G of memory.

The processors have an input clock that drives all processor operation. Memory accesses typically take eight input clock cycles, provided that wait states are not introduced. Many processors based on the 68000 incorporate in-built address coding and software-configurable wait-state generation, making interfacing much simpler.

INTEL EXHIBIT 1008 Page 157
IPR2023-00787

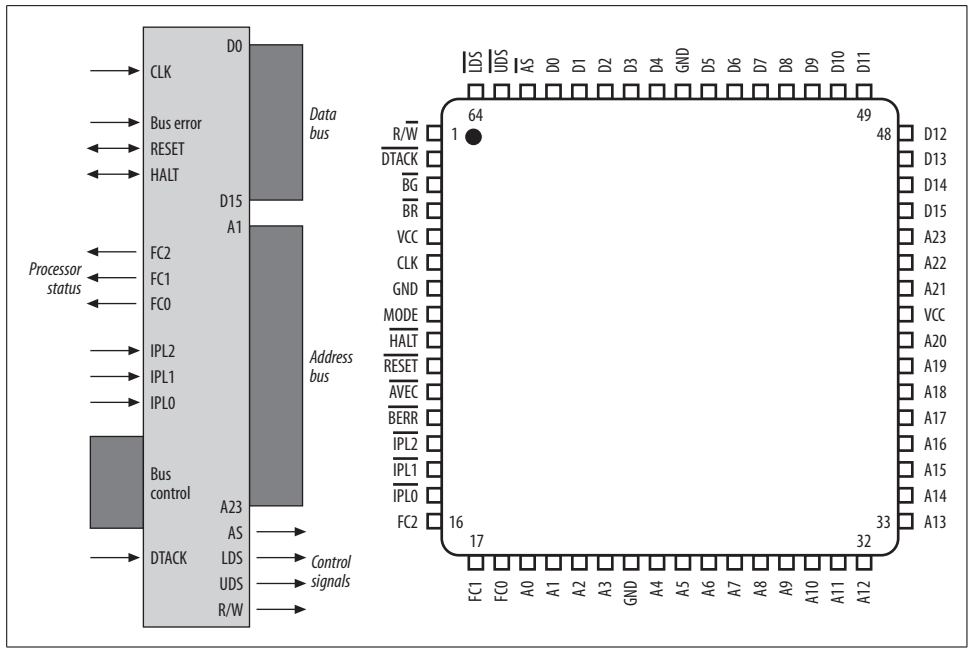


Figure 18-2. MC68000 block diagram and pinout

The processors have an address strobe (**AS**) indicating when a valid address is present on the bus, data strobes (**LDS**, **UDS**) indicating valid data, and a **R/W** line that shows the direction of the transfer. In addition, a Data Transfer Acknowledge input, **DTACK**, is used by external devices to indicate to the processor that it may terminate its current memory cycle. (Some 68000 processors call their Data Transfer Acknowledge **DTACKB**.) The function code outputs (**FC0**, **FC1**, and **FC2**) indicate the current operating mode (supervisor or user) of the processor. Bus Error (**BERR**) is used by an external address decoder to indicate an error condition. This allows the system to trap out accesses to unused regions of memory space, or in combination with the status lines, to detect user access to memory space allocated for supervisor use only. For example, if a program crashes and, in the process of crashing, attempts to access a region of memory to which no device is allocated, the address decoder is able to signal that fault back to the processor. An assertion of **BERR** causes the processor to execute an interrupt and take appropriate action. **HALT** is used to suspend processor operation without generating a reset. Three interrupt inputs (**IPL0**, **IPL1**, and **IPL2**) are used to generate seven levels of external interrupt handling. Bus Grant (**BG**) and Bus Request (**BR**) are DMA control signals by which another processor can arbitrate to acquire the computer's buses. The **MODE** pin, present on only some 68000 processors, determines whether the 68000 uses its data bus as 16 bits or 8 bits. **MODE** is sampled as the processor comes out of reset. **AVEC**, also found in only some 68000 processors, determines whether the processor uses auto-vectoring for its interrupts. If auto-vectoring is enabled, the processor will expect the

INTEL EXHIBIT 1008 Page 158
IPR2023-00787

interrupting peripheral to supply the appropriate vector. This allows a peripheral to specify what type of action the processor needs to take when a given interrupt is generated. Other 68000 processors may have other signals as well, but these are the main ones.

The basic timing diagram for a 68000 memory access is shown in Figure 18-3.

