

MODERN EMBEDDED COMPUTING

Designing Connected, Pervasive, Media-Rich Systems

Peter Barry Patrick Crowley

MK
MORGAN KAUFMANN



Modern Embedded Computing

Designing Connected, Pervasive,
Media-Rich Systems

Modern Embedded Computing

Designing Connected, Pervasive, Media-Rich Systems

Peter Barry

Patrick Crowley



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

Morgan Kaufmann Publishers is an Imprint of Elsevier



Acquiring Editor: Todd Green
Development Editor: Robyn Day
Project Manager: Andre Cuello
Designer: Joanne Blank

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

© 2012 Elsevier Inc. All rights reserved. Portions previously published by and are © Intel Corporation.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data

Application submitted

British Library Cataloguing-in-Publication Data

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

ISBN: 978-0-12-391490-3

Printed in the United States of America

12 13 14 15 16 10 9 8 7 6 5 4 3 2 1

Working together to grow libraries in developing countries		
www.elsevier.com www.bookaid.org www.sabre.org		
ELSEVIER	BOOK AID International	Sabre Foundation

For information on all MK publications visit our website at www.mkp.com

To Viviane, Brianna, and Jason
—Peter

To Laura, Sophia, and Alice
—Patrick

Contents

Preface	xix
Foreword	xxi
Acknowledgments.....	xxv

PART 1 PRINCIPLES OF MODERN EMBEDDED SYSTEMS

CHAPTER 1 Embedded Systems Landscape	3
What Is an Embedded Computer System?	3
Applications and Form Factors	4
Power	4
System Resources and Features	5
User Assumptions	5
Why Is This Transition Inevitable?	5
What Range of Embedded Systems Exists?	7
What to Expect from the Rest of This Book.....	8
CHAPTER 2 Attributes of Embedded Systems	9
Embedded Platform Characteristics.....	12
Central Processing Unit (CPU)	12
Integration Level.....	13
Power Consumption.....	14
Form Factor	15
Expansion.....	17
Application-Specific Hardware	17
Certification	18
Reliability/Availability.....	18
User Interfaces	19
Connectivity.....	20
Security	20
Summary.....	21
CHAPTER 3 The Future of Embedded Systems.....	23
Technology Trends	23
Computation.....	24
Connectivity	25
Storage	29
Sensing.....	30
Issues, Applications, and Initiatives.....	30
Energy.....	30

Security	32
Health	33
Challenges and Uncertainties	34
Open Systems, Internet Access, and Neutrality	34
Privacy	35
Successful Commercialization	36
Summary	36

PART 2 EMBEDDED SYSTEMS ARCHITECTURE AND OPERATION

CHAPTER 4 Embedded Platform Architecture	41
Platform Overview	41
Processor	41
System Memory Map	43
Interrupt Controller	44
Timers	55
Volatile Memory Technologies	61
DRAM Controllers	62
SRAM Controllers	66
Nonvolatile Storage	67
NOR Flash	68
NAND Flash	70
Hard Disk Drives and Solid State Drives	72
Device Interface—High Performance	73
Peripheral Component Interconnect (PCI)	74
Universal Serial Bus	80
Programming Interface	85
Linux Driver	89
Device Interconnect—Low Performance	89
Inter-Integrated Circuit Bus	89
System Management Bus (SMBus)	91
Serial Peripheral Interface (SPI)	92
Audio Buses	93
Inter IC Sound (I ² S)	93
Universal Asynchronous Receiver/Transmitter	93
General-Purpose Input/Output	96
Power Delivery	97
Summary	97
CHAPTER 5 Embedded Processor Architecture	99
Basic Execution Environment	99
Privilege Levels	103
Floating-Point Units	104

Processor Specifics	105
Application Binary Interface.....	107
Processor Instruction Classes.....	112
Immediate Operands.....	113
Register Operands.....	113
Memory Operands	114
Data Transfer Instructions	114
Arithmetic Instructions.....	115
Branch and Control Flow Instructions	118
Structure/Procedure Instructions	119
SIMD Instructions	120
Exceptions/Interrupts Model.....	121
Precise and Imprecise Exceptions.....	122
Vector Table Structure.....	124
Exception Frame	126
Masking Interrupts	126
Acknowledging Interrupts.....	128
Interrupt Latency.....	128
Memory Mapping and Protection.....	130
Memory Management Unit.....	131
Translation Caching.....	135
MMU and Processes	135
Memory Hierarchy	136
Local Memory	138
Cache Hierarchy	138
Cache Coherency.....	142
System Bus Interface.....	145
Memory Technology.....	145
Intel Atom Microarchitecture (Supplemental Material).....	145
Microarchitecture.....	146
Front End	149
Memory Execution Cluster	151
Bus Cluster.....	152
CHAPTER 6 Embedded Platform Boot Sequence	153
Multi-Core and Multi-Processor Boot.....	153
Boot Technology Considerations	154
Hardware Power Sequences (the Pre-Pre-Boot).....	156
Reset: The First Few Steps and a Jump	157
Early Initialization.....	159
CPU Initialization	159
IA Microcode Update	159
Device Initialization	161
Memory Configuration	161

Post-Memory Setup	162
Shadowing.....	163
AP Processor Initialization.....	163
Advanced Initialization	164
General-Purpose Input/Output.....	164
Interrupt Controller.....	164
Timers	165
Cache Control.....	165
UART Serial Ports.....	166
Debug Output.....	166
Configuration Storage.....	167
PCIe Bus Initialization	167
Image Storage	168
USB.....	168
SATA.....	168
SDIO	168
Legacy BIOS and UEFI Framework Software.....	169
Legacy Operating System Boot.....	169
Extensible Firmware Interface	173
Cold and Warm Boot	176
Summary.....	177

CHAPTER 7 Operating Systems Overview..... 179

Application Interface.....	179
OS Application Interface.....	179
OS Service Calls.....	180
Processes, Tasks, and Threads	181
Task Context	184
Task State and State Transitions	184
Scheduling.....	186
Simple FIFO Scheduler	186
Round-Robin Scheduler with Priority and Preemption	187
Linux Kernel's Scheduler.....	189
POSIX-Compliant Scheduler	190
Memory Allocation	191
Virtual Memory and Protection.....	193
Freeing Memory	195
Swapping Memory.....	195
Clocks and Timers.....	195
Synchronous Execution	195
Asynchronous Execution	196
Time of Day.....	197
Mutual Exclusion/Synchronization.....	197
Device Driver Models	202

Low-Level Data Path	207
Direct Memory Access	209
Memory Addresses	210
Bus Drivers	212
Networking	213
Buffer Management	215
Polling Interface	215
Acceleration	216
Storage File Systems	216
Device Wear and Tear	218
Power Interactions	219
Power Management	219
Real Time	221
Device Interrupt Delivery	221
Processor Interrupt Handler	222
Deferred Task	222
RTOS Characteristics	223
Licensing	224
CHAPTER 8 Embedded Linux	227
Tool Chain	227
Getting the Tools	228
Tools Overview	229
Anatomy of an Embedded Linux	231
Building a Kernel	234
Kernel Build	234
Kernel Options	236
Root File System Build	239
Busybox	242
C Library	243
Boot Sequence	244
Debugging	246
Debugging Applications	246
Kernel Debugging	247
Driver Development	249
Character Driver Model	250
PCI Device Drivers	256
Interrupt Handling	258
Memory Management	262
User Space	262
Access to User Space Memory from the Kernel	262
Kernel Allocation	263
Page Allocation	264
The <code>kmalloc()</code> Function	265

