



INCLUDES

NEWNES ONLINE
MEMBERSHIP

EMBEDDED HARDWARE

know it all

- A 360 degree view from our best-selling authors
- Key facts, techniques, and applications fully detailed
- The ultimate hard-working desk reference:
all the essential information, techniques, and tricks
of the trade in one volume

Ganssle • Noergaard • Eady • Edwards
Katz • Gentile • Arnold • Hyder • Perrin

Embedded Hardware

Newnes Know It All Series

PIC Microcontrollers: Know It All

Lucio Di Jasio, Tim Wilmshurst, Dogan Ibrahim, John Morton,
Martin Bates, Jack Smith, D.W. Smith, and Chuck Hellebuyck
ISBN: 978-0-7506-8615-0

Embedded Software: Know It All

Jean Labrosse, Jack Ganssle, Tammy Noergaard, Robert Oshana, Colin Walls, Keith Curtis,
Jason Andrews, David J. Katz, Rick Gentile, Kamal Hyder, and Bob Perrin
ISBN: 978-0-7506-8583-2

Embedded Hardware: Know It All

Jack Ganssle, Tammy Noergaard, Fred Eady, Creed Huddleston, Lewin Edwards,
David J. Katz, Rick Gentile, Ken Arnold, Kamal Hyder, and Bob Perrin
ISBN: 978-0-7506-8584-9

Wireless Networking: Know It All

Praphul Chandra, Daniel M. Dobkin, Alan Bensky, Ron Olexa,
David A. Lide, and Farid Dowla
ISBN: 978-0-7506-8582-5

RF & Wireless Technologies: Know It All

Bruce Fette, Roberto Aiello, Praphul Chandra, Daniel Dobkin,
Alan Bensky, Douglas Miron, David A. Lide, Farid Dowla, and Ron Olexa
ISBN: 978-0-7506-8581-8

For more information on these and other Newnes titles visit: www.newnespress.com

Embedded Hardware

Jack Ganssle
Tammy Noergaard
Fred Eady
Lewin Edwards
David J. Katz
Rick Gentile
Ken Arnold
Kamal Hyder
Bob Perrin
Creed Huddleston



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier



Cover image by iStockphoto

Newnes is an imprint of Elsevier

30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Copyright © 2008 by Elsevier Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (<http://elsevier.com>), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."



Recognizing the importance of preserving what has been written, Elsevier prints its books on acid-free paper whenever possible.

Library of Congress Cataloging-in-Publication Data

Ganssle, Jack G.

Embedded hardware / Jack Ganssle ... [et al.].

p. cm.

Includes index.

ISBN 978-0-7506-8584-9 (alk. paper)

1. Embedded computer systems. I. Title.

TK7895.E42G37 2007

004.16—dc22

2007027559

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

For information on all Newnes publications
visit our Web site at www.books.elsevier.com

07 08 09 10 10 9 8 7 6 5 4 3 2 1

Typeset by Charon Tec Ltd (A Macmillan Company), Chennai, India

www.charontec.com

Printed in the United States of America

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

Contents

About the Authors.....	xiii
Chapter 1: Embedded Hardware Basics	1
1.1 Lesson One on Hardware: Reading Schematics.....	1
1.2 The Embedded Board and the von Neumann Model	5
1.3 Powering the Hardware	9
1.3.1 A Quick Comment on Analog Vs. Digital Signals.....	10
1.4 Basic Electronics	12
1.4.1 DC Circuits	12
1.4.2 AC Circuits	21
1.4.3 Active Devices	28
1.5 Putting It Together: A Power Supply	32
1.5.1 The Scope	35
1.5.2 Controls.....	35
1.5.3 Probes.....	38
Endnotes	41
Chapter 2: Logic Circuits.....	43
2.1 Coding	43
2.1.1 BCD	46
2.2 Combinatorial Logic.....	47
2.2.1 NOT Gate	47
2.2.2 AND and NAND Gates.....	48
2.2.3 OR and NOR Gates.....	49
2.2.4 XOR	50
2.2.5 Circuits	50
2.2.6 Tristate Devices.....	53
2.3 Sequential Logic.....	53
2.3.1 Logic Wrap-Up	57
2.4 Putting It All Together: The Integrated Circuit	58
Endnotes	61

Chapter 3: Embedded Processors.....	63
3.1 Introduction	63
3.2 ISA Architecture Models.....	65
3.2.1 Operations	65
3.2.2 Operands	68
3.2.3 Storage	69
3.2.4 Addressing Modes.....	71
3.2.5 Interrupts and Exception Handling	72
3.2.6 Application-Specific ISA Models	72
3.2.7 General-Purpose ISA Models	74
3.2.8 Instruction-Level Parallelism ISA Models.....	76
3.3 Internal Processor Design.....	78
3.3.1 Central Processing Unit (CPU)	82
3.3.2 On-Chip Memory	99
3.3.3 Processor Input/Output (I/O).....	113
3.3.4 Processor Buses.....	130
3.4 Processor Performance	131
3.4.1 Benchmarks.....	133
Endnotes	133
Chapter 4: Embedded Board Buses and I/O	137
4.1 Board I/O	137
4.2 Managing Data: Serial vs. Parallel I/O.....	140
4.2.1 Serial I/O Example 1: Networking and Communications: RS-232	144
4.2.2 Example: Motorola/Freescale MPC823 FADS Board RS-232 System Model	146
4.2.3 Serial I/O Example 2: Networking and Communications: IEEE 802.11 Wireless LAN	148
4.2.4 Parallel I/O	153
4.2.5 Parallel I/O Example 3: “Parallel” Output and Graphics I/O	153
4.2.6 Parallel and Serial I/O Example 4: Networking and Communications—Ethernet.....	156
4.2.7 Example 1: Motorola/Freescale MPC823 FADS Board Ethernet System Model	158
4.2.8 Example 2: Net Silicon ARM7 (6127001) Development Board Ethernet System Model	160
4.2.9 Example 3: Adastr Neptune x86 Board Ethernet System Model.....	161
4.3 Interfacing the I/O Components	161
4.3.1 Interfacing the I/O Device with the Embedded Board.....	162
4.3.2 Interfacing an I/O Controller and the Master CPU	164

4.4 I/O and Performance	165
4.5 Board Buses.....	166
4.6 Bus Arbitration and Timing.....	168
4.6.1 Nonexpandable Bus: I ² C Bus Example	174
4.6.2 PCI (Peripheral Component Interconnect) Bus Example: Expandable	175
4.7 Integrating the Bus with Other Board Components	179
4.8 Bus Performance	180
Chapter 5: Memory Systems.....	183
5.1 Introduction	183
5.2 Memory Spaces	183
5.2.1 L1 Instruction Memory	186
5.2.2 Using L1 Instruction Memory for Data Placement.....	186
5.2.3 L1 Data Memory	187
5.3 Cache Overview	187
5.3.1 What Is Cache?	188
5.3.2 Direct-Mapped Cache	190
5.3.3 Fully Associative Cache.....	190
5.3.4 N-Way Set-Associative Cache	191
5.3.5 More Cache Details.....	191
5.3.6 Write-Through and Write-Back Data Cache.....	193
5.4 External Memory.....	195
5.4.1 Synchronous Memory	195
5.4.2 Asynchronous Memory.....	203
5.4.3 Nonvolatile Memories.....	206
5.5 Direct Memory Access.....	214
5.5.1 DMA Controller Overview	215
5.5.2 More on the DMA Controller	216
5.5.3 Programming the DMA Controller	218
5.5.4 DMA Classifications	228
5.5.5 Register-Based DMA	228
5.5.6 Descriptor-Based DMA	231
5.5.7 Advanced DMA Features.....	234
Endnotes	236
Chapter 6: Timing Analysis in Embedded Systems.....	239
6.1 Introduction	239
6.2 Timing Diagram Notation Conventions	239
6.2.1 Rise and Fall Times.....	241

6.2.2 Propagation Delays	241
6.2.3 Setup and Hold Time.....	241
6.2.4 Tri-State Bus Interfacing.....	243
6.2.5 Pulse Width and Clock Frequency	244
6.3 Fan-Out and Loading Analysis: DC and AC.....	244
6.3.1 Calculating Wiring Capacitance.....	247
6.3.2 Fan-Out When CMOS Drives LSTTL	249
6.3.3 Transmission-Line Effects	251
6.3.4 Ground Bounce	253
6.4 Logic Family IC Characteristics and Interfacing	255
6.4.1 Interfacing TTL Compatible Signals to 5 V CMOS	258
6.5 Design Example: Noise Margin Analysis Spreadsheet	261
6.6 Worst-Case Timing Analysis Example.....	270
Endnotes	272
Chapter 7: Choosing a Microcontroller and Other Design Decisions	273
7.1 Introduction	273
7.2 Choosing the Right Core	276
7.3 Building Custom Peripherals with FPGAs.....	281
7.4 Whose Development Hardware to Use—Chicken or Egg?.....	282
7.5 Recommended Laboratory Equipment.....	285
7.6 Development Toolchains	286
7.7 Free Embedded Operating Systems	289
7.8 GNU and You: How Using “Free” Software Affects Your Product.....	295
Chapter 8: The Essence of Microcontroller Networking: RS-232.....	301
8.1 Introduction	301
8.2 Some History	303
8.3 RS-232 Standard Operating Procedure	305
8.4 RS-232 Voltage Conversion Considerations	308
8.5 Implementing RS-232 with a Microcontroller	310
8.5.1 Basic RS-232 Hardware	310
8.5.2 Building a Simple Microcontroller RS-232 Transceiver	313
8.6 Writing RS-232 Microcontroller Routines in BASIC	333
8.7 Building Some RS-232 Communications Hardware.....	339
8.7.1 A Few More BASIC RS-232 Instructions.....	339
8.8 I ² C: The Other Serial Protocol	342
8.8.1 Why Use I ² C?.....	343
8.8.2 The I ² C Bus.....	344
8.8.3 I ² C ACKS and NAKS	347

8.8.4 More on Arbitration and Clock Synchronization.....	347
8.8.5 I ² C Addressing	351
8.8.6 Some I ² C Firmware.....	352
8.8.7 The AVR Master I ² C Code.....	352
8.8.8 The AVR I ² C Master-Receiver Mode Code	358
8.8.9 The PIC I ² C Slave-Transmitter Mode Code	359
8.8.10 The AVR-to-PIC I ² C Communications Ball	365
8.9 Communication Options.....	378
8.9.1 The Serial Peripheral Interface Port.....	378
8.9.2 The Controller Area Network	380
8.9.3 Acceptance Filters.....	386
Endnote.....	387
Chapter 9: Interfacing to Sensors and Actuators	389
9.1 Introduction	389
9.2 Digital Interfacing	389
9.2.1 Mixing 3.3 and 5 V Devices	389
9.2.2 Protecting Digital Inputs	392
9.2.3 Expanding Digital Inputs	398
9.2.4 Expanding Digital Outputs.....	402
9.3 High-Current Outputs.....	404
9.3.1 BJT-Based Drivers	405
9.3.2 MOSFETs	409
9.3.3 Electromechanical Relays	411
9.3.4 Solid-State Relays	417
9.4 CPLDs and FPGAs.....	418
9.5 Analog Interfacing: An Overview	420
9.5.1 ADCs.....	420
9.5.2 Project 1: Characterizing an Analog Channel.....	421
9.6 Conclusion.....	434
Endnote.....	435
Chapter 10: Other Useful Hardware Design Tips and Techniques	437
10.1 Introduction	437
10.2 Diagnostics	437
10.3 Connecting Tools.....	438
10.4 Other Thoughts.....	439
10.5 Construction Methods	440
10.5.1 Power and Ground Planes.....	441
10.5.2 Ground Problems	441

10.6	Electromagnetic Compatibility	442
10.7	Electrostatic Discharge Effects	442
10.7.1	Fault Tolerance	443
10.8	Hardware Development Tools	444
10.8.1	Instrumentation Issues	445
10.9	Software Development Tools	445
10.10	Other Specialized Design Considerations	446
10.10.1	Thermal Analysis and Design	446
10.10.2	Battery-Powered System Design Considerations	447
10.11	Processor Performance Metrics	448
10.11.1	IPS	448
10.11.2	OPS	448
10.11.3	Benchmarks	449
Appendix A: Schematic Symbols		451
Appendix B: Acronyms and Abbreviations		459
Appendix C: PC Board Design Issues		469
C.1	Introduction	469
C.2	Resistance of Conductors	470
C.3	Voltage Drop in Signal Leads—"Kelvin" Feedback	471
C.4	Signal Return Currents	472
C.5	Grounding in Mixed Analog/Digital Systems	474
C.6	Ground and Power Planes	475
C.7	Double-Sided versus Multilayer Printed Circuit Boards	477
C.8	Multicard Mixed-Signal Systems	478
C.9	Separating Analog and Digital Grounds	479
C.10	Grounding and Decoupling Mixed-Signal ICs with Low Digital Currents	480
C.11	Treat the ADC Digital Outputs with Care	481
C.12	Sampling Clock Considerations	483
C.13	The Origins of the Confusion About Mixed-Signal Grounding: Applying Single-Card Grounding Concepts to Multicard Systems	485
C.14	Summary: Grounding Mixed-Signal Devices with Low Digital Currents in a Multicard System	486
C.15	Summary: Grounding Mixed-Signal Devices with High Digital Currents in a Multicard System	487
C.16	Grounding DSPs with Internal Phase-Locked Loops	487
C.17	Grounding Summary	488
C.18	Some General PC Board Layout Guidelines for Mixed-Signal Systems	489