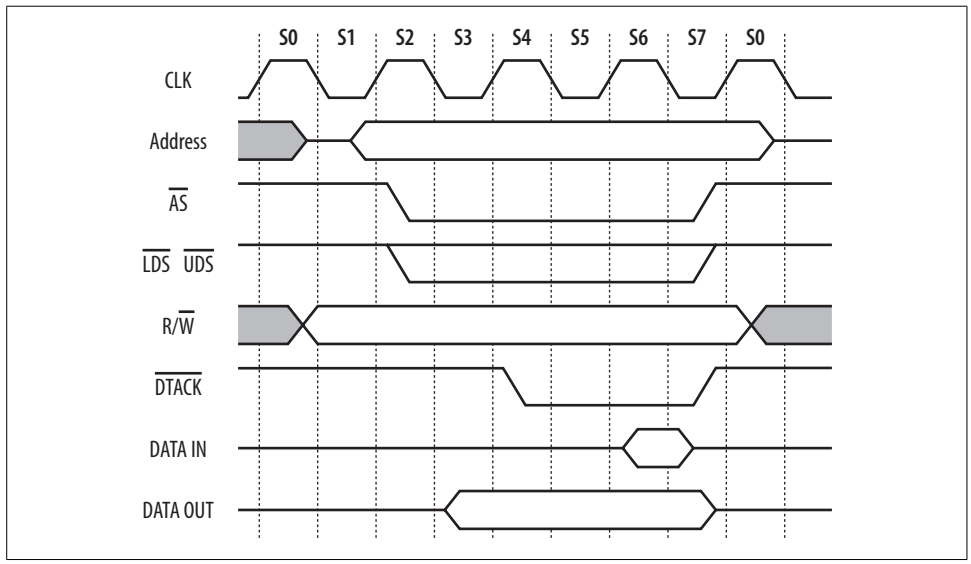


Figure 18-3. MC68000 timing diagram

The memory cycle of a 68000 is divided into a number of clock states, S0 through S7. The cycle begins with state S0. The processor validates **R/$\overline{\text{W}}$** for the coming cycle, sending it low for a write access, driving it high for a read access. The processor also tristates its address bus from the previous memory access. By S2, the processor has output a valid address and drives the address strobe (**$\overline{\text{AS}}$**) low, indicating that a valid address is present. The lower and upper data strobes (**$\overline{\text{LDS}}$** and **$\overline{\text{UDS}}$**) go low as appropriate and indicate the width of the memory access taking place. For a 16-bit transfer, both **$\overline{\text{LDS}}$** and **$\overline{\text{UDS}}$** assert. For an 8-bit transfer, only one of **$\overline{\text{LDS}}$** or **$\overline{\text{UDS}}$** asserts, depending on whether the upper byte or lower byte is being transferred. If the current memory access is a write cycle, the processor outputs valid data in state S3. At this point, all outputs from the processor are valid, and the processor waits for the device being accessed to respond.

At the falling edge of the clock in S4, the processor begins checking the state of the **DTACK** input. If **DTACK** is high, the processor inserts wait states and continues to do so until **$\overline{\text{DTACK}}$** is found to be low on the falling edge of the clock. (You'll learn how to generate wait states in "Wait States," later in the chapter.) When **$\overline{\text{DTACK}}$** is low, the processor recognizes this as an indication that the device being accessed has had sufficient time to respond and prepares to terminate the cycle. If the cycle is a

INTEL EXHIBIT 1008 Page 159
IPR2023-00787

read cycle, the processor will latch data on the falling edge of the clock in state S6. If it is a write cycle, the device being accessed will latch data as the data strobes go high in S7.

Support for synchronous operation is also provided for, using control signals found in the old 6800 series of processors. Since 6800s have long since passed into history, and 6800-based peripherals are now exceptionally rare, just ignore the 6800 control signals. Most 68000-based derivative processors no longer include support for 6800 peripherals.

A Simple 68000-Based Computer

Let's look now at a small 68000-based computer. For simplicity, we'll give it just a small amount of memory and a single peripheral, an MK68901 MFP (Multi-Function Peripheral) produced by ST Electronics. The MFP gives us a UART, parallel I/O, and interrupt control. A block diagram of the system is shown in Figure 18-4.