PCI Memory Allocation and Mapping.....	265
Synchronization/Locking	267
Atomic Operations.....	267
Spinlock	267
Semaphore	268
Summary.....	268
CHAPTER 9 Power Optimization	269
Power Basics	269
The Power Profile of an Embedded Computing System.....	270
Constant Versus Dynamic Power.....	271
Constant Power	271
Dynamic Power	271
A Simple Model of Power Efficiency	273
Advanced Configuration and Power Interface (ACPI).....	275
Idle Versus Sleep	277
ACPI System States.....	277
Global System States (Gx States).....	278
Sleep States (Sx States)	278
Device Power States (Dx States).....	279
Processor Power States (Cx States).....	280
Processor Performance States (Px States).....	280
Enhanced Intel SpeedStep Technology	281
Optimizing Software for Power Performance	281
Race to Sleep	281
The Linux PowerTOP Tool	282
Basic PowerTOP Usage.....	282
Using PowerTOP to Evaluate Software and Systems	284
Summary.....	289
CHAPTER 10 Embedded Graphics and Multimedia Acceleration	291
Screen Display	293
Display Engine.....	293
Window Management	296
Screen Composition	296
Embedded Pannels.....	297
Display Query and Timing	299
Copy Protection.....	299
Graphics Stack.....	299
Accelerated Media Decode	301
Lip Syncing	303
Video Capture and Encoding	304
Video Capture	304

Media Frameworks	310
GStreamer	310
OpenMAX™	313
Framework Summary	315
Summary	315
 CHAPTER 11 Digital Signal Processing Using General-Purpose Processors ...	317
Overview	318
Signals	318
DSP Building Blocks	319
Data Acquisition	321
Fixed-Point and Floating-Point Implementations	322
Single Instruction Multiple Data	324
SIMD Microarchitecture and Instructions	324
Operating System	324
Microarchitecture Considerations	325
Implementation Options	325
Intrinsics and Data Types	326
Vectorization	328
Performance Primitives	331
Finite Impulse Response Filter	332
FIR Example: C Code	333
FIR Example: Intel Performance Primitives	333
FIR Example: Intel SSE	334
Application Examples	337
Codec	337
Medical Ultrasound Imaging	339
Performance Results	344
Summary	346
 CHAPTER 12 Network Connectivity	347
Networking Basics	349
Layering and Network Software	349
Node Operation and Network Hardware	350
Sockets and a Simple Example	351
TCP/IP Networking	353
Governance, the IETF, and RFCs	354
Addresses, Packets, and Routes	355
Port Numbers, Byte Ordering, and OS Tools	359
Supporting Protocols and Services	360
Ethernet	361
History	361

Protocol Description	362
Ethernet MAC Addresses	363
Ethernet Packet Format.....	363
A Gigabit Ethernet Controller and Its Features	364
Wi-Fi and IEEE 802.11	365
History	366
Protocol Description	366
Frame Format	367
A Wi-Fi Adapter and Its Features.....	368
Bluetooth.....	369
History	369
Protocol Details.....	369
Packet Format	370
Linux Networking.....	372
Tools and Monitor and Control Network Interfaces and Sockets	372
Programming Sockets in C.....	372
Linux Kernel Networking Structures	376
Summary.....	378
CHAPTER 13 Application Frameworks	379
Overview.....	379
Android	379
Android Framework Architecture.....	381
Android Application Architecture	384
Android Development Environment.....	389
Deployment	391
Qt	392
Qt Application Development Framework	392
Qt Creator.....	394
Other Environments.....	394
More Resources	395
Summary	395
CHAPTER 14 Platform and Content Security	397
Security Principles.....	398
Confidentiality, Integrity, and Availability (CIA).....	400
Security Concepts and Building Blocks	402
Encryption and Cryptography.....	402
Secure Web Communications: TLS	404
Secure Shell (SSH)	407
Security Architecture for IP: IPSec.....	408
Two-Factor Authentication	411
Major Categories of Security Attacks	411

Firewalls	416
Servers and Logs	419
Platform Support for Security	420
Summary	421
CHAPTER 15 Advanced Topics: SMP, AMP, and Virtualization	423
Multiprocessing Basics	424
History and Motivation	424
A Concrete Example	428
Physical versus Logical Cores	432
Impact on Systems and Software	432
Symmetric Multiprocessing	433
Overview	433
Linux SMP Support	433
Interprocess Communication	435
Asymmetric Multiprocessing	435
Concepts and Motivation	436
System Organization	436
Virtualization Basics	437
History and Motivation	437
Basic Concepts	437
Methods for Platform Virtualization	438
Paravirtualization	438
Hardware Support for Virtualization	439
Linux VServers	439
Xen™	439
Xenomai	440
Summary	442

PART 3 DEVELOPING AN EMBEDDED SYSTEM

CHAPTER 16 Example Designs	445
Intel Atom E6XX Series Platforms	445
Architecture Overview	446
Platform Controller Hub(s)	448
Multi-Radio Communications Design	452
Hardware Platform	452
Software Platform	455
Multimedia Design	458
Hardware Platform	458
Software Platform	459
Modular References	463
Summary	464

CHAPTER 17	Platform Debug	465
	Debugging New Platforms	465
	A Process for Debugging a New Platform	466
	Debug Tools and Chipset Features.....	467
	Oscilloscopes.....	468
	Logic Analyzers	468
	Bus Analyzers	468
	Power-On Self-Test (POST) Cards.....	468
	JTAG Adapters.....	469
	Debug Process Details	469
	Visual Examination	469
	Hardware Evaluation.....	470
	Software Evaluation	474
	Additional Resources.....	474
	Summary	475
 CHAPTER 18	 Performance Tuning	 477
	What Are Patterns?.....	477
	General Approaches	478
	Defined Performance Requirement.....	478
	Performance Design	478
	Premature Code Tuning Avoided	479
	Step-by-Step Records.....	479
	Slam-Dunk Optimization	480
	Best Compiler for Application	480
	Compiler Optimizations.....	481
	Data Cache	482
	Code and Design	483
	Reordered Struct	483
	Supersonic Interrupt Service Routines	483
	Assembly-Language-Critical Functions	484
	Inline Functions.....	484
	Cache-Optimizing Loop.....	484
	Minimizing Local Variables	485
	Explicit Registers	485
	Optimized Hardware Register Use	485
	Avoiding the OS Buffer Pool.....	486
	C Language Optimizations	486
	Disabled Counters/Statistics	487
	Processor-Specific.....	488
	Stall Instructions	488
	Profiling Tools.....	488
	Prefetch Instructions	489