C.19 Skin Effect	491
C.20 Transmission Lines	493
C.21 Be Careful with Ground Plane Breaks.....	494
C.22 Ground Isolation Techniques	495
C.23 Static PCB Effects	497
C.24 Sample MINIDIP and SOIC Op Amp PCB Guard Layouts.....	500
C.25 Dynamic PCB Effects.....	502
C.26 Stray Capacitance	503
C.27 Capacitive Noise and Faraday Shields.....	504
C.28 The Floating Shield Problem	506
C.29 Buffering ADCs Against Logic Noise.....	506
Endnotes	509
Acknowledgments	509
Index	511

This page intentionally left blank

About the Authors

Ken Arnold (Chapters 6 and 10) is the author of *Embedded Controller Hardware Design*. He is the Embedded Computer Engineering Program Coordinator and an instructor at UCSD Extension, as well as founding director of the On-Line University of California, where he manages, develops and teaches courses in engineering and embedded systems design. Ken has been developing commercial embedded systems and teaching others how for more than two decades. As the champion of the embedded program at UCSD, he lead the inception and growth of the program as well as introducing the world's first on-line embedded course well over a decade ago. Ken was also the founder and CEO of HiTech Equipment Corp., CTO of Wireless Innovation, and engineering chief at General Dynamics.

Fred Eady (Chapter 8) is the author of *Networking and Internetworking with Microcontrollers*. As an engineering consultant, he has implemented communications networks for the space program and designed hardware and firmware for the medical, retail and public utility industries. He currently writes a monthly embedded design column for a popular electronics enthusiast magazine. Fred also composes monthly articles for a popular robotics magazine. Fred has been dabbling in electronics for over 30 years. His embedded design expertise spans the spectrum and includes Intel's 8748 and 8051 microcontrollers, the entire Microchip PIC microcontroller family and the Atmel AVR microcontrollers. Fred recently retired from his consulting work and is focused on writing magazine columns and embedded design books.

Lewin Edwards (Chapter 7) is the author of *Embedded System Design on a Shoestring*. He hails from Adelaide, Australia. His career began with five years of security and encryption software at PC-Plus Systems. The next five years were spent developing networkable multimedia appliances at Digi-Frame in Port Chester, NY. Since 2004 he has been developing security and fire safety devices at a Fortune 100 company in New York. He has written numerous technical articles and three embedded systems books, with a fourth due in early 2008.

Jack Ganssle (Chapters 1, 2, and 10) is the author of *The Firmware Handbook*. He has written over 500 articles and six books about embedded systems, as well as a book about his sailing fiascos. He started developing embedded systems in the early 70s using the 8008. He's started and sold three electronics companies, including one of the bigger embedded tool businesses. He's developed or managed over 100 embedded products, from deep-sea navigation gear to

the White House security system... and one instrument that analyzed cow poop! He's currently a member of NASA's Super Problem Resolution Team, a group of outside experts formed to advise NASA in the wake of Columbia's demise, and serves on the boards of several high-tech companies. Jack now gives seminars to companies world-wide about better ways to develop embedded systems.

Rick Gentile (Chapter 5) is the author of *Embedded Media Processing*. Rick joined ADI in 2000 as a Senior DSP Applications Engineer, and he currently leads the Processor Applications Group, which is responsible for Blackfin, SHARC and TigerSHARC processors. Prior to joining ADI, Rick was a Member of the Technical Staff at MIT Lincoln Laboratory, where he designed several signal processors used in a wide range of radar sensors. He has authored dozens of articles and presented at multiple technical conferences. He received a B.S. in 1987 from the University of Massachusetts at Amherst and an M.S. in 1994 from Northeastern University, both in Electrical and Computer Engineering.

Creed Huddleston (Chapter 8) is the author of *Intelligent Sensor Design Using the Microchip dsPIC*. With over twenty years of experience designing real-time embedded systems, he is President and founder of Real-Time by Design, LLC, a certified Microchip Design Partner based in Raleigh, NC that specializes in the creation of hard real-time intelligent sensing systems. In addition to his duties with Real-Time by Design, Creed also serves on the Advisory Board of Quickfilter Technologies Inc., a Texas-based company producing mixed-signal integrated circuits that provide high-speed analog signal conditioning and digital signal processing in a single package. A graduate of Rice University in Houston, TX with a BSEE degree, Creed performed extensive graduate work in digital signal processing at the University of Texas at Arlington before heading east to start Omnisys. To her great credit and his great fortune, Creed and his wife Lisa have been married for 23 years and have three wonderful children: Kate, Beth, and Dan.

Kamal Hyder (Chapter 9) is the author of *Embedded Systems Design Using the Rabbit 3000 Microprocessor*. He started his career with an embedded microcontroller manufacturer. He then wrote CPU microcode for Tandem Computers for a number of years, and was a Product Manager at Cisco Systems, working on next-generation switching platforms. He is currently with Brocade Communications as Senior Group Product Manager. Kamal's BS is in EE/CS from the University of Massachusetts, Amherst, and he has an MBA in finance/marketing from Santa Clara University.

David Katz (Chapter 5) is the author of *Embedded Media Processing*. He has over 15 years of experience in circuit and system design. Currently, he is the Blackfin Applications Manager at Analog Devices, Inc., where he focuses on specifying new convergent processors. He has published over 100 embedded processing articles domestically and internationally, and he has presented several conference papers in the field. Previously, he worked at Motorola, Inc., as a

senior design engineer in cable modem and automation groups. David holds both a B.S. and M. Eng. in Electrical Engineering from Cornell University.

Walt Kester (Appendix C) is the editor of *Data Conversion Handbook*. He is a corporate staff applications engineer at Analog Devices. For more than 35 years at Analog Devices, he has designed, developed, and given applications support for high-speed ADCs, DACs, SHAs, op amps, and analog multiplexers. Besides writing many papers and articles, he prepared and edited eleven major applications books which form the basis for the Analog Devices world-wide technical seminar series including the topics of op amps, data conversion, power management, sensor signal conditioning, mixed-signal, and practical analog design techniques. Walt has a BSEE from NC State University and MSEE from Duke University.

Tammy Noergaard (Chapters 1, 2, 3, 4, Appendices B and C) is the author of *Embedded Systems Architecture*. Since beginning her embedded systems career in 1995, she has had wide experience in product development, system design and integration, operations, sales, marketing, and training. Noergaard worked for Sony as a lead software engineer developing and testing embedded software for analog TVs. At Wind River she was the liaison engineer between developmental engineers and customers to provide design expertise, systems configuration, systems integration, and training for Wind River embedded software (OS, Java, device drivers, etc.) and all associated hardware for a variety of embedded systems in the Consumer Electronic market. Most recently she was a Field Engineering Specialist and Consultant with Esmertec North America, providing project management, system design, system integration, system configuration, support and expertise for various embedded Java systems using Jbed in everything from control systems to medical devices to digital TVs. Noergaard has lectured to engineering classes at the University of California at Berkeley and Stanford, the Embedded Internet Conference, and the Java User's Group in San Jose, among others.

Bob Perrin (Chapter 9) is the author of *Embedded Systems Design Using the Rabbit 3000 Microprocessor*. He got his start in electronics at the age of nine when his mother gave him a "150-in-one Projects" kit from Radio Shack for Christmas. He grew up programming a Commodore PET. In 1990, Bob graduated with a BSEE from Washington State University. Since then Bob has been working as an engineer designing digital and analog electronics. He has published about twenty technical articles, most with *Circuit Cellar*.

hardware. This means becoming familiar with everything from where the name of the diagram is located to how the states of the components shown within the diagrams are represented.

One of the most efficient ways of learning how to learn to read and/or create a hardware diagram is via the *Traister and Lisk* method^[1,3], which involves:

Step 1. Learning the basic symbols that can make up the type of diagram, such as timing or schematic symbols. To aid in the learning of these symbols, rotate between this step and steps 2 and/or 3.

Step 2. Reading as many diagrams as possible until reading them becomes boring (in that case, rotate between this step and steps 1 and/or 3) or comfortable (so there is no longer the need to look up every other symbol while reading).

Step 3. Writing a diagram to practice simulating what has been read, again until it becomes either boring (which means rotating back through steps 1 and/or 2) or comfortable.

1.2 The Embedded Board and the von Neumann Model

In embedded devices, all the electronics hardware resides on a board, also referred to as a *printed wiring board (PW)* or *printed circuit board (PCB)*. PCBs are often made of thin sheets of fiberglass. The electrical path of the circuit is printed in copper, which carries the electrical signals between the various components connected on the board. All electronic components that make up the circuit are connected to this board, either by soldering, plugging into a socket, or some other connection mechanism. All the hardware on an embedded board is located in the hardware layer of the Embedded Systems Model (see Figure 1.3).

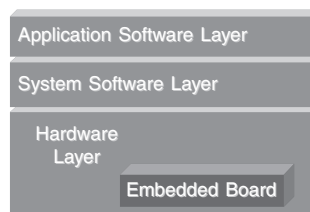


Figure 1.3: Embedded board and the Embedded Systems Model.

At the highest level, the major hardware components of most boards can be classified into five major categories:

- *Central processing unit (CPU)*. The master processor.
- *Memory*. Where the system's software is stored.
- *Input device(s)*. Input slave processors and relative electrical components.

- *Output device(s)*. Output slave processors and relative electrical components.
- *Data pathway(s)/bus(es)*. Interconnects the other components, providing a “highway” for data to travel on from one component to another, including any wires, bus bridges, and/or bus controllers.

These five categories are based on the major elements defined by the *von Neumann model* (see Figure 1.4), a tool that can be used to understand any electronic device’s hardware architecture. The von Neumann model is a result of the published work of John von Neumann in 1945, which defined the requirements of a general-purpose electronic computer. Because embedded systems are a type of computer system, this model can be applied as a means of understanding embedded systems hardware.

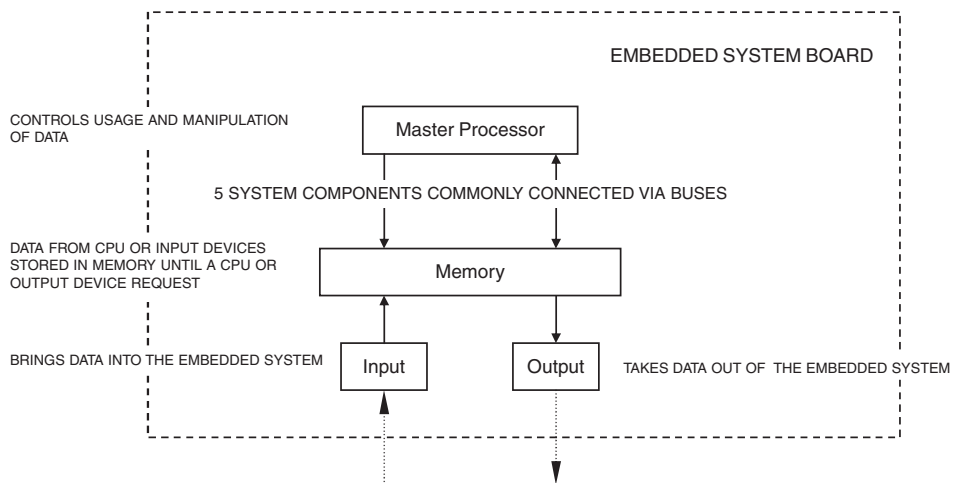


Figure 1.4: Embedded system board organization.^[1.4]

Based on the von Neumann architecture model (also referred to as the Princeton architecture).

While board designs can vary widely, as demonstrated in the examples of Figures 1.5a–d, all the major elements on these embedded boards—and on just about any embedded board—can be classified as either the master CPU(s), memory, input/output, or bus components.