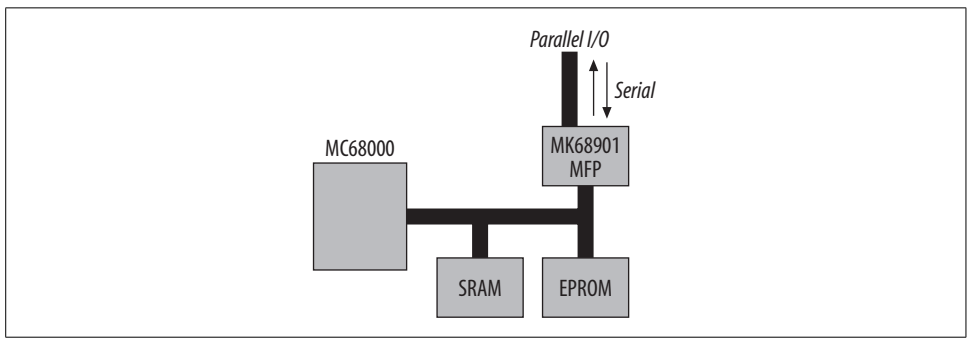


Figure 18-4. 68000-based computer

This system is designed with only a small amount of memory, to keep the design uncomplicated. While this is not much compared to many desktop machines, it is sufficient for many small, control applications. This design could be used for a number of simple applications. The counters of the MK68901 may be used to monitor external event pulses or to generate PWM for motor control. This computer could also be used to accept commands through its serial port and activate (or deactivate) external subsystems using the parallel I/O pins of the MK68901. This basic design could also be adapted to provide a bridge between an RS-232C interface and a parallel port. You could use this to interface a parallel-port printer to a serial-port-only computer. Alternatively, you could use it to put a serial modem on your PC's parallel port. Using the bus-interfacing techniques we learned in the AVR chapter, you could add additional peripherals such as ADCs and DACs, Ethernet, or a whole range of other devices. The list of possible applications is endless. And it all starts with this core design.

INTEL EXHIBIT 1008 Page 160
IPR2023-00787

So, let's start our tour of a 68000-based computer system. We'll look at the reset circuit, address decoder, I/O, and memory.

Reset Circuit

To reset an MC68000, both **RESET** and **HALT** must be driven low simultaneously. In addition, both of these signal lines may also act as outputs from the processor. Therefore, both must be independently driven by the reset circuit through open-collector gates. The conventional way of doing a 68000 reset circuit is shown in Figure 18-5.

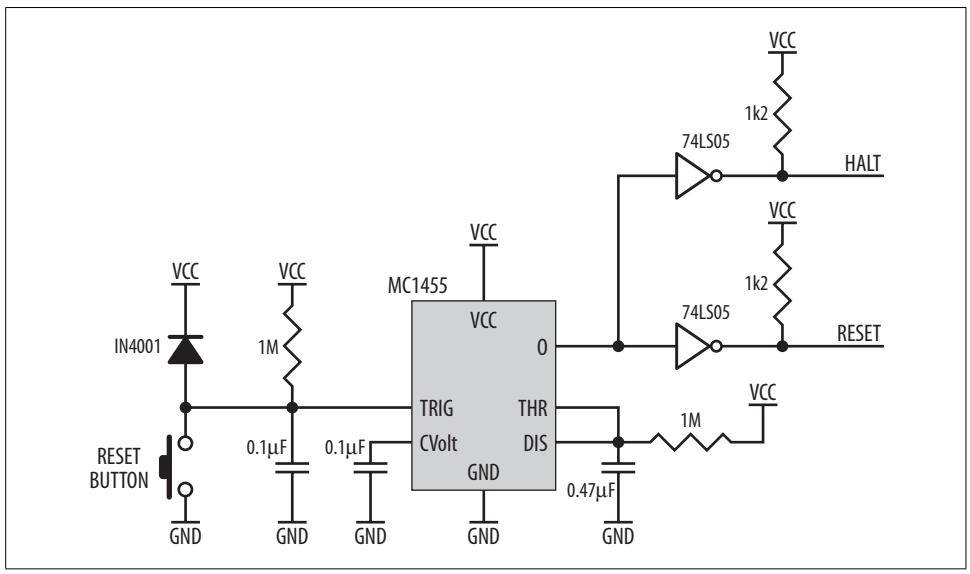


Figure 18-5. Reset circuit

The MC1455 will respond to a disruption on **VCC** by sending its output low. This output is used to drive **RESET** and **HALT** low simultaneously. In normal operation, **RESET** is held high by the pull-up resistor, unless pulled low through the reset switch being pressed. The diode is present to remove any glitches that might send **RESET** above **VCC**.

A better reset circuit is shown in Figure 18-6, using a MAX825 integrated reset controller. Again, both **RESET** and **HALT** need to be driven low.