Separate DRAM Memory Banks..... 489

Line-Allocation Policy 490

Cache Write Policy 490

Cache-Aligned Data Buffers..... 491

On-Chip Memory 491

Optimized Libraries 491

Modulo/Divide Avoided..... 492

Networking Techniques 492

 Bottleneck Hunting 492

 Evaluating Traffic Generator and Protocols 493

 Environmental Factors 494

References..... 497

Index 503

The IA watchdog timer provides a resolution that ranges from 1 μ s to 10 minutes. The watchdog timer on the Embedded Atom SOC platform uses a 35-bit down-counter. The counter is loaded with the value from the 1st Preload register. The timer is then enabled and it starts counting down. The time at which the watchdog timer first starts counting down is called the first stage. If the host fails to reload the watchdog timer before the 35-bit down-counter reaches zero, the watchdog timer generates an internal interrupt. After the interrupt is generated, the watchdog timer loads the value from the second Preload register into the watchdog timer's 35-bit down-counter and starts counting down. The watchdog timer is now in the second stage. If the processor still fails to reload the watchdog timer before the second timeout, the watchdog triggers one of the following events:

- *Assert a General-Purpose Input/Output Pin* (GPIO4 on Intel Atom E600 Series). The GPIO pin is held high until the system is reset by a circuit external to the SOC.
- *Warm Reset*. This triggers the CPU to restart from the startup vector.
- *Cold Reset*. This is a reset of the SOC device and the CPU restarts from the startup vector.

The process of reloading the WDT involves the following sequence of writes, as we mentioned to reduce the likelihood of an accidently servicing the timer.

1. Write "80" to Watchdog Reload Register zero.
2. Write "86" to Watchdog Reload Register zero.
3. Write "1" to the Reload Register.

When a watchdog restart event occurs, it is very useful for the software to understand if the watchdog timer triggered the restart. To that end, the watchdog controller has a bit that indicates if the last restart was caused by a system reset. The bit itself is not cleared as part of the system reset.

Timer Summary

The hardware timers all provide a similar set of capabilities; the mechanism of setup and control all vary depending on the platform, but at heart they are all the same. The operating system will take ownership of at least one timer in the system for its uses. The application software then most often uses operating system timer services that are derived from this single source. In some cases you will write your own software for a particular timer; then you have to develop the setup code as well as, critically, the software to handle the interrupt generated by the timer expiration. In many cases you may just want to read the counter value to implement a simple delay function that does not depend on the frequency of the processor or speed of executing a particular code sequence.

VOLATILE MEMORY TECHNOLOGIES

A complete embedded system is composed of many different memory technologies.

Intel platforms have two distinct address types. The first is for I/O devices; this is read using either IN/OUT assembly instructions or, more likely, using normal MOV instructions to a dedicated part of the address map known as memory-mapped I/O space (MMIO). When a program reads to an MMIO space it is routed to a device. The other key address space is memory. The memory space is mapped to memory devices on the platform such as the DRAM on the platform, a flash ROM device, or, in some SOCs, local on-die SRAM memory. The hardware block that converts internal memory transactions to

access the memory device is known as a memory controller. We discuss some key attributes of the DRAM memory controller in this section.

DRAM Controllers

Dynamic random access memory is a form of volatile storage. The bit value at a memory location is stored in a very small capacitor on the device. A capacitor is much smaller than a logic gate, and therefore the densities are greater than memory technologies using logic such as caches for the processor or SRAM block on an SOC. The ratio is significant. For example, a level two cache of 1 MB is reasonably large for a cache on an embedded device, whereas a device such as the Intel Atom Processor E6xx Series supports up to 2 GB of DRAM.

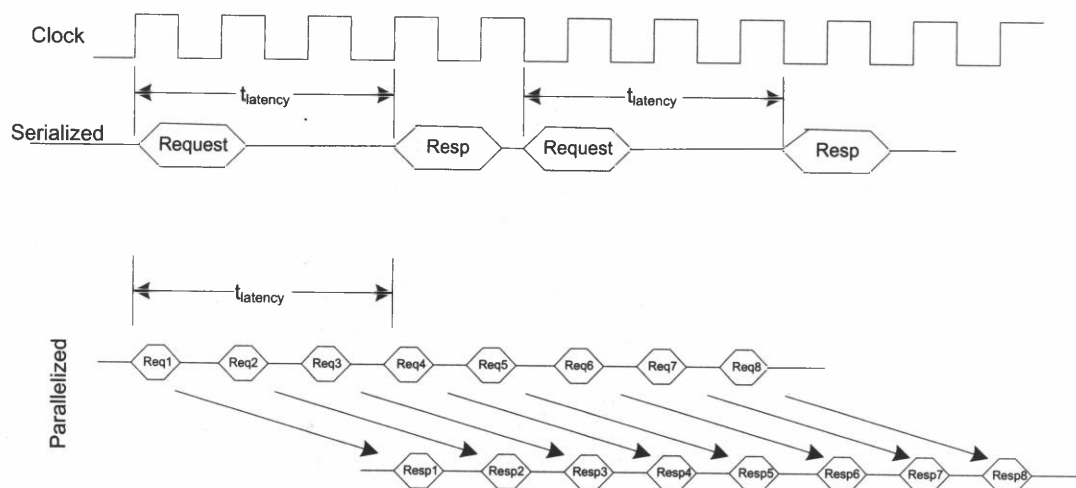
Although the densities are considerably higher, the price per bit is lower. However, there is a system trade-off that takes place. The DRAM read time is usually far slower than memory that is made from logic gates, such as the CPU cache or SRAM devices (both internal or external).

The read latency is a key performance attribute of memory. Although many strategies are used in processor and cache design to reduce the effect of read latency to memory, in many cases it will directly impact the performance of the system. After all, eventually the processor will have to wait for the result of the read from memory before it can continue to make progress. In order to read a DRAM bit, a logic device must check the value stored on the capacitor for the bit. This is done using a sense amplifier. The sense amplifier takes a little time to figure out whether the bit has a charge or not.

Another key performance attribute is the overall bandwidth or throughput supported by the DRAM devices. Whereas the read latencies of DRAM technology have not been reduced at nearly the same pace as the increase in density or overall throughput performance, the throughput performance has increased dramatically over time. The key improvements in throughput performance of DRAM are accomplished by pipelining as many read and write requests to the memory devices as possible. The DRAM device duplicates many of the logic resources in the device to have many concurrent lookups (sensing) occurring. Figure 4.12 shows the increased performance through multiple parallel requests to the DRAM. DDR memories have separate requests and response lines; this allows the requester interface to enqueue multiple requests while waiting for the response to review requests.

It is important that memory interfaces are standardized to allow many different manufacturers to develop compatible devices for the industry. At the time of writing, DDR2 SDRAM is the predominant DRAM type used in embedded platforms. DDR3 is becoming the DRAM technology used on mainstream compute platforms. The adoption of memory technologies in embedded platforms usually lags behind the adoption of the technology in mainstream platforms. This is often due to a price premium for the memory when it is first introduced into the market.