To understand how the major components on an embedded board function, it is useful to first understand what these components consist of and why. All the components on an embedded board, including the major components introduced in the von Neumann model, are made up of one or some combination of interconnected *basic electronic devices*, such as wires, resistors,

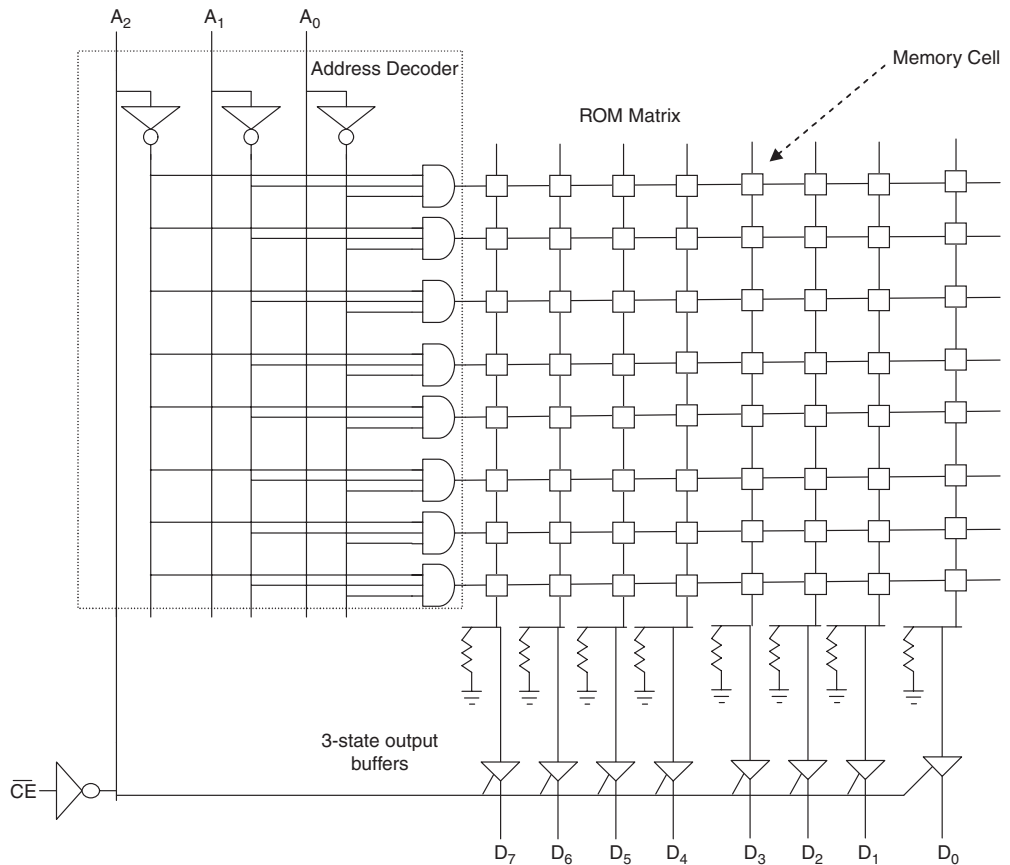


Figure 3.6: Block diagram of memory array.^[3.7]

the bits (or bytes) that make up the data are stored, or *byte ordering*. The two byte-ordering approaches are big-endian, in which the most significant byte or bit is stored first, and little-endian, in which the least significant bit or byte is stored first.

For example:

- 68000 and SPARC are big-endian
- x86 is little-endian
- ARM, MIPS and PowerPC can be configured as either big-endian or little-endian using a bit in their machine state registers

B. *Register set*. A register is simply fast programmable memory normally used to store operands that are immediately or frequently used. A processor's set of registers is commonly referred to as the *register set* or the *register file*. Different processors have

different register sets, and the number of registers in their sets varies between very few to several hundred (even over a thousand). For example, the SA-1110 register set has 37 32-bit registers, whereas the MPC823, on the other hand, has about a few hundred registers (general purpose, special purpose, floating-point, etc.).

- C. *How registers are used.* An ISA defines which registers can be used for what transactions, such as special purpose or floating point, and which can be used by the programmer in a general fashion (general-purpose registers).

As a final note on registers, one of many ways processors can be referenced is according to the size (in bits) of *data* that can be processed and the *size* (in bits) of the *memory space* that can be addressed in a single instruction by that processor. This specifically relates back to the basic building block of registers, the flip-flop; we'll discuss this concept in more detail in Section 3.3.

Commonly used embedded processors support 4-bit, 8-bit, 16-bit, 32-bit, and/or 64-bit processing, as shown in Table 3.2. Some processors can process larger amounts of data and can access larger memory spaces in a single instruction, such as 128-bit architectures, but they are not commonly used in embedded designs.

Table 3.2: “X”-bit architecture examples.

“X”-Bit	Architecture
4	Intel 4004, ...
8	Mitsubishi M37273, 8051, 68HC08, Intel 8008/8080/8086, ...
16	ST ST10, TI MSP430, Intel 8086/286, ...
32	68K, PowerPC, ARM, x86 (386+), MIPS32, ...

3.2.4 Addressing Modes

Addressing modes define the way the processor can access operand storage. In fact, the use of registers is partly determined by the ISA’s *Memory Addressing Modes*. The two most common types of addressing mode models are:

- *Load-store architecture*, which only allows operations to process data in registers, not anywhere else in memory. For example, the PowerPC architecture has only one addressing mode for load and store instructions: register plus displacement (supporting register indirect with immediate index, register indirect with index, etc.).
- *Register-memory architecture*, which allows operations to be processed within both registers and other types of memory. Intel’s i960 Jx processor is an example of an addressing mode architecture that is based on the register-memory model (supporting absolute, register indirect, etc.).

Data is usually stored in cache in one of three schemes:

- *Direct mapped*, where data in cache is located by its associated block address in memory (using the “tag” portion of the block)
- *Set associative*, where cache is divided into *sets* into which multiple blocks can be placed; blocks are located according to an index field that maps into a cache’s particular set
- *Full associative*, where blocks are placed anywhere in cache and must be located by searching the entire cache every time

In systems with memory management units (MMU) to perform the translation of addresses (see Section 3.3), cache can be integrated between the CPU and the MMU or between the MMU and main memory. There are advantages and disadvantages to both methods of cache integration with an MMU, mostly surrounding the handling of DMA (direct memory access), which is the direct access of off-chip main memory by slave processors on the board without going through the main processor. When cache is integrated between the CPU and MMU, only the CPU accesses to memory affect cache; therefore DMA writes to memory can make cache inconsistent with main memory unless CPU access to memory is restricted while DMA data is being transferred or cache is being kept updated by other units within the system besides the CPU. When cache is integrated between the MMU and main memory, more address translations need to be done, since cache is affected by both the CPU and DMA devices.

3.3.2.4 On-Chip Memory Management

Many different types of memory can be integrated into a system, and there are also differences in the way software running on the CPU views memory addresses (*logical/virtual addresses*) and the actual *physical memory addresses* (the two-dimensional array or row and column). *Memory managers* are ICs designed to manage these issues and in some cases are integrated onto the master processor.

The two most common types of memory managers that are integrated into the master processor are memory controllers (MEMC) and memory management units (MMUs). A *memory controller* (MEMC) is used to implement and provide glueless interfaces to the different types of memory in the system, such as cache, SRAM, and DRAM, synchronizing access to memory and verifying the integrity of the data being transferred. Memory controllers access memory directly with the memory’s own physical (two-dimensional) addresses. The controller manages the request from the master processor and accesses the appropriate banks, awaiting feedback and returning that feedback to the master processor. In some cases, where the memory controller is mainly managing one type of memory, it may be referred to by that memory’s name (such as DRAM controller, cache controller, and so forth).

Memory management units (MMUs) are used to translate logical addresses into physical addresses (*memory mapping*) as well as handle memory security, control cache, handle bus arbitration between the CPU and memory, and generate appropriate exceptions. Figure 3.47 shows the MPC860, which has both an integrated MMU (in the core) and an integrated memory controller (in the system interface unit).

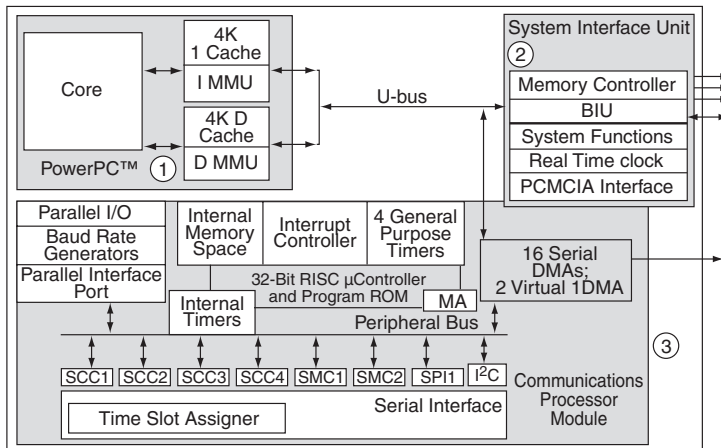


Figure 3.47: Memory management and the MPC860.^[3.38]
Copyright of Freescale Semiconductor, Inc., 2004. Used by permission.

In the case of translated addresses, the MMU can use level-1 cache on the processor, or portions of cache allocated as buffers for caching address translations, commonly referred to as the *translation lookaside buffer*, or TLB, to store the mappings of logical addresses to physical addresses. MMUs also must support the various schemes in translating addresses, mainly segmentation, paging, or some combination of both schemes. In general, *segmentation* is the division of logical memory into large variable-size sections, whereas *paging* is the dividing of logical memory into smaller fixed-size units.

The memory protection schemes then provide shared, read/write, or read-only accessibility to the various pages and/or segments. If a memory access is not defined or allowed, an interrupt is typically triggered. An interrupt is also triggered if a page or segment isn't accessible during address translation (i.e., in the case of a paging scheme, a page fault, etc.). At that point the interrupt would need to be handled; the page or segment would need to be retrieved from secondary memory, for example.

The scheme supporting segmentation and/or paging of the MMU typically depends on the software—that is, the operating system.

Embedded Board Buses and I/O

Tammy Noergaard

4.1 Board I/O

Input/output (I/O) components on a board are responsible for moving information into and out of the board to I/O devices connected to an embedded system. Board I/O can consist of input components, which only bring information from an input device to the master processor; output components, which take information out of the master processor to an output device; or components that do both (see Figure 4.1).

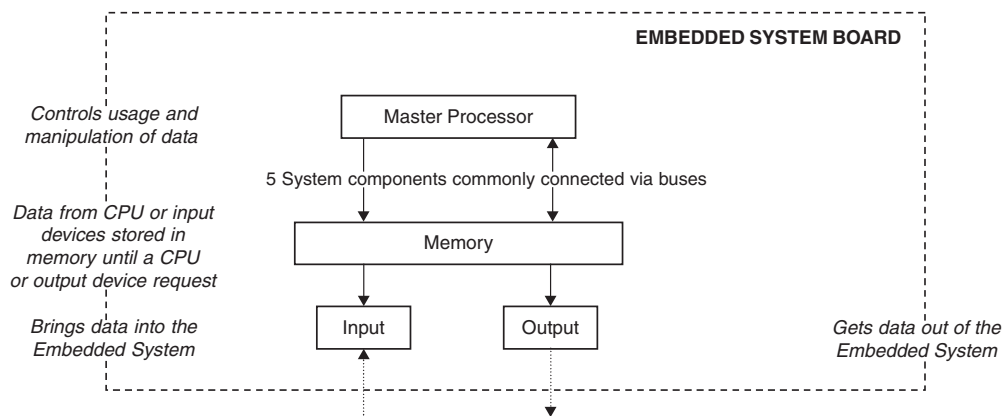


Figure 4.1: Von Neumann-based I/O block diagram.

Any electromechanical system, both embedded and nonembedded and whether conventional or unconventional, can be connected to an embedded board and act as an I/O device. I/O is a high-level group that can be subdivided into smaller groups of output devices, input devices, and devices that are both input and output devices. Output devices receive data from board I/O components and display that data in some manner, such as printing it to paper, to a disk, or to a screen or a blinking LED light for a person to see. An input device such as a mouse,

keyboard, or remote control transmits data to board I/O components. Some I/O devices can do both, such as a networking device that can transmit data to and from the Internet, for instance. An I/O device can be connected to an embedded board via a wired or wireless data transmission medium such as a keyboard or remote control or can be located on the embedded board itself, such as an LED.