Address Decoder

Logic to perform address decoding and the generation of separate read and write strobes is implemented in a PAL. In each case, **AS** (Address Strobe) of the processor

INTEL EXHIBIT 1008 Page 161
IPR2023-00787

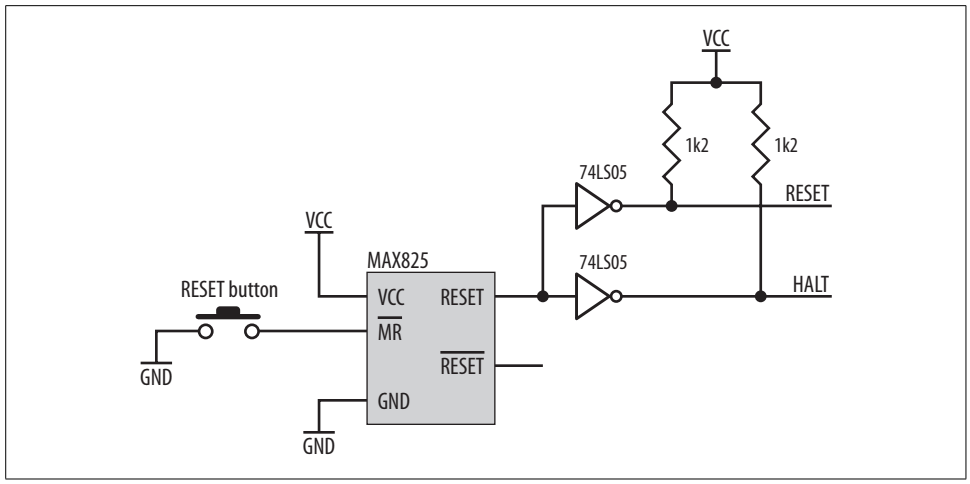


Figure 18-6. MAX825 reset circuit for a 68000

is used as an indication of a valid address present on the bus. The address-decode equations are as follows:

```
ROM = /(/AS * /A23 * /A22)
RAM0 = /(/AS * /A23 * A22 * /LDS)
RAM1 = /(/AS * /A23 * A22 * /UDS)
MFP = /(/AS * A23 * /A22)
```

With the exception of the MFP, which generates its own **DTACK**, **DTACK** for all other devices is generated as part of the address decoding. Since **DTACK** from the PAL must be OR-tied with **DTACK** from the MFP, it must be driven from an open-collector gate. Therefore, we generate an active-high acknowledge (which we'll designate **TACK**) from the PAL and invert this through a 74LS05 open-collector inverter.

The PAL equation to generate **TACK** is simply:

```
TACK = (/AS * MFP)
```

Therefore, **TACK** is active (high) whenever the processor accesses its address space, so long as it is not accessing the MFP. If the address strobe is high, or if there is an access to the MFP, then **TACK** is low. The **TACK** output from the PAL is inverted through an open-collector 74LS05 and "OR-tied" (directly connected together) with **DTACK** from the MFP. **DTACK** requires a pull-up 1 kΩ resistor, since this input must have a sharp rise time. A block diagram is shown in Figure 18-7.

No provision for generating a **BERR** is made because our simple address decoding allocates all of the address space. If we had any unused regions of the memory space, we would use our address decoder to generate a **BERR** when accesses to the unused regions were made.

INTEL EXHIBIT 1008 Page 162
IPR2023-00787

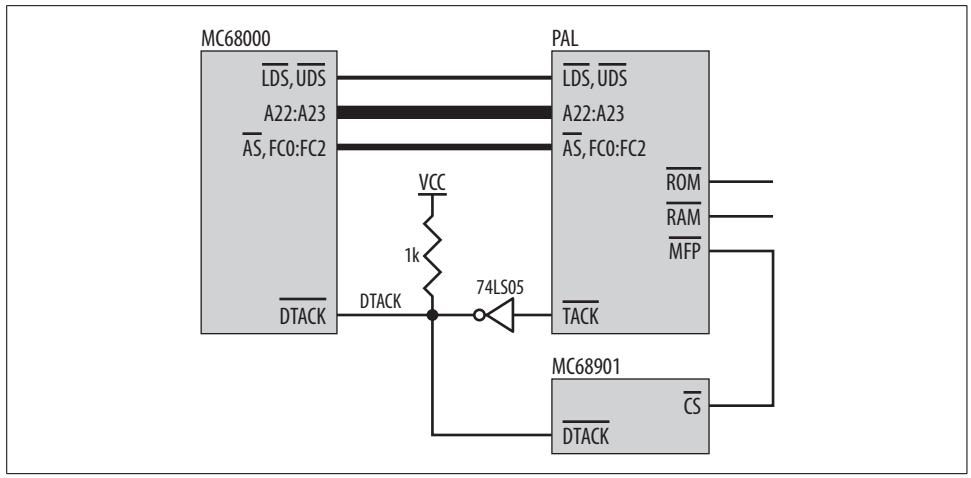


Figure 18-7. Address decode and DTACK generation

The PAL equations to generate separate read and write strobes for the memory chips are:

```
UWE = /(/UDS * RW)
LWE = /(/LDS * RW)
UOE = /(/UDS * /RW)
LOE = /(/LDS * /RW)
```