The DRAM interface as well as many physical attributes are standardized by JEDEC. This organization is the custodian of the DDR1, 2, and 3 SDRAM standards. These devices are known as synchronous DRAMs because the data are available synchronously with a clock line. DDR stands for double data rate and indicates that data are provided/consumed on both the rising and falling edge of the clock. As we mentioned above, the throughput of the memory technology can be increased by increasing the number of parallel requests to the memory controller. Both the depth of pipelining and the clock speed have increased from DDR1 to DDR2, and DDR3 has resulted in a significant increase in throughput throughout the generations of DRAM (see Table 4.4).

**FIGURE 4.12**

Increased Performance through Pipelining.

The memory controller on the Intel Atom Processor E6xx Series supports DDR2-667 and DDR2-800 megatransfers per second (MT/s).

Unlike mainstream computers, embedded platforms usually solder the memory directly down on the printed circuit board (PCB). This is often due to the additional cost associated with a module and connector, or mechanical/physical concerns with respect to having the memory not soldered to the platform. Clearly, if the memory is directly soldered to the platform, there is limited scope for upgrading the memory after the product has been shipped. The ability to expand the memory in an embedded system post-shipment is not usually required. When the memory is directly soldered down, the boot loader or BIOS software is typically preconfigured to initialize the exact DRAM that has been placed on the board. However, when modules are used such as the dual in-line modules (DIMM) prevalent on mainstream computers, the BIOS or boot loader has to establish the attributes of the DIMM plugged in. This is achieved by a Serial Presence Detect (SPD) present on the DIMM modules. The SPD is an EPROM that is accessed via a serial interface SPI explained later. The EPROM contains information about the DIMM size and memory timing required. The BIOS takes this information and configures the memory controller appropriately.

The memory cells within the DRAM device are organized in the device in a matrix of rows by columns, shown in Figure 4.13. A memory transaction is split into two phases; the first is called the

Table 4.4 DRAM Performance

DRAM Tech	Memory Clock Speeds	Data Rates
DDR2	100–266 MHz	400s–1066 MT/s
DDR3	100–266 MHz	800–2133 MT/s

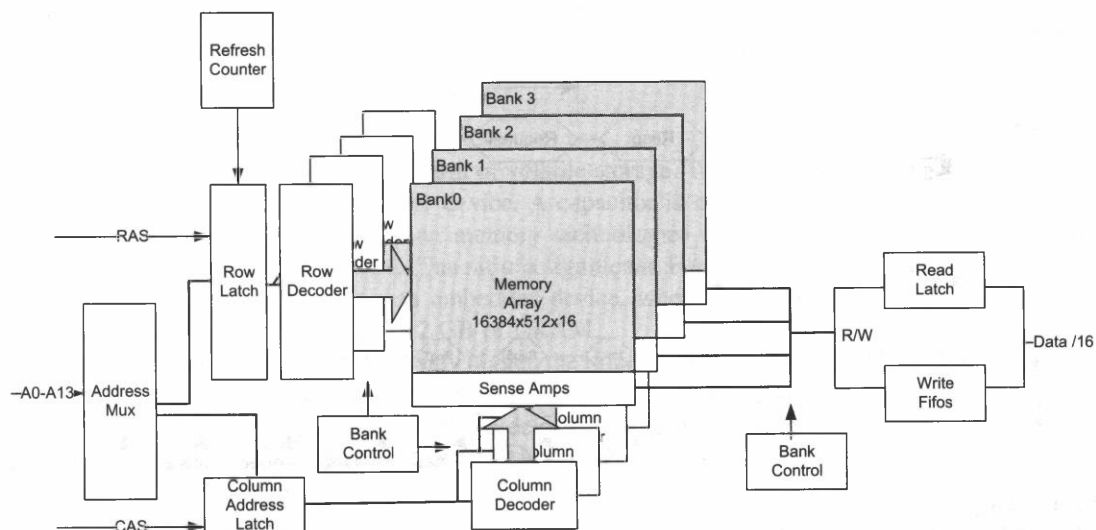


FIGURE 4.13

DDR Overview.

Row Address Strobe (RAS). In this phase an entire row of address bits is selected from the matrix; all the bits in the row are sent to the bit sense amplifiers and then into a row buffer (also called a Page). The process of reading a row of data from the capacitors is actually destructive, and the row needs to be written back to the capacitors at some point. Before entering the second phase for the transaction, the memory controller must wait for the RAS to CAS Latency, called RAS CAS Delay (t_{RCD}). The second phase of the transaction is called the Column Address Strobe (CAS). In this phase the word being read is selected from the previous row selected. The CAS Latency (t_{CL}) is often a figure quoted; it is the time from the presentation of the column address strobe to getting the data back to the memory controller. It is measured in memory clocks, not nanoseconds. Once the memory transaction has been completed, a command called Precharge needs to be issued unless the next request falls in the same row. Precharge closes the memory row that was being used. RAS Precharge Time (t_{RP}) is the time taken between the Precharge command and the next active command that can be issued. That is, the next memory transaction generally won't start until the time has been completed (more on that later). As you can see, the overall latency performance of the memory device consists of three separate latencies. If we can continue to make use of the row that has been loaded into the sense amplifier logic, then we can get subsequent data with much lower latency. To this end the memory devices and controller support a burst memory command. This is where the subsequent data are provided with much lower latency. The typical burst size supported is 4. Given that modern processors have a cache subsystem, the burst memory command is a very useful transaction to exchange data between the cache and memory.

The normal burst mode support expects the initial address to be aligned to the size of the burst; for example, if the DRAM were a 32-bit device with a burst size of 4, the initial address for a burst would have to be 0xFFFF,XX00. The initial data returned would be for the first double word, then 0xFFFF,XX04, 0xFFFF,XX08, and lastly 0xFFFF,XX0C. If, however, the processor were

waiting for the last double word in the burst (0xFFFF,XX0C), then using a burst command would decrease the performance of the processor waiting for the read result. To mitigate this, many memory controllers support a feature called Critical Word First. This CRW feature provides the data that the processor are actually waiting for first and then returns the other data elements of the burst. In the example above, the device would return the data for 0xFFFF,XX0C, 0xFFFF,XX00, 0xFFFF,XX04, and finally 0xFFFF,XX08. This would reduce the overall latency experienced by the processor but would still make use of data locality to load the cache line while the DRAM row was still active.

Another optimization to the DRAM subsystem is to have a policy on how long to keep the row buffer (page) active. As we mentioned, subsequent access to an active or open page takes less time than that required to load a new row. Given that programs often have high degrees of spatial and temporal locality, it is often best for the controller to keep a page open until a new row is accessed. This does, however, delay the time to open the new row when a new row is needed. Keeping a row active until a request is made outside that row is known as a page open policy. If the controller proactively closes the page once the transaction is complete, it is known as having a page close policy. The page sizes are relatively large; on Embedded Atom SOC's the memory controller supports devices with 1-kB, 2-kB, and 4-kB pages. Since there can be a considerable advantage to reading from an open page, the DRAM devices are partitioned into independent banks, each with its own page (row buffer) support. The SOC supports 4- and 8-bank DRAM devices. With 4–8 banks, memory controllers can thus keep multiple pages open, improving chances of burst transfers from different address streams.