Because I/O devices are so varied, ranging from simple circuits to other complete embedded systems, board I/O components can fall under one or more of several different categories, the most common including:

- Networking and communications I/O (the physical layer of the OSI model)
- Input (keyboard, mouse, remote control, vocal, etc.)
- Graphics and output I/O (touch screen, CRT, printers, LEDs, etc.)
- Storage I/O (optical disk controllers, magnetic disk controllers, magnetic tape controllers, etc.)
- Debugging I/O (BDM, JTAG, serial port, parallel port, etc.)
- Real-time and miscellaneous I/O (timers/counters, analog-to-digital converters and digital-to-analog converters, key switches, and so on)

In short, board I/O can be as simple as a basic electronic circuit that connects the master processor directly to an I/O device, such as a master processor's I/O port to a clock or LED located on the board, to more complex I/O subsystem circuitry that includes several units, as shown in Figure 4.2. I/O hardware is typically made up of all or some combination of six main logical units:

- The *transmission medium*, a wireless or wired medium connecting the I/O device to the embedded board for data communication and exchanges
- A *communication port*, to which the transmission medium connects on the board or, if a wireless system, which receives the wireless signal
- A *communication interface*, which manages data communication between master CPU and I/O device or I/O controller and is responsible for encoding data and decoding data to and from the logical level of an IC and the logical level of the I/O port; this interface can be integrated into the master processor or can be a separate IC
- An *I/O controller*, a slave processor that manages the I/O device
- *I/O buses*, the connection between the board I/O and master processor
- The *master processor integrated I/O*

- *STROBE** (nSTROBE), *AUTOFD** (nAUTOFEED), *INIT** (nINIT), *HSELECT** (nSELECTIN), **ACK* (nACK), *BUSY*, *PE*, *PSELECT* (SELECT), **FAULT* (nERR-ROR), ...

4.2.6 Parallel and Serial I/O Example 4: Networking and Communications—Ethernet

One of the most widely implemented LAN protocols is Ethernet, which is primarily based on the IEEE 802.3 family of standards. These standards define the major components of any Ethernet system. Thus, to fully understand an Ethernet system design, you first need to understand the IEEE specifications. (Remember, this is not a book about Ethernet, and there is a lot more involved than is covered here. This example is about understanding a networking protocol and then being able to understand the design of a system based on a networking protocol such as Ethernet.)

The first step is understanding the main components of an Ethernet system, regardless of whether these components are implemented in hardware or software. This is important since different embedded architectures and boards implement Ethernet components differently. On most platforms, however, Ethernet is implemented almost entirely in hardware.

The hardware components can all be mapped to the physical layer of the OSI model. The firmware (software) required to enable Ethernet functionality maps to the lower section of the OSI data-link layer but will not be discussed in this section.

Several Ethernet system models are described in the IEEE 802.3 specification, so let's look at a few to get a clear understanding of some of the most common Ethernet hardware components.

Ethernet devices are connected to a network via *Ethernet cables*: thick coax (coaxial), thin coax, twisted-pair, or fiber optic cables. These cables are commonly referred to by their IEEE names. These names are made up of three components: the data transmission rate, the type of signaling used, and either the cable type or cable length.

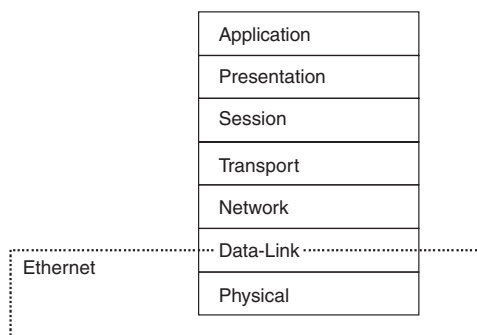


Figure 4.17: OSI model.

mark data that has changed in cache under write-back mode. For example, if we read a value from memory that ends up in cache and the value is subsequently modified by the processor, it is marked as “dirty.” The cache uses this bit as a reminder that before the data is completely removed from cache, it needs to be written out to its source memory. Processors often have a *victim buffer* that holds data that was replaced in the cache. Let’s consider why and how this helps performance.

When a data miss occurs in write-back mode and the entire cache contains valid data, something must be removed from the cache to hold the data about to be fetched. Recall that when this happens, the cache (assuming an LRU policy) will replace the data least recently used. What if there’s “dirty” data in the line that is replaced—that is, data which has changed and needs to be updated in the source memory? The processor is most immediately interested in obtaining data for the currently executing instruction. If it had to wait for the dirty data to be written back to source memory, and then wait again for the new data to be fetched, the core would stall longer than desired. This is where the victim buffer comes in; it holds the data that needs to be written back, while the core gets its new data quickly. Once the cache-line fill completes, the victim buffer empties and the source memory is current.

5.4 External Memory

So far in this chapter, our discussions have centered on internal memory resources. Let’s now focus on the many storage options available external to the processor. We generically refer to external memory as “L3” throughout this text, but you will soon see that the choices for L3 memory vary considerably in terms of the way they operate and their primary uses. They all can play an important role in media-based applications. We will start with the highest-performance volatile memories and move to various nonvolatile options. It is important to note here that the synchronous and asynchronous memories described here are directly memory-mapped to a processor’s memory space. Some of the other memories we’ll discuss later in this chapter, such as NAND flash, are also mapped to an external memory bank, but they have to be indirectly accessed.

5.4.1 Synchronous Memory

We begin our discussion with synchronous memory because it is the highest-performance external memory. It’s widely available and provides a very cost-effective way to add large amounts of memory without completely sacrificing performance. We focus on SDRAM technology to provide a good foundation for understanding, but then we’ll proceed to an overview of current and near-term follow-ons to SDRAM: DDR-SDRAM 1 and 2.

Both SDRAM and DDR are widely available and very cost-effective because the personal computer industry uses this type of memory in standard DIMM modules.

5.4.1.1 SDRAM

SDRAM is synchronous addressable memory composed of banks, rows, and columns. All reads and writes to this memory are locked to a processor-sourced clock. Once the processor initializes SDRAM, the memory must be continually refreshed to ensure that it retains its state.

Clock rates vary for SDRAM, but the most popular industry specifications are PC100 and PC133, indicating a 100MHz and 133MHz maximum clock, respectively. Today's standard SDRAM devices operate at voltages between 2.5 V and 3.3 V, with the PC133 memory available at 3.3 V.

Memory modules are specified in terms of some number of Mbits in addition to a variety of data bus sizes (e.g., $\times 8$, $\times 16$, $\times 32$). This sometimes causes confusion, but it is actually a straightforward nomenclature. For example, the $\times 16$ designation on a 256-Mbit device implies an arrangement of $16\text{Mbits} \times 16$. Likewise, for a $\times 32$ bus width, a 256-Mbit device would be configured as $8\text{Mbits} \times 32$. This is important when you are selecting your final memory size. For example, if the external bus on your processor is 32 bits wide and you want 128 Mbytes of total memory, you might connect two $32\text{Mbits} \times 16$ modules.

At its lowest level, SDRAM is divided into rows and columns. To access an element in an SDRAM chip, the row must first be "opened" to become "active." Next, a column within that row is selected, and the data is transferred from or written to the referenced location. The process of setting up the SDRAM can take several processor system clock cycles. Every access requires that the corresponding row be active.

Once a row is active, it is possible to read data from an entire row without re-opening that row on every access. The address within the SDRAM will automatically increment once a location is accessed. Because the memory uses a high-speed synchronous clock, the fastest transfers occur when performing accesses to contiguous locations, in a burst fashion.

Figure 5.11 shows a typical SDRAM controller (SDC) with the required external pins to interface properly to a memory device. The data access size might be 8, 16, or 32 bits. In addition, the actual addressable memory range may vary, but an SDC can often address hundreds of Mbytes or more.

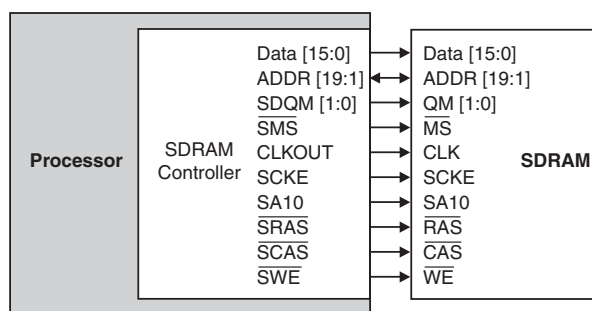


Figure 5.11: Representative SDRAM controller.

SDRAMs are composed of internal banks—most commonly, four equally sized banks. So if you connected 64 Mbytes of SDRAM to a processor, the SDRAM would consist of four 16-Mbyte internal banks. This is important to remember because you'll derive performance benefits from partitioning your application across the internal banks. Another thing to note—the term *bank* is unfortunately used to describe both the internal structure of an SDRAM and an entire SDRAM module (as viewed from the system level). For example, two 64-Mbyte *external* SDRAM banks may connect to a processor, and each 64-Mbyte module may consist of four 16-Mbyte *internal* banks each.

The SDRAM controller uses the external pins shown in Figure 5.11 to issue a series of commands to the SDRAM. Table 5.1 provides a brief description of each of the pins, and Table 5.2 shows how the pins work in concert to send commands to the SDRAM. It generates these commands automatically based on writes and reads by the processor or DMA controller.

Table 5.1: SDRAM pin description.

ADDR	External address bus
DATA	External data bus
$\overline{\text{SRAS}}$	SDRAM row address strobe (connect to SDRAM's $\overline{\text{RAS}}$ pin)
$\overline{\text{SCAS}}$	SDRAM column address strobe (connect to SDRAM's $\overline{\text{CAS}}$ pin)
$\overline{\text{SWE}}$	SDRAM write-enable pin (connect to SDRAM's $\overline{\text{WE}}$ pin)
SDQM	SDRAM data mask pins (connect to SDRAM's DQM pins)
$\overline{\text{SMS}}$	Memory select pin of external memory bank configured for SDRAM
SA10	SDRAM A10 pin (used for SDRAM refreshes; connect to SDRAM's A[10] pin)
SCKE	SDRAM clock-enable pin (connect to SDRAM's CKE pin)
CLKOUT	SDRAM clock pin (connect to SDRAM's CLK pin; Operates at SCLK frequency)

Table 5.2: SDRAM commands.

Command	$\overline{\text{SMS}}$	$\overline{\text{SCAS}}$	$\overline{\text{SRAS}}$	$\overline{\text{SWE}}$	SCKE	SA10
Precharge All	low	high	low	low	high	high
Single Precharge	low	high	low	low	high	low
Bank Activate	low	high	low	high	high	–
Load Mode Register	low	low	low	low	high	–
Load Extended Mode Register	low	low	low	low	high	low
Read	low	low	high	high	high	low
Write	low	low	high	low	high	low
Auto-Refresh	low	low	low	high	high	–
Self-Refresh	low	low	low	high	low	–
NOP (No Operation)	low	high	high	high	high	–
Command Inhibit	high	high	high	high	high	–

Figure 5.12 illustrates an SDRAM transaction in a simplified manner. First, the higher bits of the address are placed on the bus and $\overline{\text{RAS}}$ is asserted. The lower address bits of the address are then placed on the bus and $\overline{\text{CAS}}$ is asserted. The number of rows and columns will depend on the device you select for connection.

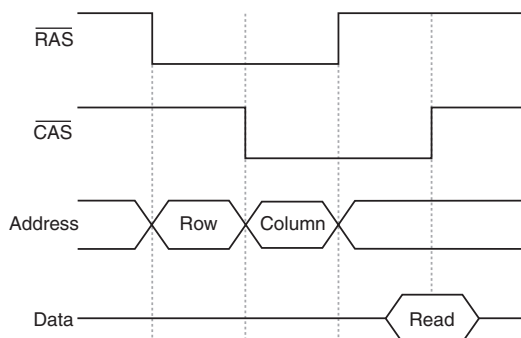


Figure 5.12: Basic SDRAM timing diagram.

It is important to consider the way your processor's SDC multiplexes SDRAM addresses. Consider two possibilities shown in Figure 5.13 as an example. In Figure 5.13b, the SDRAM row addresses are in the higher bit positions. In Figure 5.13a, the bank address lines are in the higher bit positions, which can result in better performance, depending on the application. Why is this the case? The SDRAM can keep four pages open across four internal banks, thus reducing page opening/closing latency penalties. If your system is connected as shown in Figure 5.13b, it would be very hard to partition your code and data to take advantage of this

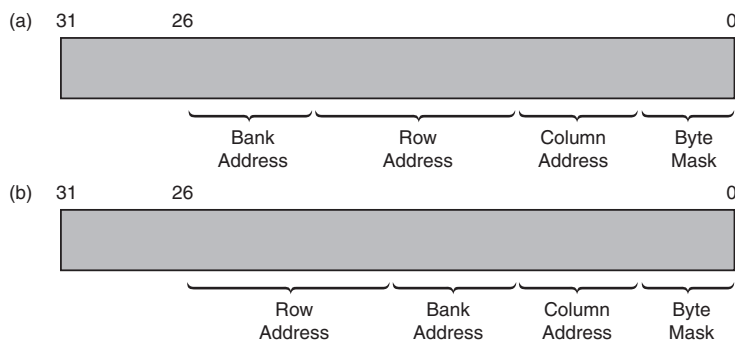


Figure 5.13: Possible SDRAM address muxing.

feature. Specifically, you would have to slice up your data and code and essentially interleave it in memory to make use of all open rows.