The connections for the PAL are shown in Figure 18-8. Additional addresses are brought into the PAL to allow for future changes to the memory map. The processor's clock (**CLK**) is used by the PAL to generate the clock for the MFP (**MFPCLK**).

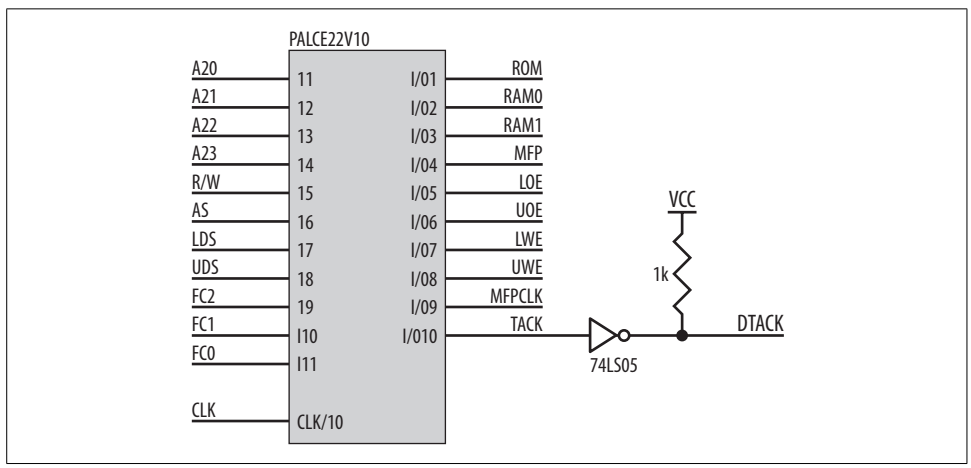


Figure 18-8. Address decode and system-logic PAL

INTEL EXHIBIT 1008 Page 163
IPR2023-00787

The function code outputs (**FC0**–**FC2**) can be decoded using a 74LS138 demultiplexer to drive three LEDs (Figure 18-9). These provide a visible indication of processor status. The function codes could also be used by the address decoder if you wanted to have separate user and supervisor address spaces. Many of the more sophisticated peripheral chips (such as the MFP) require the processor to acknowledge when they have generated an interrupt. The 74LS138 also uses the function codes to generate an interrupt acknowledge (**IACK**) for peripherals, since the function codes also indicate an IACK condition.