In an embedded system you often have more flexibility and control over how memory is allocated and portioned in the system. A reasonable performance advantage can be seen by allocating different usage patterns to different banks. For example, a gain of approximately 5% can in some cases be achieved in a simple network router where the memory allocated to the network packets, operating system code space, and applications were all allocated from a separate banks.

The memory devices come in varying bit densities and data bus widths (x4, x8, x16, x32). The total memory required can be built up using any number of options. For example, one could use a 256-Mb device that has a 32-bit interface or two 128-Mb devices each with a 16-bit interface connected to the memory bus. The DRAM controller needs to be set up with the appropriate configuration to operate correctly. The rationale for determining the appropriate configuration usually comes down to the size of the devices on the platform and the amount of space available, as fewer higher density devices are required. The decision also has an economic dimension; higher-density memories are usually more expensive than lower-density ones.

No discussion of DRAM is complete without mentioning the refresh cycle. The DRAM cells are made up of tiny capacitors, and because a capacitor's charge can leak away over time, this could ultimately result in a 1 bit turning into a 0 bit. Clearly, this is not acceptable. Each row in each device must go through a refresh cycle with sufficient frequency to guarantee that the cell does not discharge. Generally, each DRAM device must be entirely refreshed within 64 ms. If a device has 8192 ROWs, then the refresh cycle must occur every 7.8 μ s. The refresh cycle consists of a CAS before RAS cycle where the ROW address to memory array determines the row to be refreshed. Some DDR devices have their own refresh counter, which updates ROW address during refresh cycle; the DRAM controller just triggers the refresh cycle and does not have to keep track of refresh ROW addresses. Some platforms support a DDRx's self-refresh capability; in this case the DRAM device performs the refresh cycle while the device is in low power mode with most of its signals tristated. This is usually used to allow

the memory controller to be powered down when the system is in a sleep state, thus reducing the power required by the system in this state. The ability of a capacitor to retain charge is temperature dependent; if you develop a platform that supports an extended temperature range, the refresh rate may need to be increased to ensure data retention in memory cells. It is important to refresh with the required interval. Most of your testing will probably be done at normal room temperature, where the chance of a bit fading is relatively low. However, when your platform is deployed and experiences real-world temperatures, you really don't want to be trying to debug the random system crashes you might experience. I've been there, and it's no fun at all.

Even if the DRAM is being refreshed with the required interval, there is a small probability that a capacitor may spontaneously change its value, or an error in the reading of a bit may occur. These errors are known as *soft errors* and are thought to be the result of cosmic radiation, where a neutron strikes a portion of the logic or memory cell. To mitigate the effects, designers add a number of extra redundant bits to the devices. These extra bits record the parity of the other bits, or an error correcting code covering the data bits. Parity protection requires just a single bit (for 8 bits of data) and allows the controller to detect a single bit error in the memory. It cannot correct the bit error, as there is insufficient information. When a parity error is detected, the memory controller will typically generate an exception to the processor, as the value read from memory is known to be bad. Using additional protection bits, single bit errors in the data can be corrected, while double bit errors can be detected. The most popular protection used today is known as Error Correcting Code (ECC). The scheme uses 2 bits per 8 bits covered. When the memory controller detects a single bit error, it has enough redundant information to correct the error; it corrects the value on the fly and sends the corrected value to the processor. The error may still exist in the memory, so the controller may automatically write back the correct value into the memory location, or in some systems an interrupt is raised and the software is responsible for reading the address and writing back the same value (while interrupts are disabled). A double bit error cannot be corrected and will raise an exception to the processor. When any new data are written to memory, the controller calculates the new ECC value and writes it along with the data. During the platform startup, there is no guarantee that the error bits in the device are consistent with the data in the device. The boot code or in some cases the memory controller hardware has to write all memory locations once to bring corresponding ECC bits to the correct state before any DDR location is read. Since ECC has the ability to correct a single bit error, it is prudent to have a background activity that reads all memory locations over a period of time and self-correct memory location if a single bit soft error is encountered. Such a process is known as scrubbing. Since fixing single bit errors obviously prevents a double bit error from occurring, scrubbing drastically reduces memory data loss due to soft errors.

SRAM Controllers

Static random access memory is a volatile storage technology. The technology used in the creation of an SRAM cell is the same as that required for regular SOC logic; as a result, blocks of SRAM memory can be added to SOCs (as opposed to DRAM, which uses a completely different technology and is not found directly on the SOC die). The speed of SRAM is usually much faster than DRAM technologies, and SRAM often responds to a request within a couple of CPU clock cycles. When the SRAM block is placed on die, it is located at a particular position in the address map. The system software and device drivers can allocate portions of the SRAM for their use. Note that it is

unusual for the operating system to manage the dynamic allocation/de-allocation from such memory; it is usually left up to the board support package to provide such features. SRAM memory is commonly allocated for a special data structure that is very frequently accessed by the processor, or perhaps a temporal streaming data element from an I/O device. In general, these SRAM blocks are not cache-coherent with the main memory system; care must be taken using these areas and they should be mapped to a noncached address space. On some platforms, portions of the cache infrastructure can be repurposed as an SRAM, the cache allocation/lookup is disabled, and the SRAM cells in the cache block are presented as a memory region. This is known as *cache as RAM*; some also refer to this very close low-latency RAM (with respect to the core) as *tightly coupled memory*.

The cells in a CPU cache are often made from high-speed SRAM cells. There is naturally a trade-off; the density of SRAM is far lower than DRAM, so on-die memories are at most in the megabyte range, whereas DRAMs is often in gigabytes.

Given that the read/write access time of SRAM memory is far faster than DRAM, we don't have to employ sophisticated techniques to pipeline requests through the controller. SRAM controllers typically either handle one transaction at time, or perhaps pipeline just a small number of outstanding transactions with a simple split transaction bus. There is no performance advantage from accessing addressing in a line as is the case for DRAM.

You should understand the memory sub-word write behavior of the system can be lower than you might expect. As the density of the memory cells reduces, they can fall victim to errors (as in the case of DRAM), so in many cases the SRAM cells have additional redundancy to provide ECC error bits. When the software performs a sub-word write (such as a single byte), the SRAM controller must first perform a word read, merge in the new byte, and then write back the update word into the SRAM with the correct ECC bits covering the entire word. In earlier SRAM designs without ECC, the SRAM often provided a byte write capability with no additional performance cost.

NONVOLATILE STORAGE

All embedded systems require some form of nonvolatile storage. Nonvolatile storage retains data even when the power is removed from the device. There are a range of technologies with varying storage capacities, densities, performance reliability, and size. There are two primary nonvolatile storage technologies in use today: the first and most prevalent for embedded systems is solid state memory, and the second is magnetic storage media in the form of hard drives.