Let's now discuss briefly some of the key commands that the SDRAM controller uses to interface with the memory device.

The *Bank Activate* command causes the SDRAM to open an internal bank (specified by the bank address) in a row (specified by the row address). The pins that are used for the bank and row addresses depend on the mappings of Figure 5.13. When the SDC issues the *Bank Activate* command, the SDRAM opens a new row address in the dedicated bank. The memory in the open internal bank and row is referred to as the *open page*. The *Bank Activate* command must be applied before issuing a read or write command.

The *Precharge* command closes a specific internal bank in the active page or all internal banks in the page.

The *Precharge All* command precharges all internal banks at the same time before executing an auto-refresh.

A *Read/Write* command executes if the next read/write access is in the present active page. During the *Read* command, the SDRAM controller drives the column address. The delay between *Bank Activate* and *Read* commands is determined by the t_{RCD} parameter in the SDRAM data sheet. The SDRAM then makes data available after the CAS latency period described below.

In the *Write* command, the SDRAM controller drives the column address. The write data becomes valid in the same cycle. The delay between *Bank Activate* and *Write* commands is determined by the t_{RCD} parameter in the SDRAM data sheet.

Whenever a page miss occurs (an access to a location on a row that is not open), the SDC executes a *Precharge* command followed by a *Bank Activate* command before executing the *Read* or *Write* command. If there is a page hit, the *Read* or *Write* command can be issued immediately without requiring the *Precharge* command.

The *Command Inhibit* and *NOP* commands are similar in that they have no effect on the current operations of the SDRAM. The only difference between the two is that the *NOP* is used when the actual SDRAM bank has been selected.

The *Load Mode* command is used to initialize an SDRAM chip. *Load Extended Mode* is an additional initialization command that's used for mobile SDRAMs.

Auto-Refresh and *Self-Refresh* commands regulate the way the contents of the SDRAM are refreshed periodically. We'll talk more about them shortly.

5.4.1.2 CAS Latency

SDRAM data sheets are swimming in numbers that specify the performance of the device. What do all these parameters really mean?

The *Column Address Strobe (CAS) latency*, abbreviated as CL2 or CL3, is the delay in clock cycles between when the SDRAM detects a *Read* command and when it provides the data at its output pins. This CAS latency is an important selection parameter. A common term used to identify SDRAM devices is either CAS2 or CAS3. These actually represent CL2 or CL3, since they refer to *CAS latency* timings (e.g., two system clocks versus three system clocks). An SDRAM with a CAS latency of two cycles will likely yield better throughput than one with a three-cycle latency. This is based on the fact that for random accesses, a cycle will be saved each time an access is made. You should specify this parameter based on application needs. Does the extra performance of the faster device justify the extra cost? For high-performance systems, the answer is usually “Yes.”

The CAS latency of a device must be greater than or equal to its column access time (t_{CAC}) and its frequency of operation (t_{CLK}). That is, the selection of CL must satisfy this equation:

$$\text{CL} \times t_{\text{CLK}} \geq t_{\text{CAC}}$$

For example, if t_{CLK} is 7.5 ns (133 MHz system clock) and t_{CAC} is 15 ns, you can select a CL2 device. If t_{CAC} is 20 ns, you must choose CL3. The PC133 SDRAM specification only allows for CAS latency values of 1, 2, or 3.

Sometimes you will see memory devices described as 3-2-2 or 2-2-2. These numbers represent the CAS latency, RAS-to-CAS delay (t_{RCD}), and Row Precharge (t_{RP}) values, respectively, in clock cycles at 100 MHz. Note that for any other speed of operation, such as 66 MHz or 133 MHz, these numbers would change. For example, let's assume that for a given module, t_{CAC} is 25 ns, t_{RCD} is 20 ns, and t_{RP} is 20 ns. This would indicate 3-2-2 timings at 100 MHz (substituting t_{RCD} or t_{RP} for t_{CAC} in the preceding equation as appropriate), but what would they be at 133 MHz? Since 133 MHz corresponds to a 7.5 ns clock cycle (t_{CLK}), our equation gives timings of 4-3-3, which would be invalid for the SDRAM, since the CAS latency cannot be higher than 3. Therefore, you would not be able to operate this module at 133 MHz.

One more point to understand about the CAS latency figure (CAS2 or CAS3) is that SDRAM suppliers often advertise their top PC100 SDRAM as CAS2. Recall that the CAS latency number is derived from t_{CAC} . Unfortunately, the vendor doesn't provide you with the t_{CAC} value. Imagine a CL2 part with a 20-ns t_{CAC} and a CL3 part with a 21-ns t_{CAC} . At 133 MHz (7.5 ns clock period), both parts would have a CAS latency value of 3. This means that although a CL2 part may be designed to handle a faster system clock speed than a CL3 part, you won't always see a performance difference. Specifically, if you don't plan on running your SDRAM faster than 125 MHz, the device with the lower value of CL doesn't provide any additional benefit over the device with the higher CL value.

5.4.1.3 Refreshing the SDRAM

SDRAM controllers have a refresh counter that determines when to issue refresh commands to the SDRAM. When the SDC refresh counter times out, the SDC precharges all banks of SDRAM and then issues an *Auto-Refresh* command to them. This causes the SDRAM to generate an internal refresh cycle. When the internal refresh completes, all internal SDRAM banks are precharged.

In power-saving modes where SDRAM data needs to be maintained, it is sometimes necessary to place the SDRAM in self-refresh mode. Self-refresh is also useful when you want to buffer the SDRAM while the changes are made to the SDRAM controller configuration. Moreover, the self-refresh mode is useful when sharing the memory bus with another resource. It can prevent conflicts on shared pins until the first processor regains ownership of the bus.

When you place the SDRAM in self-refresh mode, it is the SDRAM's internal timer that initiates auto-refresh cycles periodically, without external control input. Current draw when the SDRAM is in self-refresh mode is on the order of a few milliamps, versus the typical "on" current of 100 mA.

The SDC must issue a series of commands, including the *Self-Refresh* command, to put the SDRAM into this low power mode, and it must issue another series of commands to exit self-refresh mode. Entering self-refresh mode is controlled by software in an application. Any access made by the processor or the DMA controller to the SDRAM address space causes the SDC to remove the SDRAM from self-refresh mode.

It is important to be aware that core or DMA accesses to SDRAM are held off until an in-process refresh cycle completes. This is significant because if the refresh rate is too high, the potential number of accesses to SDRAM decreases, which means that SDRAM throughput declines as well. Programming the refresh rate to a higher value than necessary is a common mistake that programmers make, especially on processors that allow frequency modification on the fly. In other words, they forget to adjust the Refresh Control register to a level commensurate with the newly programmed system clock frequency. As long as the effective refresh rate is not too slow, data will not be lost, but performance will suffer if the rate is too high.

5.4.1.4 Mobile SDRAM

A variant of SDRAM that targets the portable device realm is called *mobile SDRAM*. It comes in a smaller form factor and smaller memory sizes than its traditional cousin, and it can operate down to 2.5 V or 1.8 V, greatly reducing SDRAM power consumption. Mobile SDRAM is also known as LP SDRAM, or low-power SDRAM. It is worth mentioning that mobile SDRAM devices typically specify a supply voltage and an I/O voltage. For the most power-sensitive applications, the supply voltage is at 2.5 V while the I/O supply is 1.8 V.

In addition to a reduced form factor and greatly reduced power budget, mobile SDRAM has three key JEDEC-specified features that set it apart from SDRAM. The first is a temperature

compensated self-refresh mode. The second is a partial array self-refresh capability, and the third is a “deep power-down” mode.

The temperature-compensated self-refresh capability allows you to program the mobile SDRAM device to automatically adjust its refresh rate in self-refresh mode to accommodate case temperature changes. Some mobile devices assume you will connect a temperature sensor to the case and adjust the parameters associated with this feature. Others have a built-in temperature sensor. Either way, the goal is to save power by adjusting the frequency of self-refresh to the minimum level necessary to retain the data.

The partial array self-refresh feature allows you to control which banks within a mobile SDRAM are actually refreshed when the device is in self-refresh mode. You can program the device so that 100%, 50%, or 25% of the banks are kept in self-refresh mode. Obviously, any data you need to keep must reside in a bank that is self-refreshed. The other banks are then used for data that does not need to be retained during power-saving modes.

Finally, the deep power-down feature allows you to remove power from the device via a write to the control register of the SDRAM chip to prevent any current draw during self-refresh mode (thus saving hundreds of μA). Of course, all data is lost when you do this, but this mode can be very useful in applications in which the entire board is not powered down (e.g., some components are operational and are powered by the same regulator) but you want to extend the battery life as long as possible.

5.4.1.5 Double Data Rate (DDR) SDRAM/DDR1

SDRAM and mobile SDRAM chips provide the bulk of today’s synchronous memory in production. This is quickly changing, however, as an evolved synchronous memory architecture provides increased performance. Double data rate (DDR) SDRAM provides a direct path to double memory bandwidth in an application. In addition, although the industry has more or less skipped 2.5 V SDRAM devices (with the notable exception of mobile SDRAM), standard DDR1 chips are all at 2.5 V.

The traditional SDRAM controller often shares processor pins with various types of asynchronous memory, but the DDR specification is much tighter to allow for much faster operation. As such, a DDR memory module will not share pins with other types of memory. Additionally, DDR memories require a DDR controller to interface with them; a processor’s SDRAM controller is incompatible with DDR memory.

As the name suggests, a key difference between SDRAM and DDR is that whereas SDRAM allows synchronous data access on each clock cycle, DDR allows synchronous access on both edges of the clock—hence the term *double data rate*. This results in an immediate increase in peak performance when you use DDR in an application.

As an example, whereas the peak transfer rate of a PC133 SDRAM running at 133 MHz would be $4 \text{ bytes} \times 133 \text{ MHz}$, the maximum peak transfer rate of the equivalent DDR1 module running at the same frequency is $4 \text{ bytes} \times 266 \text{ MHz}$.

Another important advantage DDR has over SDRAM is the size of the “prefetch” it accepts. DDR1 has a prefetch size of $2n$, which means that data accesses occur in pairs. When reads are performed, each access results in a fetch of two data words. The same is true for a single write access—that is, two data words must be sent.

This means that the minimum burst size of DDR1 is two external data words. For reads, the DDR controller can choose to ignore either of the words, but the cycles associated with both will still be spent. So you can see that although this feature greatly enhances performance of sequential accesses, it erodes performance on random accesses.

DDR SDRAM also includes a strobe-based data bus. The device that drives the data signals also drives the data strobes. This device is the DDR module for reads and the DDR controller for writes. These strobes allow for higher data rates by eliminating the synchronous clock model to which SDRAM adheres.

When all these DDR feature enhancements are combined, performance really shines. For example, at 400 MHz, DDR1 increases memory bandwidth to 3.2 Gbytes/s, compared to PC133 SDRAM theoretical bandwidth of $133 \text{ MHz} \times 4 \text{ bytes}$, or 532 Mbytes/s.

Just as a low-power version of SDRAM exists, mobile DDR1 also exists, with many similarities to mobile SDRAM. Mobile DDR1 devices run as low as 1.8 V. Another advantage of mobile DDR1 is that there is no minimum clock rate of operation. This compares favorably to standard DDR1, whose approximate 80 MHz minimum operating frequency can cause problems in power-sensitive applications.

5.4.1.6 DDR2 SDRAM

DDR2 SDRAM is the second generation of DDR SDRAM. It offers data rates of up to 6.4 Gbytes/s, lower power consumption, and improvements in packaging. It achieves this higher level of performance and lower power consumption through faster clocks, 1.8 V operation and signaling, and a simplified command set. Like DDR1, DDR2 has a mobile variety as well, targeted for handheld devices.

Table 5.3 shows a summary of the differences between DDR1, mobile DDR1, and DDR2. It should be noted that DDR1 is not compatible with a conventional SDRAM interface, and DDR2 is not compatible with DDR1. However, DDR2 is planned to be forward-compatible with next-generation DDR technology.

5.4.2 Asynchronous Memory

As we have just seen, SDRAM and its successors both are accessed in a synchronous manner. Asynchronous memory, as you can guess, does not operate synchronously to a clock. Each

Table 5.3: Comparison of DDR1, mobile DDR1, and DDR2.