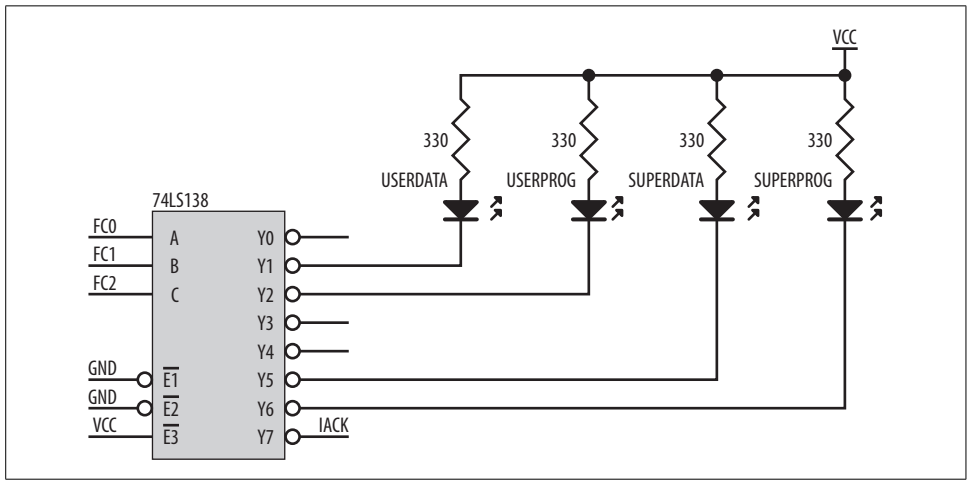


Figure 18-9. Status LEDs indicating processor mode

I/O

The MK68901 Multifunction Peripheral (MFP) provides a serial port, as well as basic parallel I/O functions, a 16-source interrupt controller, and four 8-bit timers. The MK68901 has an internal oscillator that drives the internal timers. A timer output (**TD0**) is fed back into the MFP as the clock for the serial interface. The internal oscillator must therefore run at a frequency appropriate for RS-232C. An external 3.6864 MHz crystal drives the oscillator. This input clock can be divided up by the MFP, providing the appropriate baud rates for the serial port. The serial lines from the MFP are converted to RS-232C voltage levels by a MAX3232 level shifter. A 9-pin, D-type connector provides access to the RS-232C signals. The parallel I/O lines and timer inputs and outputs are also made available through a 26-pin IDC connector.

The schematic for the MFP is shown in Figure 18-10.

INTEL EXHIBIT 1008 Page 164
IPR2023-00787

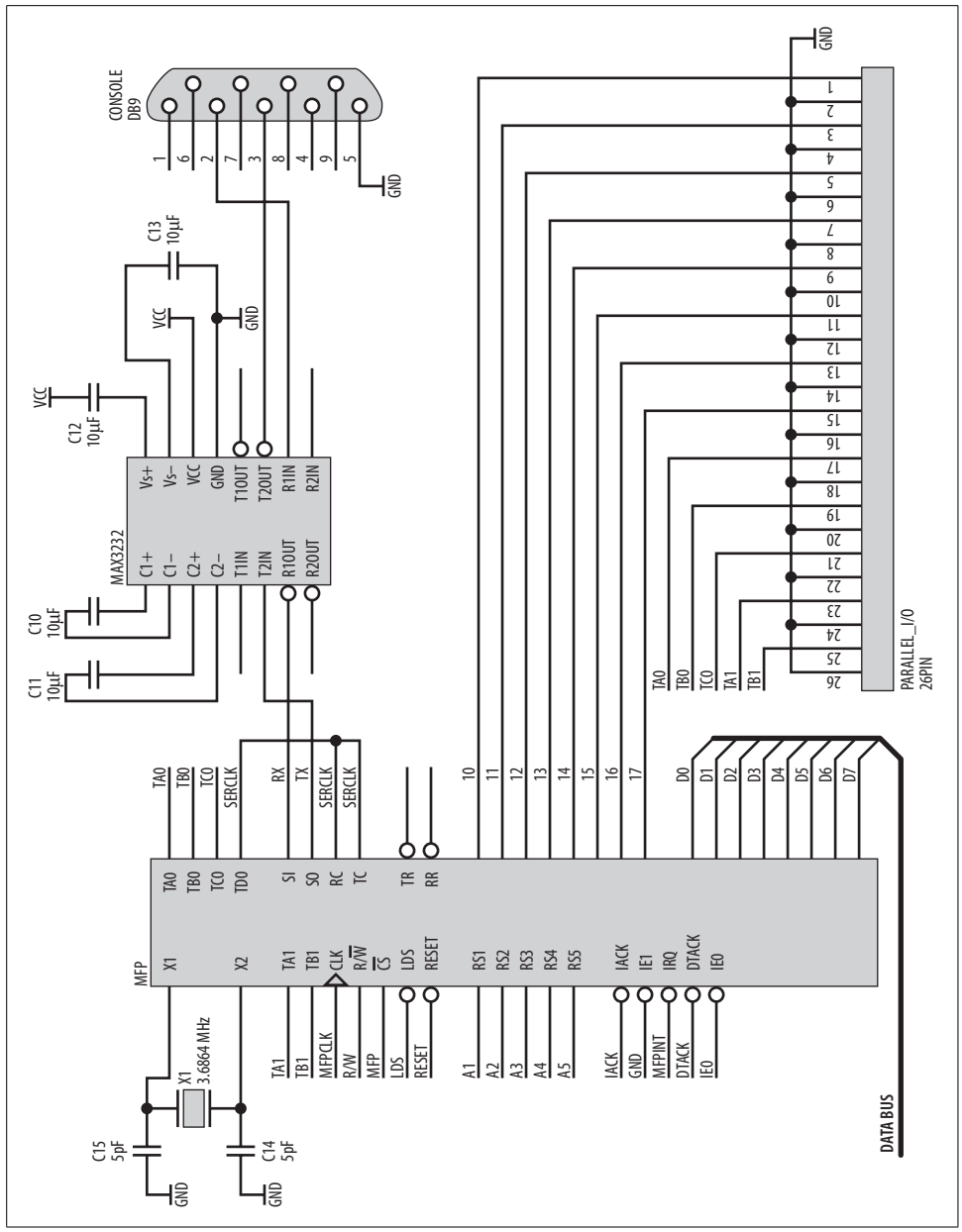


Figure 18-10. Multifunction Peripheral

Memory

The system is designed with 256K of EPROM and 512K of static RAM. The connections to the SRAM are shown in Figure 18-11. Note that since the data bus of a 68000 is 16 bits wide, two SRAMs are required. For 68000-based derivatives with

INTEL EXHIBIT 1008 Page 165
IPR2023-00787

32-bit external data buses, four memory chips would be required in parallel. Note how half the data bus goes to one chip and the other half goes to the other chip.