Modern solid state memory is usually called flash memory. It can be erased and reprogrammed by a software driver. Flash memory read speed is relatively fast (slower than DRAM, but faster than hard drives). Write times are typically much slower than the read time. There are two distinct flash memory device types: NOR flash and NAND flash. They are named after the characteristic logic gate used in the construction of the memory cells. These two device types differ in many respects, but one key difference is that NAND flash is much higher density than NOR devices. Typically, NOR flash devices provide several megabytes of storage (at the time of writing, Spansion offered 1-MB to 64-MB devices), whereas NAND devices provide up to gigabytes of storage (devices in the 512-MB range are typical). Most commercial external solid state storage devices such as USB pen drives and SD cards use NAND devices with a controller to provide access to the device. In embedded use cases, it is usual

to directly attach the flash devices to the SOC through the appropriate interface and solder the devices directly to the board.

NOR Flash

NOR flash memory devices are organized as a number of banks. Each bank contains a number of sectors. These sectors can be individually erased. You can typically continue to read from one bank while programming or erasing another. When a device powers on, it comes up in read mode. In fact, when the device is in read mode, the device can be accessed in a random access fashion: you can read any part of the device by simply performing a read cycle. In order to perform operations on the device, the software must write specific commands to specific addresses and data patterns into command registers on the device. Before you can program data into a flash part, you must first erase it. You can perform either a sector erase or full chip erase. In most use cases you will perform sector erases. When you erase a sector, it sets all bits in the sector to one. Programming of the flash can be performed on a byte/word basis.

An example code sequence to erase a flash sector and program a word is shown below. This is the most basic operation, and there are many optimizations in devices to improve write performance. This code is written for a WORD device (16-bit data bus).

```
//Sector Erase
// ww(address,value) : Write a Word
// See discussion on volatile in GPIO section below
unsigned short sector_address
ww(sector_address + 0x555), 0x00AA); // write unlock cycle 1
ww(sector_address + 0x2AA), 0x0055); // write unlock cycle 2
ww(sector_address + 0x555), 0x0080); // write setup command
ww(sector_address + 0x555), 0x00AA); // write additional unlock cycle 1

ww(sector_address + 0x2AA), 0x0055); // write additional unlock cycle 2
ww(sector_address), 0x0030); // write sector erase command */
.. you should now poll to ensure the command completed successfully.

// Word Program into flash device at program_address
ww(sector_address+ 0x555), 0x00AA); // write unlock cycle 1
ww(sector_address+ 0x2AA), 0x0055); // write unlock cycle 2
ww(sector_address+ 0x555), 0x00A0); // write program setup command
ww(program_address), data); // write data to be programmed
// Poll for program completion
```

When programming a byte/word in a block, you can program the bytes in the block in any order. If you need to program only one byte in the block, that's all you have to write to.

A parallel NOR interface typically consists of the following signals:

- Chip Enable: a signal to select a device, set up by an address decoder.
- Output Enable: used to signal that the device place the output on the data bus.
- Address bus: A0.A20 required number of address bits to address each byte/word in the device (directly).
- Data bus: D0–D7, (and D8–D15 for word devices). The data for a read/write are provided on the data bus.

- Write Enable: signaled when a write to the device is occurring.
- Write Protect: if active it prevents erase or writes to the flash device. As a precaution, this is hardwired on the board (perhaps with a jumper switch) or connected to a general-purpose output pin under software control.

The cycles used to access the device are consistent with many devices using a simplified address and data bus.

In order to discover the configuration of a flash device such as size, type, and performance, many devices support a Common Flash Interface. A flash memory industry standard specification [JEDEC 137-A and JESD68.01] is designed to allow a system to interrogate the flash.

Many flash devices designate a boot sector for the device. This is typically write-protected in a more robust manner than the rest of the device. Usually it requires a separate pin on the flash device to be set to a particular level to allow programming the device. The boot sectors usually contain code that is required to boot the platform (the first code fetched by the processor). The boot sector usually contains enough code to continue the boot sequence and find the location of the remaining code to boot from, and importantly it usually contains recovery code to allow you to reprogram the flash remainder of the flash device.

The initial program of flash devices that are soldered on to a platform is usually carried out through the JTAG chain. This is presented through a connector that can be attached to a JTAG programmer or ICE (see Chapter 17, “Platform Debug”). The device has access to the flash device directly and can program it by emulating the behavior of a host processor in read/writing the device.

On Intel platforms, the initial instructions fetched (BIOS or boot loader) are usually stored on a NOR flash device. The flash devices can provide either a serial or parallel interface. In most cases, a serial interface is used on Intel platforms, although many others use a parallel interface.

There are some reliability differences between NOR and NAND flash devices. NOR devices are more reliable and less susceptible to bit loss. As a result, NOR flash devices are often considered the safest memory type for storing the initial program sequence for the processor (although NAND devices do have techniques to mitigate this by duplicating copies of the boot block and adding additional ECC bits to the sectors).

With random access read/write capabilities, the NOR flash more naturally supports a mode of operation called eXecute in Place (XIP). This is where a program can run directly from the flash (without first copying the data from flash to DDR memory).

Flash devices are characterized by the number of erase cycles a device supports before it starts to fail. Current devices support up to 100,000 erase cycles per sector. If flash memory is being used for a file system or to store logs of frequently changed data, then the sectors are frequently erased and reprogrammed. Software drivers must ensure that all the activity is not focused on a small number of sectors. This strategy is known as wear leveling, and all flash file systems support the principle.

The Linux kernel uses the Memory Technology Device (MTD) interface. MTD provides a generic interface for flash devices. This layer manages the individual devices at the lowest level. There are a number of flash file systems that are also provided (with differing characteristics) built up on top of the MTD layer. The most commonly used at present is the Journaling Flash File System 2 (JFFS2).

We mentioned that erasing a block sets all bits to 1. Through programming you can convert any bit with a value of 1 to zero. You can even do this without erasing. You can never convert a 0 bit to a 1 without an erase cycle. Many flash file systems make use of this attribute to help to maintaining metadata associated with a file system.

In addition to programmable storage, many devices (NOR and NAND) offer One Time Protect/Programmable (OTP) storage. This is used to store permanent information such as a serial number IMEI/ESN information, secure boot code, or SIM-Lock. The OTP bits are programmed at the time of manufacturing and cannot be updated.

NAND Flash

In contrast to the NOR flash, NAND flash does not provide a direct random access address interface. The devices work in a page/block mode. To read the device the driver must first request a page before the data from the page can be accessed. This is not unlike other secondary storage mechanisms such as disk drives, and as such it is relatively straightforward to build a file system on top of NAND devices. Erasing and programming occur in block modes. Unlike the NOR device, the programming of a block must occur in serial fashion (incremental address programmed in order). Each block has a number of pages. The page size is typically 2 kB. The time taken to load the page into the page register is relatively slow in comparison to the subsequent reads from the page register. To improve performance, many divides contain a number of page registers.

NAND (and to a lesser extent NOR) devices may contain bad pages. These bad pages must be managed by the controller (hardware or software) that provides the interface to the device. To improve the yield of NAND devices, the devices may be released with marked bad blocks. NAND devices may start life with bad blocks and further degrade over the lifetime of the device. The bad blocks found during manufacture are typically marked in spare bits within each block. You must not erase or program pages that are marked as being bad. Also, the increased density of the NAND devices brings with it some compromises, such as a possibility of a spontaneous bit change. To ensure correct overall operation, a number of additional bits are set aside for error correct control (ECC).