Feature	DDR1	Mobile DDR1	DDR2
Data transfer rate	266, 333, 400 MHz	200, 250, 266, 333 MHz	400, 533, 667, 800 MHz
Operating voltage	2.5V	1.8V	1.8V
Densities	128 Mb–1 Gb	128–512 Mb, 1 Gb(future)	256 Mb–4 Gb
Internal banks	4	4	4 and 8
Prefetch	2	2	4
CAS latency	2, 2.5, 3	2, 3	3, 4, 5, 6
Additive latency	No	No	0, 1, 2, 3, 4, 5
READ latency	CAS latency	CAS latency	Additive latency + CAS latency
WRITE latency	Fixed = 1 cycle	Fixed = 1 cycle	READ latency – 1 cycle
I/O width	×4, ×8, ×16	×4, ×8, ×16, ×32	×4, ×8, ×16
On-die termination	No	No	Selectable
Off-chip driver	No	No	Yes
Burst length	2, 4, 8	2, 4, 8, 16, full page	4, 8
Burst terminate command	Yes	Yes	No
Partial array self-refresh	No	Full, 1/2, 1/4, 1/8, 1/16	No
Temperature-compensated self-refresh	No	Supported	No
Deep power-down	No	Supported	No

access has an associated read and write latency. Burst operations are always available with synchronous memory, but the same can't be said of asynchronous memory.

The asynchronous bus of today has evolved from buses that were popular in the past, such as IBM's *Industry Standard Architecture* (ISA) bus. Asynchronous devices come in many different flavors. The most common include Static RAM (SRAM), which is volatile, and nonvolatile memories like PROM and Flash. SRAM can substitute for the pricier SDRAM when high performance isn't required.

Processors that have an asynchronous memory controller (AMC) typically share its data and address pins with the SDRAM controller. As we mentioned earlier, because DDR has much tighter capacitive loading requirements, an AMC would not share pins with a DDR controller.

A characteristic AMC is shown in Figure 5.14. It contains read- and write-enable pins that can interface to memory devices. Table 5.4 shows a summary of the common AMC pins. Several

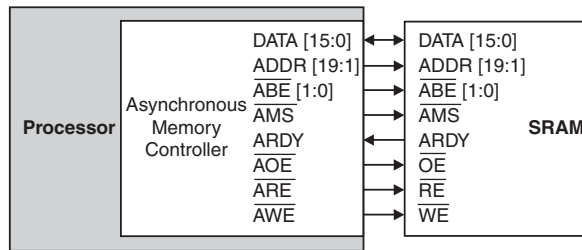


Figure 5.14: A typical asynchronous memory controller (AMC).

Table 5.4: Typical AMC pins.

ADDR	External address bus (outputs)
DATA	External data bus (inputs/outputs)
AMS	Asynchronous memory selects (outputs)
AWE	Asynchronous memory write enable (output)
ARE	Asynchronous memory read enable (output)
AOE	Asynchronous memory read enable (output)
ARDY	Asynchronous memory ready response (input)
ABE[1:0]	Byte enables (outputs)

memory-select lines allow the AMC to choose which of several connected devices it's talking to at a given point in time. The AMC has programmable wait states and setup/hold timing for each connected device, to accommodate a wide array of asynchronous memories.

The AMC is especially useful because, in addition to interfacing with memory devices, it allows connection to many other types of components. For example, FIFOs and FPGAs easily map to an asynchronous bank. Chip sets for USB and Ethernet, as well as bridges to many other popular peripherals, also easily interface to the asynchronous memory interface.

When connecting a nonmemory device to an AMC, it is always best to use a DMA channel to move data into and out of the device, especially when the interface is shared with SDRAM. Access to these types of components usually consumes multiple system clock cycles, whereas an SDRAM access can occur in a single system clock cycle. Be aware that, when the AMC and SDC share the same L3 memory bus, slow accesses on an asynchronous bank could hold off access to a SDRAM bank considerably.

Synchronous random access memory (synchronous SRAM) is also available for higher performance than traditional SRAMs provide, at increased cost. Synchronous SRAM devices are capable of either pipelined or flow-through functionality. These devices take the asynchronous devices one step closer to SDRAM by providing a burst capability.

Whereas synchronous SRAM provides for higher performance than ordinary SRAM, other technologies allow for lower power. Pseudo-SRAM (and a variant called CellularRAM) connect to a processor via an SDRAM-like interface. Additionally, they sport an I/O supply requirement in line with other processor I/O (2.5 V or 3.3 V, for instance) while powering the V_{cc} supply of the memory itself at 1.8 V. This presents a good compromise that allows processors to take advantage of some power savings even when they don't have 1.8 V-capable I/O.

Figure 5.15 shows a high-level view comparing several representative types of external memory from the dual standpoints of performance and capacity.

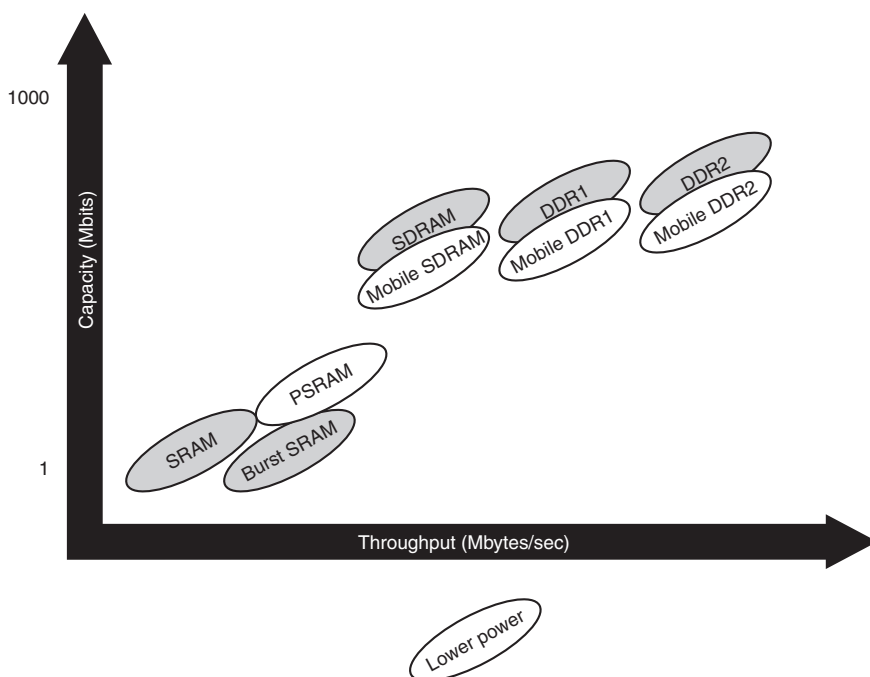


Figure 5.15: External memory comparison of performance and capacity.

5.4.3 Nonvolatile Memories

Nonvolatile memories—memories that retain their contents even when not powered—come in several forms. The simplest, a ROM (read-only memory), is written once (at the factory) but can be read many times. A PROM, or Programmable ROM, also can only be written once, but this can be done in the field, not just at the time of manufacture. An erasable PROM (EPROM) can be reprogrammed after erasing its contents via ultraviolet light exposure. An electrically erasable PROM (EEPROM), commonly used today, needs only electricity for erasure. A Flash

EEPROM takes this technology one step further by allowing data to be stored and erased in blocks, rather than in byte-sized increments. This significantly speeds up access rates to Flash memories compared with regular EEPROM access times. Finally, a burst-mode Flash sharply reduces read access times for sequential bytes by adding a few extra control pins to the standard Flash interface.

5.4.3.1 NAND and NOR Flash Memories

The two main types of nonvolatile Flash memory widely used today are NOR-based and NAND-based. There are many differences between these two technologies, each of which is optimal for certain classes of use. Table 5.5 gives a rundown of their major characteristics.

Table 5.5: Summary of NOR and NAND Flash characteristics.

Trait	NOR Flash	NAND Flash
Capacity/bit density	Low (<64 MB)	High (16–512 MB)
Directly execute code from?	Yes	No
Erase performance	Very slow (5 sec)	Fast (3 ms)
Write performance	Slow	Fast
Read performance	Fast	Medium
Reliability	OK	Low; requires error checking and bad-block checking
Erase cycles	10 K–100 K	100–1000 K
Life span	OK	Excellent (10× NOR)
Interface	ISA-like/MCU-friendly; (Addr + Data + Control), Serial (SPI or I2C)	I/O only (command sequence)
Pin count	High	Low
Access method	Random	Sequential
Ease-of-use	Easy; memory-mapped address + data scheme	Difficult (file system needed)
Cost/bit	High	Low
Primary usage	Low-density, high-speed code access, some data storage	High-density data block storage
Bootable?	Yes	Not generally
Power/Energy Dissipation	Higher (due to long program and erase cycles)	Lower

Fundamentally, NAND and NOR Flash technologies differ in the structure of their respective memory cell arrays. In NOR devices, memory cells are connected in parallel between a bit line and ground, such that selecting any number of memory cells can end up grounding a bit line. Because this is similar to a wired-OR configuration, it is termed *NOR Flash*. This

arrangement enables very fast read times and thus makes NOR Flash a good candidate for random access—a trait associated with processor code.

NAND realizes a more efficient cell architecture that results in only about half the space taken by a NOR cell. However, this space-saving layout connects memory cells in series, sharing bit-select lines across several cells. To ground a bit line, an entire group of memory cells must be turned on simultaneously. This makes NAND Flashes a poor choice for random accesses, but it provides excellent cell compactness. What's more, it allows several NAND cells to be programmed simultaneously, providing very fast write performance compared to NOR devices.

NOR Flash holds the edge in read access speed, but NAND is superior in programming and erasing speeds. Erase time is especially crucial because Flash devices must all be erased (each bit set to 1) before programming occurs (selectively setting bits back to 0). Therefore, erasing is an integral part of writing to a Flash device. Moreover, NAND Flashes can tolerate many more erase cycles than NOR Flashes can, usually on the order of tenfold, thus providing much longer life spans than NOR devices.

One intricacy of NAND Flashes is that a file system is necessary to use them. This is because, to maintain high yields, NAND Flashes are shipped with randomly located bad blocks. It is the job of the file system (running on the embedded processor) to find these bad blocks, tag them, and avoid using them. Usually, the memory vendor has already scanned and marked the bad blocks at the factory, and the file system just needs to maintain a table of where these are located, keeping in mind that some additional bad blocks will be formed over the lifetime of the part. The good news is that because each block is independent of all others, the failure of one block has no impact on the operation of others.

As it turns out, NOR Flash can also have bad blocks, but manufacturers typically allocate extra blocks on the chip to substitute for any bad blocks discovered during factory tests. NAND has no such spare blocks because it's assumed that a file system is necessary anyway for the mass storage systems of which NAND devices are a part.

As another consideration, NAND is somewhat prone to bit errors due to periodic programming (one bit error out of every 10 billion bits programmed). Therefore, error-correcting codes (Hamming codes, usually) are employed by the file system to detect and correct bit errors.

As far as processor interface, NOR devices have a couple of options. For one, they hook directly up to the asynchronous memory controller of microprocessors, with the conventional address and data bus lines as well as write/read/select control. Depending on the address space desired, this interface can encompass a large number of pins. As a slower, lower pin-count alternative, serial Flash devices can connect through just a few wires via an SPI or I²C interface. Here, the address and data are multiplexed on the serial bus to achieve memory access rates only a fraction of those attainable using parallel address/data buses.

Interfacing to NAND devices is more complex and occurs indirectly, requiring a sequence of commands on the 8-bit bus to internal command and address registers. Data is then accessed in pages, usually around 528 bytes in length. Although this indirect interface makes NAND unsuitable for booting, this approach provides a fixed, low pin-count interface that remains static as higher-density devices are substituted. In this respect (and many others), NAND Flash acts like a conventional hard disk drive. This accounts for the similar structure and access characteristics between the two storage technologies.

In fact, NAND Flash devices were specifically intended to replace magnetic hard disk drives in many embedded applications. To their advantage, they are solid-state devices, meaning that they have no moving parts. This leads to more rugged, reliable devices than magnetic disk drives, as well as much less power dissipation.

NAND flash serves as the basis for all of the most popular removable solid-state mass storage cards: CompactFlash, SmartMedia, Secure Digital/Multimedia Card (SD/MMC), Extreme Digital Picture Card (xD), MemoryStick, and the like. With the exception of SmartMedia, all these cards have built-in controllers that simplify access to the NAND memory on the device. These products find wide use in consumer electronics (like cameras, PDAs, cell phones, etc.) and other embedded applications requiring mass storage. Table 5.6 provides a high-level comparison of these storage cards.