Now, note the address lines going to the SRAMs. The lowest address bit from the processor is **A1**, and this is connected to the **A0** inputs of the SRAMs, and so on. Since the processor accesses external memory in 16-bit words, **A1** represents the least significant address bit. In other words, as you move from word to subsequent word in memory, it is **A1** that increments. However, **A0** is the least significant address bit of the SRAMs, but since the two SRAMs together form a 16-bit word of memory, the **A0** of the SRAMs must connect to **A1** of the processor. The other address bits follow on from that starting point.

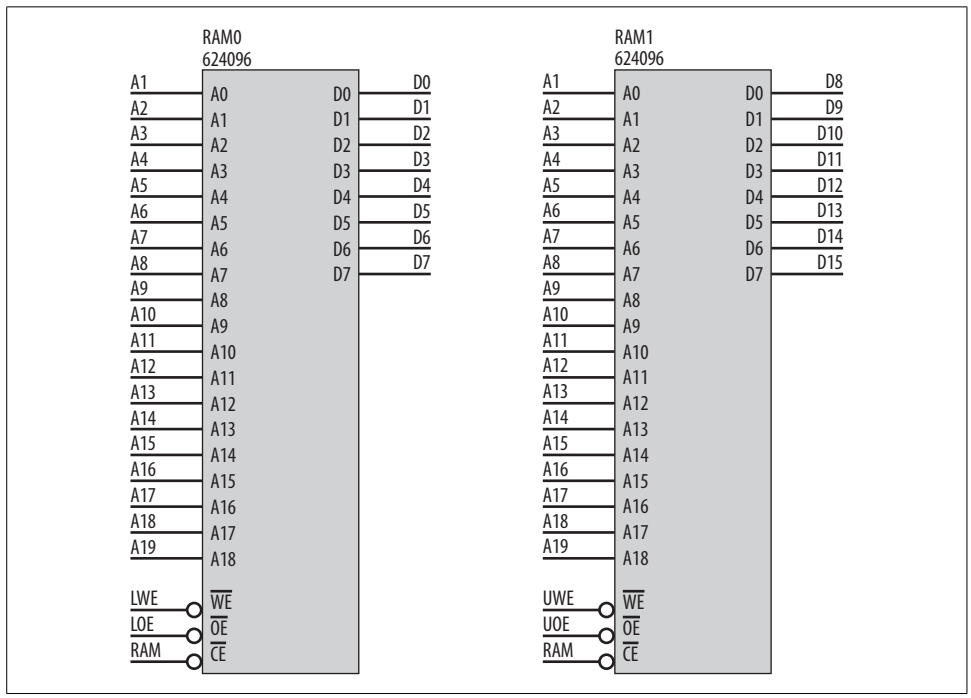


Figure 18-11. Interfacing to SRAM

Similarly, the connections for the ROMs are shown in Figure 18-12.

Wait States

Depending on the speed of your processor and the access times of your memory and peripheral chips, it may be necessary to introduce wait states into the 68000's memory cycle. Wait-state generation follows basically the same principle for processors that support asynchronous memory cycles. The processor will have an input (sometimes more than one) that will cause it to delay the memory cycle, giving slower