A native NAND physical interface typically consists of the following signals:

- Chip Enable: a signal to select a device, set up by an address decoder.
- Write and Read: to indicate direction of access.
- Command Latch Enable: latch a command on the I/O pins.
- Address Latch Enable: latch the address.
- I/O 0–7 (0–15 for 16-bit devices): this is a multiplexed command, address, and data protocol.
- Write Protect: if active it prevents erase or writes to the flash device.
- Busy indication: used to indicate if the device can accept a command.

A NAND read operation consists of loading a page from the flash array and then reading the data from the page register. Figure 4.14 shows the representation of the page register and blocks.

The following steps are required to read from the device:

1. Issue a Read Command by placing 0x00h on I/O[0:7]. Bring command latch signal and write enable active.
2. Issue the address to read by placing each byte of the address on the I/O[0:7] bus, activating the address latch and write enable for each byte of address.
3. Issue the second cycle of the read command by placing 0x30h on the I/O[0:7] while activating the command latch and write enable signals.

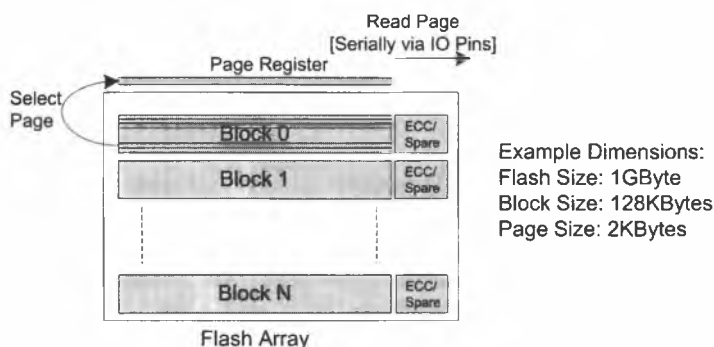


FIGURE 4.14

NAND Device Representation.

4. The busy line should now go active while the flash array is being read and places the contents in the page register.
5. Data can now be clocked out of the page register; by issuing subsequent Read Enable signals, the device places data on the I/O pins.

There are device optimizations to improve the performance, but this is the general process of accessing the device. As you can see, it really is block oriented.

In order to reduce the need for both NAND and NOR devices in a system, some NAND devices provide a special boot block. The boot block is designed to be as reliable as a traditional NOR device (using additional ECC or reduced density or an actual NOR partition). This boot page is automatically placed in the read page buffer, so that a processor can start random accesses to this page when the system first boots.

NAND Interfaces

When NAND devices were first introduced into the market, no standard interface was defined. The Open NAND Flash Interface (OnFi) industry workgroup was formed in 2006 to drive the standardization for raw flash devices. The OnFi workgroup has developed a number of specifications since 2006 (<http://www.onfi.org>):

- *ONFi 1.0* defines a standard electrical and protocol interface, including the base command set.
- *Block Abstracted NAND 1.0* defines a managed NAND solution that utilizes the raw NAND interface.
- *ONFi 2.0* defines a high-speed double data rate interface (synchronous) of up to 133 megatransfers/second.
- *NAND connector 1.0* defines a physical connector that supports a DIMM type form factor. This allows for upgrade of density/devices, and so on.
- *ONFI 2.1* increased bus speed to 200 megatransfers/second.
- *ONFI 2.2* includes interruptible read/erase to allow support for high-priority reads.

Some SOCs provided a “native NAND” interface. In many cases the SOC incorporates an NAND controller, and the NAND devices are connected to the SOC via an ONFi standard interface. In other

cases the interface (especially the asynchronous model) uses low level software to manage the flash transactions at a low level. In the simplest form (and probably the lowest performance) the interface can be driven by GPIO pins, as you can review in a Linux kernel driver (`linux/drivers/mtd/nand/gpio.c`).

NAND Controllers

As you can see, the interface to the NAND devices is complex enough that a NAND controller is required, especially if high performance is needed. The NAND controllers are usually within the SOC. A controller generally supports a DMA engine that takes a block read request and a pointer to the buffer. Then the controller issues the commands to the NAND device using the ONFi-defined mechanism, performs the NAND register read, places the results in the target memory buffer, and raises an interrupt to indicate the page has been read. An example of such a driver can be found in `linux/drivers/mtd/nand/denali.c`.

In many cases, the SOC does not include a NAND controller or direct support for ONFi standard NAND devices. In this case the SOC usually provides an alternative interface known as Multi-MediaCard (MMC). MMC memory usually takes the form of MMC cards. The definition is now standardized under the stewardship of the JEDEC organization (<http://www.jedec.org/>). The definition includes a flash memory controller and the flash memory devices. Of particular interest to embedded applications is the Embedded MMC (eMMC) specification (<http://www.jedec.org/standards-documents/docs/jesd84-a441>). In some cases eMMC devices can be soldered down onto the motherboard. The eMMC devices consist of a controller and memory. The controller offers a higher level application interface to the SOC. The driver for MMC devices can be found in `Linux/drivers/mmc`. The code is split into a card-specific folder, general host files, and files specific to the MMC controller. Secure Digital (SD) memory interface is the new flash interface standard based on MMC. The standard MMC/SD host controller interface is defined by the SD Association. Most controller covers operation with SD and MMC/eMMC devices/cards. The Linux kernel supports a device driver for the SDHC host controller in `linux/drivers/mmc/host/sdhc.c`. The driver provides access routines to file systems such as FAT32 on the device. The I/O hub device in the Intel Atom E600 Series design provides a standard-compliant SDHC controller.

Hard Disk Drives and Solid State Drives

Hard disks have been a traditional low-cost form of mass storage in platforms for many years. The hard disk drive is a device where data are stored on circular magnetically coated platters. This form of storage has the least cost per bit of all storage types considered. There are mechanical and environmental considerations that you must consider for these forms of mass storage device, and the environmental conditions associated with embedded devices often precludes their use. Because the media is revolving on a spindle, they are sometimes known as spindle-based drives. As the cost of NAND devices has come down, a new type of storage known as the solid state drive (SSD) has become cost-effective in many embedded cases. The SSD is a drive that is made up of NAND devices and a sophisticated NAND controller. The NAND controller presents a traditional hard disk drive interface to the host. The controller takes care of all the low-level flash management including wear leveling.

The latest drive interface is Serial Advanced Technology Attachment (SATA). It is defined by the Serial ATA International Organization (www.serialata.org). It is an interface used to connect host controller adaptors to mass storage devices. It is the natural evolution of the previous Parallel ATA

Table 4.5 Summary of NAND and NOR Flash and SSD

NAND Flash	NOR Flash	SSD
Lowest cost per bit	Higher cost per bit	Cost per bit similar to NAND
Slow random access	Random access	Faster than magnetic drives
Fast writes	Slower writes	Drive performance is based on NAND devices and controller algorithms
Fast erases	Slower erases	Drive performance is based on NAND devices and controller algorithms
Page writes only	Byte writes	Block-based device
Sustained sequential read performance similar to NOR	Sustained sequential or random read performance is similar	Drive performance is based on NAND devices and controller algorithms
Aligns with file system block semantics	Supports execute in place, similar to ROM	Supports standard file systems through traditional mass storage interfaces
Controller must perform some sanity checks before it can be used	Available instantly after power on	The controller must perform some NAND integrity checks before it is used

standard. In most platforms the host controller follows a standard register level interface known as AHCI. The AHCI standard is defined by Intel (<http://www.intel.com/technology/serialata/ahci.htm>).