Devices like SD, MemoryStick, and CompactFlash also have an I/O layer, which makes them quite attractive as interfaces on embedded processors. They enable a wide range of peripherals through these interfaces, including Bluetooth, 802.11b, Ethernet transceivers, modems, FM radio, and the like.

5.4.3.2 Hard Disk Storage: IDE, ATA, and ATAPI

The terms *IDE* and *ATA* actually refer to the same general interface, but they mean different things, and people often get confused and use the terms interchangeably. For simplicity, we'll just refer to the interface as *ATA* or *ATAPI*, as explained below.

IDE is short for *Integrated Drive Electronics*, which refers to a pivotal point in the PC storage market when the disk drive controllers were integrated directly on the hard disk itself, rather than as a separate board inside the PC. This reduced the cost of the interface and increased reliability considerably. *IDE* drives can support a master and a slave device on each *IDE* channel.

ATA stands for *AT Attachment*, dating back to the days of the IBM PC/AT. It describes a device interface for hooking up to hard disk drives. *ATAPI* stands for *ATA Packet Interface*, which specifies a way to hook up to CD-ROM, DVD and tape drives, which communicate by means of packetized commands and data. Although the packet layer of *ATAPI* adds considerable complexity, it still maintains the same electrical interface as *ATA*. *ATA* is such a ubiquitous standard for

Table 5.6: Storage card comparison.

	SmartMedia	MMC	CompactFlash	Secure Digital (SD)	MemoryStick	xD	TransFlash/ Micro SD
Developer	Toshiba	Infineon, SanDisk	SanDisk	SanDisk, Panasonic, Toshiba	Sony (later with SanDisk)	Fujifilm, Olympus	SanDisk, Motorola
Variants		MMCplus, HS-MMC (High Speed), RS-MMC (Reduced Size), SecureMMC		Mini-SD	MemoryStick, ProMemoryStick, DuoMemory Stick Pro Duo		
Volume (mm ³)	1265	1075605 (RS-MMC, MMCmobile)	5141 (Type I) 7790 (Type II)	1613 (SD) 602 (mini-SD)	3010992 (Duo and Pro Duo)	850	165
Weight (g)	2	1.5	11.4	2 (1 for mini-SD)	4 (2 for Duo & Pro Duo)	2	0.4



Interface pins	22	7-13	50	9	10	18	8
Present capacity	128 MB	1 GB	8 GB	2 GB (1 GB for mini-SD)	128 MB 4 GB (Pro and Pro Duo)	1 GB	512 MB
Content security	ID copy protection	Depends on variant	No	CPRM, SD	MagicGate	ID copy protection	CPRM, SD
Max data transfer rate	1 MB/s Write, 3.5 MB/s Read	Up to 52 MB/s (depends on variant)	66 MB/s	10 MB/s	1.8 MB/s Write, 2.5 MB/s Read, 20 MB/s (Pro and Pro Duo)	3 MB/s Write, 5 MB/s Read	1.8 MB/s typical
Comments	No on-board controller, limited capacity	Small form factor, compatible with SD interfaces	IDE-compatible	Mini-SD has adapter to fit SD card slot	Mostly Sony products, supports real-time DVD-quality video transfer	Adapters available to other card types	bridges between embedded and removable memory worlds
I/O capability?	No	No	Yes	Yes	Yes	No	No
Voltage	3.3 V/5 V	3.3 V (1.8 V/3.3 V for MMCplus and MMCmobile)	3.3 V/5 V	3.3 V	3.3 V	3.3 V	3.3 V
Interface	8-bit I/O	SPI, MMC	IDE	SPI, SD	MemoryStick	NAND Flash	SPI, SD

mass storage (dwarfing SCSI, for instance) that ATA-compliant storage has found its way into the PC-MCIA and CompactFlash worlds as well.

The ATA interface specifies the way commands are passed through to the hard drive controller, interpreted, and processed. The interface consists of eight basic registers: seven are used by the processor's I/O subsystem to set up a read or write access, and the eighth one is used to perform block reads/writes in 512-byte chunks called sectors. Each sector has 256 words, and each access utilizes a 16-bit data bus.

The original scheme for addressing drive data followed a CHS method of accessing a particular cylinder, head, and sector (CHS) of the hard disk. A more intuitive mode known as *logical block addressing (LBA)*, in which each sector on a hard disk has its own unique identification number, was soon added. LBA led the way toward breaking through the 8 GB addressing barrier inherent in the CHS approach (which allows for only 1,024 cylinders, 256 heads, and 63 sectors).

ATA lays down timing requirements for control and data accesses. It offers two modes of data access—Programmed I/O (PIO) and DMA—each offering various levels of performance. PIO modes operate in analogous fashion to direct processor accesses to the hard drive. That is, the processor has to schedule time out of what it's doing to set up and complete a data processing cycle. DMA modes, on the other hand, allow data transfer between the drive and processor memory to occur without core intervention. The term *Ultra DMA* refers to higher-performance levels of DMA that involve double-edge clocking, where data is transferred on both the rising and falling edges of the clock. Ultra DMA usually requires an 80-conductor IDE cable (instead of the standard 40-pin one for parallel ATA drives) for better noise immunity at the higher transfer rates, and it also incorporates cyclical redundancy checking (CRC) for error reduction.

Table 5.7 lists the several different levels of ATA, starting with ATA-1 and extending to ATA/ATAPI-7, which is still under development. At Level 4, ATAPI was introduced, so every level from 4 onward is designated ATA/ATAPI. The levels differ mainly in I/O transfer rates, DMA capability, and reliability. All new drives at a given ATA level are supposed to be backward-compatible to the previous ATA levels.

5.4.3.3 Other Hard Drive Interfaces

SATA

Serial ATA (SATA) is a follow-on to ATA that serializes the parallel ATA data interface, thus reducing electrical noise and increasing performance while allowing for longer, skinnier cabling. SATA uses differential signaling at very high clock rates to achieve data transfer rates of several hundred megabytes per second. SATA sheds the legacy 5 V power supply baggage that parallel ATA carries, and its differential signaling operates in the range of 200–300 mV. More good news: It's software-compatible with parallel ATA. This allows simple

Table 5.7 ATAPI summary.

ATA Version	Max PIO Transfer Rate (MB/s)	Max DMA Transfer Rate (MB/s)	Added Features
ATA-1	8.3	8.3	Original standard for IDE drives
ATA-2	16.7	16.7	Logical Block Addressing Block Transfers
ATA-3	16.7	16.7	Improved reliability, Drive security, SMART (self-monitoring)
ATA/ATAPI-4	16.7	33.3	Packet Interface extension. Ultra DMA (80-conductor ribbon cable), CRC error checking and correction
ATA/ATAPI-5	16.7	66.7	Faster Ultra DMA transfer modes
ATA/ATAPI-6	16.7	100	Support for drives larger than 137 GB
ATA/ATAPI-7	16.7	133	SATA and DVR support

SATA-to-parallel-ATA converters to translate between the two technologies in the near term, without disturbing the software layer underneath the different physical layers. On the downside, SATA allows only for point-to-point connections between a controller and a device, although external port multipliers are available to effect a point-to-multipoint scheme.

SATA is targeted primarily at the PC market, to improve reliability and increase data transfer speeds. It will slowly make its way into embedded applications, first targeting those systems where the storage density of solid state flash cards isn't adequate, leaving parallel or serial ATA as the only options. But as flash card densities grow into the several-Gbyte range, they will maintain a firm foothold in a wide swath of embedded applications due to the increased reliability of their solid-state nature and their small form factors.

SCSI

SCSI stands for Small Computer Systems Interface, a high-end interconnect that outperforms ATA but is also much more complex. It offers expandability to many more devices than ATA and has a feature set geared toward higher performance. SCSI is more expensive than ATA for the same storage capacity, and it's typically the realm of niche applications, not very popular in the embedded world.

Microdrive

The Microdrive is an actual miniature hard disk in a CompactFlash Type II form factor. Invented by IBM, the product line is now owned and propagated by Hitachi. Whereas CompactFlash is a solid-state memory, Microdrives are modeled after conventional magnetically based hard drives, with tiny spinning platters. For this reason, they are not as rugged and reliable as CompactFlash memory, and they also consume more power. Currently they are available at capacities up to several gigabytes.

USB/Firewire

These drives are nothing more than Parallel ATA disk drives with a USB 2.0 high-speed or Firewire (IEEE 1394) front end, usually used to facilitate access to a PC. However, in the embedded world, where USB and Firewire are gaining traction, these interfaces provide a handy way to add storage to a system.

USB pen drives, also called keychain drives, are not related to these ATA drives. Instead, they are Flash memory devices with a USB front end, similar in all respects to the NAND memory devices described previously.

5.4.3.4 Emerging Nonvolatile Memory Technologies

There are some exciting new technologies on the horizon that will probably be very important in high-performance embedded applications. Two of particular interest are FRAM and MRAM. FRAM, or Ferroelectric RAM, uses electric field orientation to store charge. This gives it almost infinite write capability. By comparison, conventional EEPROMs can only be written on the order of 10,000 times. MRAM, or Magnetoresistive RAM, uses electron spin to store information. It represents the best of several memory domains: it's nonvolatile, it has bit densities rivaling DRAM, and it operates at SRAM-like speeds.

5.5 Direct Memory Access

The processor core is capable of doing multiple operations, including calculations, data fetches, data stores, and pointer increments/decrements, in a single cycle. In addition, the core can orchestrate data transfer between internal and external memory spaces by moving data into and out of the register file.

All this sounds great, but in reality you can only achieve optimum performance in your application if data can move around without constantly bothering the core to perform the transfers.

This is where a direct memory access (DMA) controller comes into play. Processors need DMA capability to relieve the core from these transfers between internal/external memory and peripherals or between memory spaces. There are two main types of DMA controllers. "Cycle-stealing" DMA uses spare (idle) core cycles to perform data transfers. This is not a workable solution for systems with heavy processing loads like multimedia flows. Instead, it is much more efficient to employ a DMA controller that operates independently from the core.

Why is this so important? Well, imagine if a processor's video port has a FIFO that needs to be read every time a data sample is available. In this case, the core has to be interrupted tens of millions of times each second. As though that's not disruptive enough, the core has to perform an equal amount of writes to some destination in memory. For every core processing cycle spent on this task, a corresponding cycle would be lost in the processing loop.

We know from experience that PC-based software designers transitioning to the embedded world are hesitant to rely on a DMA controller for moving data around in an application. This usually stems from their impression that the complexity of the programming model increases exponentially when DMA is factored in. Yes, it is true that a DMA controller adds another dimension to your solution. We will, in fact, explore some intricacies that DMA introduces—such as contention for shared resources and new challenges in maintaining coherency between data buffers. Our goal, however, is to put your mind at ease, to show you how DMA is truly your friend. In this chapter, we'll focus on the DMA controller itself.

5.5.1 DMA Controller Overview

Because you'll typically configure a DMA controller during code initialization, the core should only need to respond to interrupts after data set transfers are complete. You can program the DMA controller to move data in parallel with the core while the core is doing its basic processing tasks—the jobs on which it's supposed to be focused! In an optimized application, the core would never have to move any data but rather only access it in L1 memory. The core wouldn't need to wait for data to arrive, because the DMA engine would have already made it available by the time the core was ready to access it. Figure 5.16 shows a typical interaction between the processor and the DMA controller. The steps allocated to the processor involve setting up the transfer, enabling interrupts, and running code when an interrupt is generated. The dashed lines/arrows between memory and the peripheral indicate operations the DMA controller makes to move data independent of the processor. Finally, the interrupt input back to the processor can be used to signal that data is ready for processing.

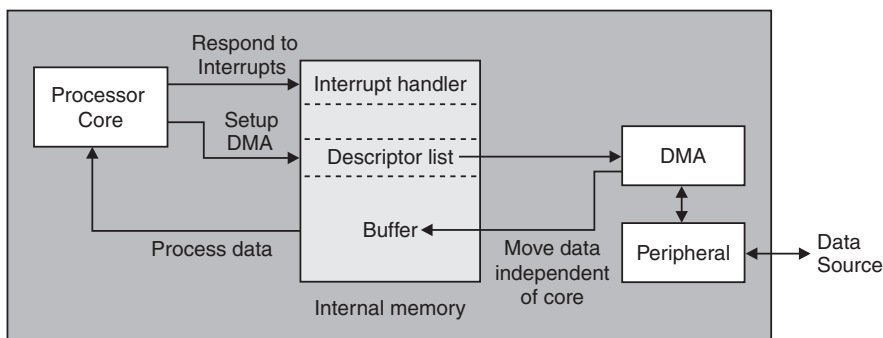


Figure 5.16: DMA controller.