INTEL EXHIBIT 1008 Page 166
IPR2023-00787

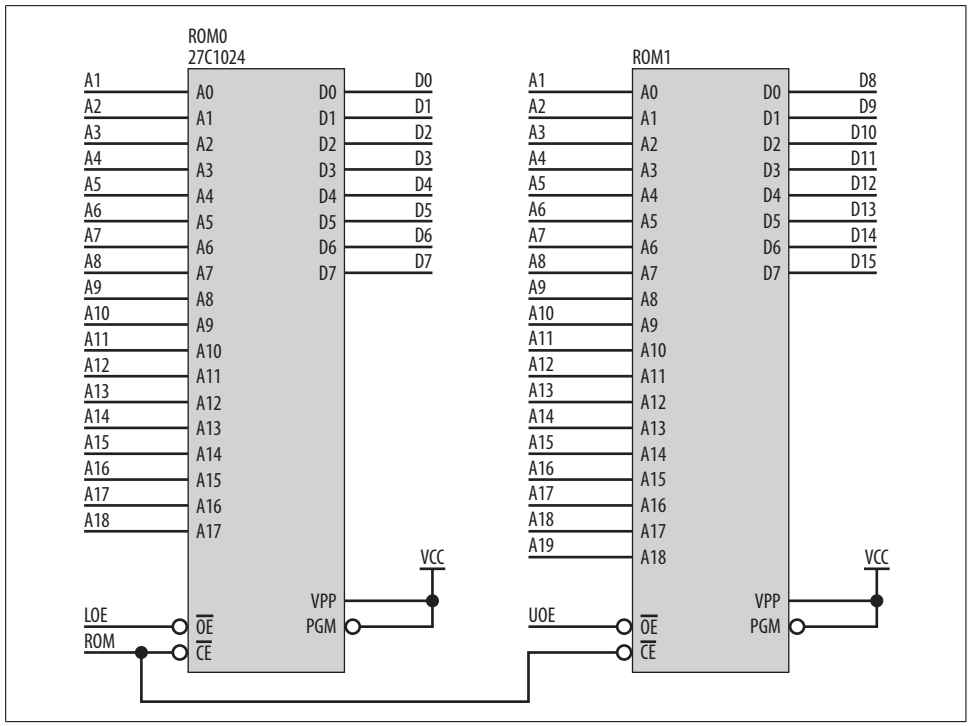


Figure 18-12. Interfacing to EPROMs

devices time to respond. In the case of the 68000, that input is **DTACK**. To insert a wait state for a given device, we need to detect an access to that device and hold **DTACK** inactive for the required additional clock cycles. In other words, we need to use the chip select for a given device to delay **DTACK** going low. The circuit to do this is simple and is best done inside a PAL or other programmable logic device. This facilitates changing the wait-state generator if faster parts are used in the design at a later stage. The wait-state generator consists of a series of D-type flip-flops* (Figure 18-13). Each flip-flop represents an additional clock cycle that the transfer acknowledge is delayed.

Between memory cycles, the address strobe, **AS**, goes high. It is first inverted and then connected to the active-low **SET** input of each of the flip-flops. Thus, the output of each of the flip-flops is driven high between each memory cycle. This resets them from any previous cycle. The address decoder generates a chip select for the particular device, and this is connected to the D input of the first flip-flop. So, on each successive clock pulse, the 0 provided by the chip select is clocked through from

* A flip-flop is a logic element that feeds the D input through to the Q output on the changing edge of a clock pulse.

INTEL EXHIBIT 1008 Page 167
IPR2023-00787

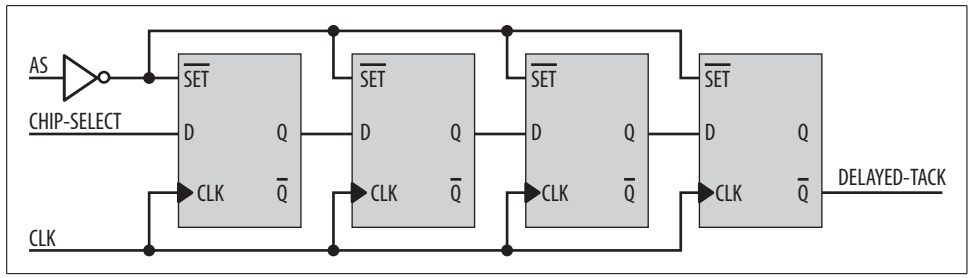


Figure 18-13. Wait-state generator

one flip-flop to the next. After four clock pulses, the 0 has arrived at the Q output of the last flip-flop. The inverted output of this flip-flop, $\overline{Q}$, becomes a 1. This is then output by the PAL to be inverted by the 74LS05 open-collector inverter to provide **DTACK** for the processor. For additional wait states, add more flip-flops. For several devices requiring different numbers of wait states, use their combined chip selects to feed the D input of the first flip-flop; then "tap" into the wait-state generator at different stages for the required delay. Each of these taps is gated with the respective chip select to enable/disable that output before they are all recombined to generate a unified acknowledge for the processor.

Most processors that support wait states now include inbuilt, software-configurable wait-state generators. This makes the task of designing the system logic much simpler.

In the next chapter, we'll take a look at a different sort of embedded processor—one based on a Digital Signal Processing (DSP) architecture.

INTEL EXHIBIT 1008 Page 168
IPR2023-00787