In addition to moving to and from peripherals, data also needs to move from one memory space to another. For example, source video might flow from a video port straight to L3 memory, because the working buffer size is too large to fit into internal memory. We don't want to make the processor fetch pixels from external memory every time we need to perform a calculation,

so a memory-to-memory DMA (MemDMA for short) can bring pixels into L1 or L2 memory for more efficient access times. Figure 5.17 shows some typical DMA data flows.

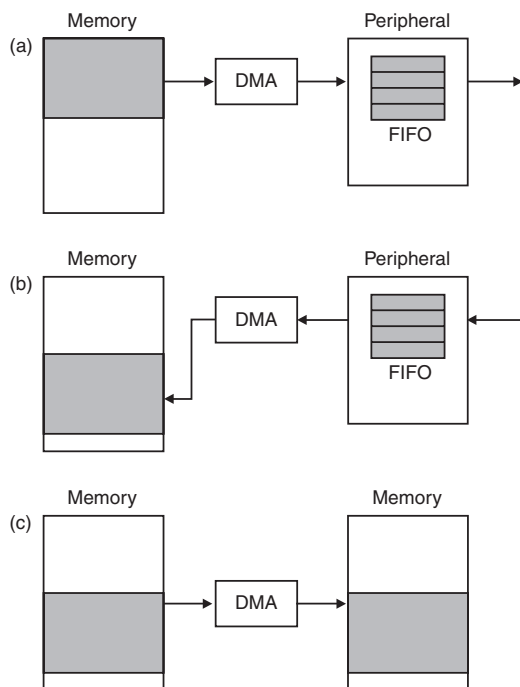


Figure 5.17: Typical DMA flows.

So far we've focused on data movement, but a DMA transfer doesn't always have to involve data. We can use code overlays to improve performance, configuring the DMA controller to move code into L1 instruction memory before execution. The code is usually staged in larger external memory.

In this chapter, we will use the Blackfin processor's DMA controller as a model to illustrate the basic concepts of direct memory access and how it can boost system performance. We will also offer some helpful ways to manage the DMA controller and review examples of "two-dimensional" transfers that can save valuable data passes by markedly reducing the time an application spends traversing a data buffer.

5.5.2 More on the DMA Controller

A DMA controller is a unique peripheral devoted to moving data around a system. Think of it as a controller that connects internal and external memories with each DMA-capable peripheral via

has to complete before the next transfer is initiated, Stop mode can guarantee this sequencing.

5.5.6 Descriptor-Based DMA

DMA transfers that are descriptor-based require a set of parameters stored within memory to initiate a DMA sequence. The descriptor contains all of the same parameters normally programmed into the DMA control register set. However, descriptors also allow the chaining together of multiple DMA sequences. In descriptor-based DMA operations, we can program a DMA channel to automatically set up and start another DMA transfer after the current sequence completes. The descriptor-based model provides the most flexibility in managing a system's DMA transfers.

Blackfin processors offer two main descriptor models—a *Descriptor Array* scheme and a *Descriptor List* method. The goal of these two models is to allow a tradeoff between flexibility and performance. Let's take a look at how this is done.

In the Descriptor Array mode, descriptors reside in consecutive memory locations. The DMA controller still fetches descriptors from memory, but because the next descriptor immediately follows the current descriptor, the two words that describe where to look for the next descriptor (and their corresponding descriptor fetches) aren't necessary. Because the descriptor does not contain this Next Descriptor Pointer entry, the DMA controller expects a group of descriptors to follow one another in memory like an array.

A Descriptor List is used when the individual descriptors are not located "back-to-back" in memory. There are actually multiple sub-modes here, again to allow a tradeoff between performance and flexibility. In a "small descriptor" model, descriptors include a single 16-bit field that specifies the lower portion of the Next Descriptor Pointer field; the upper portion is programmed separately via a register and doesn't change. This, of course, confines descriptors to a specific 64 K ($=2^{16}$) page in memory. When the descriptors need to be located across this boundary, a "large" model is available that provides 32 bits for the Next Descriptor Pointer entry.

Regardless of the descriptor mode, using more descriptor values requires more descriptor fetches. This is why Blackfin processors specify a "flex descriptor model" that tailors the descriptor length to include only what's needed for a particular transfer, as shown in Figure 5.27. For example, if 2D DMA is not needed, the YMODIFY and YCOUNT registers do not need to be part of the descriptor block.

5.5.6.1 Descriptor Management

So what's the best way to manage a descriptor list? Well, the answer is application-dependent, but it is important to understand what alternatives exist.

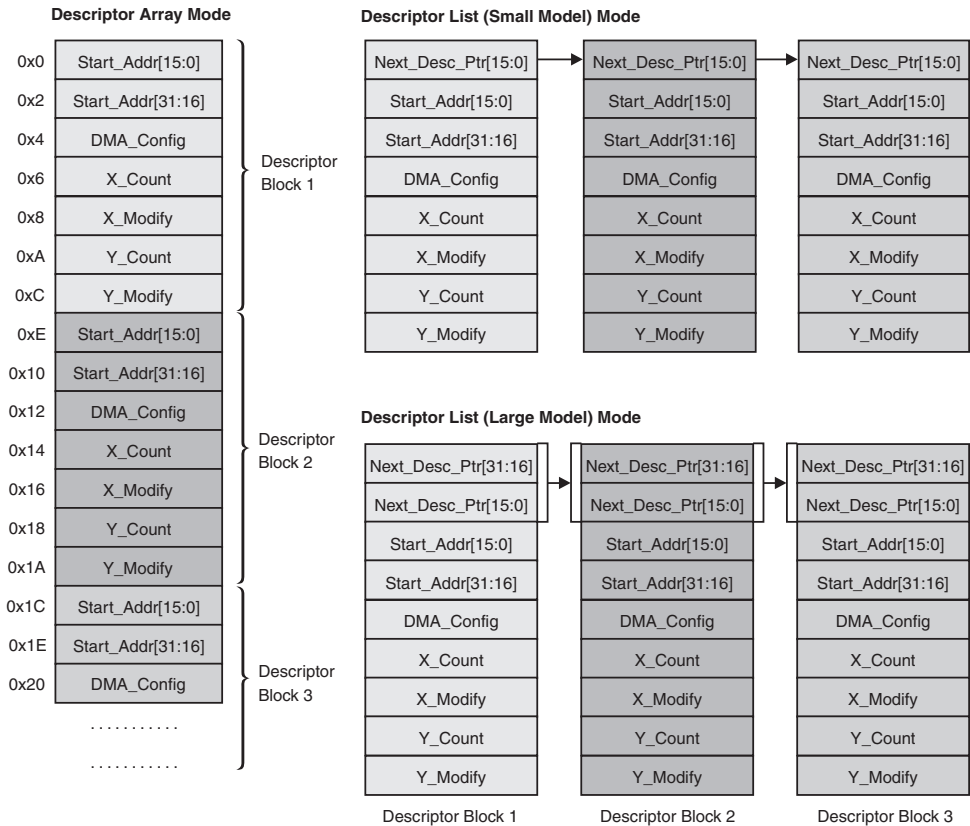


Figure 5.27: DMA descriptor models.

The first option we will describe behaves very much like an Autobuffer DMA. It involves setting up multiple descriptors that are chained together as shown in Figure 5.28a. The term “chained” implies that one descriptor points to the next descriptor, which is loaded automatically once the data transfer specified by the first descriptor block completes. To complete the chain, the last descriptor points back to the first descriptor, and the process repeats. One reason to use this technique rather than the Autobuffer mode is that descriptors allow more flexibility in the size and direction of the transfers. In our YCbCr example (Example 5.5), the Y buffer is twice as large as the other buffers. This can be easily described via descriptors and would be much harder to implement with an Autobuffer scheme.

The second option involves the processor manually managing the descriptor list. Recall that a descriptor is really a structure in memory. Each descriptor contains a configuration word, and each configuration word contains an “Enable” bit which can regulate when a transfer starts. Let’s

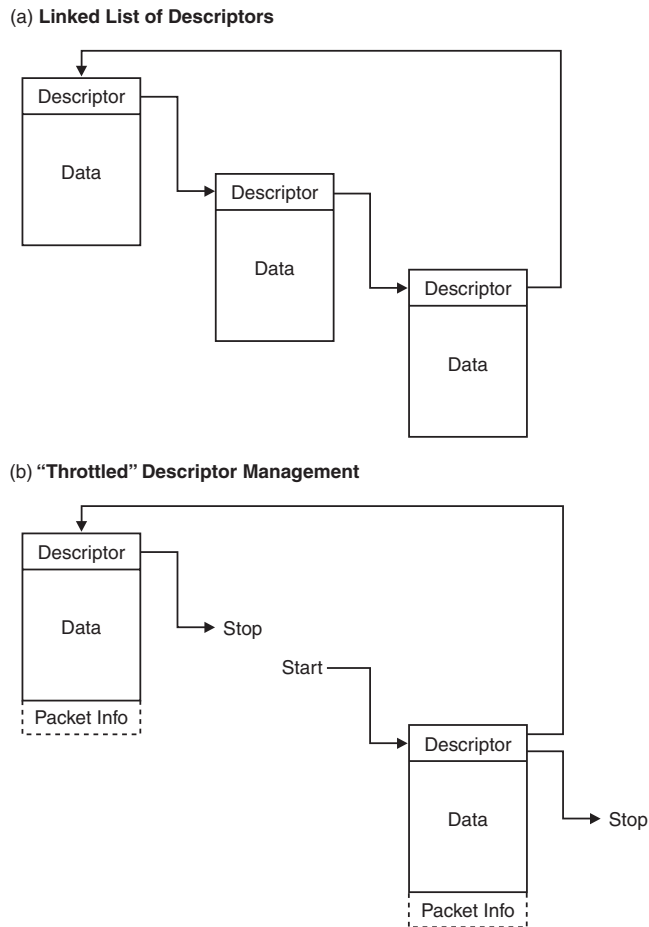


Figure 5.28: DMA descriptor throttled by the processor.

assume we have four buffers that have to move data over some given task interval. If we need to have the processor start each transfer specifically when the processor is ready, we can set up all of the descriptors in advance, but with the “Enable” bits cleared. When the processor determines the time is right to start a descriptor, it simply updates the descriptor in memory and then writes to a DMA register to start the stalled DMA channel. Figure 5.28b shows an example of this flow.

When is this type of transfer useful? EMP applications often require us to synchronize an input stream to an output stream. For example, we may receive video samples into memory at a rate that is different from the rate at which we display output video. This will happen in real systems even when you attempt to make the streams run at exactly the same clock rate.

In cases where synchronization is an issue, the processor can manually regulate the DMA descriptors corresponding to the output buffer. Before the next descriptor is enabled, the processor can synchronize the stream by adjusting the current output descriptor via a semaphore mechanism. For now, you can simply consider semaphores tools that guarantee only one entity at a time accesses a shared resource.

When using internal DMA descriptor chains or DMA-based streams between processors, it can also be useful to add an extra word at the end of the transferred data block that helps identify the packet being sent, including information on how to handle the data and, possibly, a time stamp. The dashed area of Figure 5.28b shows an example of this scheme.

Most sophisticated applications have a “DMA Manager” function implemented in software. This may be provided as part of an operating system or real-time kernel, but it can also run without either of these. In both cases, an application submits DMA descriptor requests to the DMA Queue Manager, whose responsibility it is to handle each request. Usually, an address pointer to a “callback” function is part of the system as well. This function carries out the work you want the processor to perform when a data buffer is ready, without needlessly making the core linger in a high-priority interrupt service routine.

There are two general methods for managing a descriptor queue using interrupts. The first is based on interrupting upon the completion of every descriptor. Use this method only if you can guarantee that each interrupt event will be serviced separately, with no interrupt overrun. The second involves interrupting only on completion of the work transfer specified by the last descriptor of a work block. A work block is a collection of one or more descriptors.

To maintain synchronization of the descriptor queue, you need to maintain in software a count of descriptors added to the queue, while the interrupt handler maintains a count of completed descriptors removed from the queue. The counts are then equal only when the DMA channel pauses after having processed all the descriptors.

5.5.7 Advanced DMA Features

5.5.7.1 System Performance Tuning

To effectively use DMA in a multimedia system, there must be enough DMA channels to support the processor’s peripheral set fully, with more than one pair of Memory DMA streams. This is an important point, because there are bound to be raw media streams incoming to external memory (via high-speed peripherals), while at the same time data blocks will be moving back and forth between external memory and L1 memory for core processing. What’s more, DMA engines that allow direct data transfer between peripherals and external memory, rather than requiring a stopover in L1 memory, can save extra data passes in numerically intensive algorithms.

As data rates and performance demands increase, it becomes critical to have “system performance tuning” controls at your disposal. For example, the DMA controller might be optimized