Most controllers can be configured in legacy Parallel ATA mode or the more advanced native interface. The native high-performance mode should be used for SATA drives. This native interface is a hardware mechanism that allows software to communicate with Serial ATA devices. AHCI is a PCI class device that acts as a data movement engine between system memory and SATA devices. The host bus adaptor (controller) supports programmed I/O mode, DMA modes (which are akin to the original legacy modes on PATA devices), and a mode known as native command queuing for SATA devices. The vast majority of operating systems provide some level of support for SATA drives, especially since the controller supports the most basic of operating modes (PIO). The source for the SATA drivers can be found in `Linux/drivers/ata/*ahci*.c`.

Table 4.5 summarizes some of the relative attributes for each mass storage device.

DEVICE INTERFACE—HIGH PERFORMANCE

In many cases an SOC needs additional capabilities such as an external application-specific I/O device. The device interface consists of a device interface controller and a defined external interconnect to attach to an external device.

A peripheral bus interface requires a number of capabilities:

- Transaction mapping from the processor to the device address space—The device controller captures transactions directed to the memory address of the external bus and associated device, and translates the transaction (read or write) into the appropriate transaction on the expansion

bus. The translation may involve adaption due to bus width changes (such as 32-bit to 16-bit), or byte order (such as endian conversions).

- Inbound transactions—For high-performances interfaces, it is important that the expansion interface allow the external devices to read and write resources in the SOC. This is often called *bus mastering*. This is primarily used to access memory so the device can read and write data. Without the ability to read or write system memory, the processor or a DMA controller on the device would need to move data to and from the device. When transactions are generated by the device the transaction contains an address, the address space may be different from that used by the processor; you should be aware of the system address map as viewed by the device. The address on the device interconnect is usually known as the bus address. Your software must translate the host system address to bus address if you want the DMA engine on the device to read the system address. The device interface block in the SOC will translate the inbound bus address to the system address.
- Interrupts—Interactions with a high-performance device require the ability to route interrupts from the device to the processor. This can be used to simply indicate events on the I/O device, but most frequently it is used to send service events from the DMA engine on the device. The interface ideally includes assignment and routing of interrupts as an integral part of the capability. In simple bus interfaces the interrupts are usually routed as signals on the side, in this case general-purpose I/O signals that are configured to generate interrupts.
- Physical standard—Some interface definitions also include a physical connector standard, such as card size. A physical standard allows the system to be expanded with many different capabilities. The Peripheral Component Interconnect (PCI) is an excellent example of this. There are other expansion connector standards that cover much more than a device interconnect such as ComExpress (<http://www.pcmg.org>) (although this standard includes a PCI bus among others as part of the interconnect). In some cases, the interface supports removable/reinsertion of devices while the system is active. This is usually known as *hot plug* and can be key to maintaining high availability on platforms.

There are many different bus definitions, from well-standardized Peripheral Component Interconnect (PCI) to de facto standard parallel address and data bus interfaces.

Peripheral Component Interconnect (PCI)

The predominant high-performance external interface standard is PCI. PCI has been on Intel platforms for many years and has evolved from a 32-bit parallel bus to a high-speed lower-pin-count serial bus (with many lanes to scale performance). Although the PCI standard has significantly increased I/O performance with every PCI generation, the logical view of the PCI system has been kept very consistent throughout the evolution of the standard. The approach to discover and access the device has remained the same, allowing full software compatibility over PCI generations.

PCI Express (PCIe) is the third generation of the PCI standard. It is a higher-performance I/O bus used to connect peripheral devices to compute systems. PCIe is the primary peripherals interconnect used on Intel architecture systems, but it has become popular on many platforms. The PCIe standard is developed by the PCI-SIG (<http://www.pcisig.com>).

The PCIe standard provides the capability to identify devices attached to a PCIe bus at runtime. Once these devices are discovered, the system software can allocate resources to these devices

dynamically. This process is called *bus enumeration*. The enumeration process allocates a range of addresses in the system address map to access the device, as well as one or more interrupt vectors. To facilitate the enumeration process, there is a special PCIe bus space known as configuration space. The PCIe configuration space size is a total of 256 MB. The bus supports device discovery and initial configuration by responding to special configuration space transactions on the bus. The configuration space is partitioned into PCIe busses (up to 256), devices per bus (up to 32), and functions within a device (up to 8 per device). This three tuple address is known as the Bus Device Function (BDF) address. The BDF address for a device depends on where it is connected to on the PCIe fabric. The function is dedicated by the design of the device. For example, if a Ethernet card has two separate Ethernet interfaces, it will mostly likely present two separate functions. The process of enumeration is a runtime search through the PCI configuration space looking for devices. Figure 4.15 shows the physical topology of a PCIe system.

The head of the PCIe hierarchy is known as the root complex. The root complex takes processor read/write transactions and routes them to the appropriate transaction type in the PCIe fabric. The root fabric takes upstream transactions from PCIe devices and routes them to the host memory. It also routes interrupts from the devices to the appropriate processor. Devices can also be integrated into the root complex; in this case the devices are presented in exactly the same way as devices that are attached via external physical PCIe links. This provides a consistent approach to discovering internal and external devices.

Each device has a configuration space header. Each device configuration space can be up to 256 bytes. A device can optionally have an extended capabilities register at offsets in range 256 to 4095. A total of 65,536 devices are theoretically supported on PCI. Processors cannot natively generate a PCI configuration cycle; the PCI configuration space transactions can be generated by either of the following mechanisms (on Intel architecture systems):

- *Indirect mode*: Use a sequence of I/O transaction to write to address port at I/O address 0xCF8 followed by read/write to Data Port register at I/O address 0xCFC. The address register is written with the PCI configuration address to read, and then content is read/written to by accessing the data port register. This mechanism allows accessing of first 256 configuration registers; extended capability registers in range [256–4095] can be assessed only through the MMIO mechanism described next.
- *MMIO*: Up to 256-MB (max 4K per device * 65,536 devices) configuration space is mapped into the processor's memory-mapped address space. Reads or writes to this address space generate the configuration transactions on the PCI bus.

The enhanced MMIO mode of configuration can directly access all registers of any device on the PCI bus. The MMIO address bits are built up from the following bit fields:

- A[32:28] = MMIO configuration space: base of which is set by the MMBAR register
- A[27:20] = PCI bus number
- A[19:15] = Device number on bus
- A[14:12] = Device function number (up to 8)
- A[11:8] = Extended register
- A[7:2] = Register number
- A[1:0] = Byte select for a register

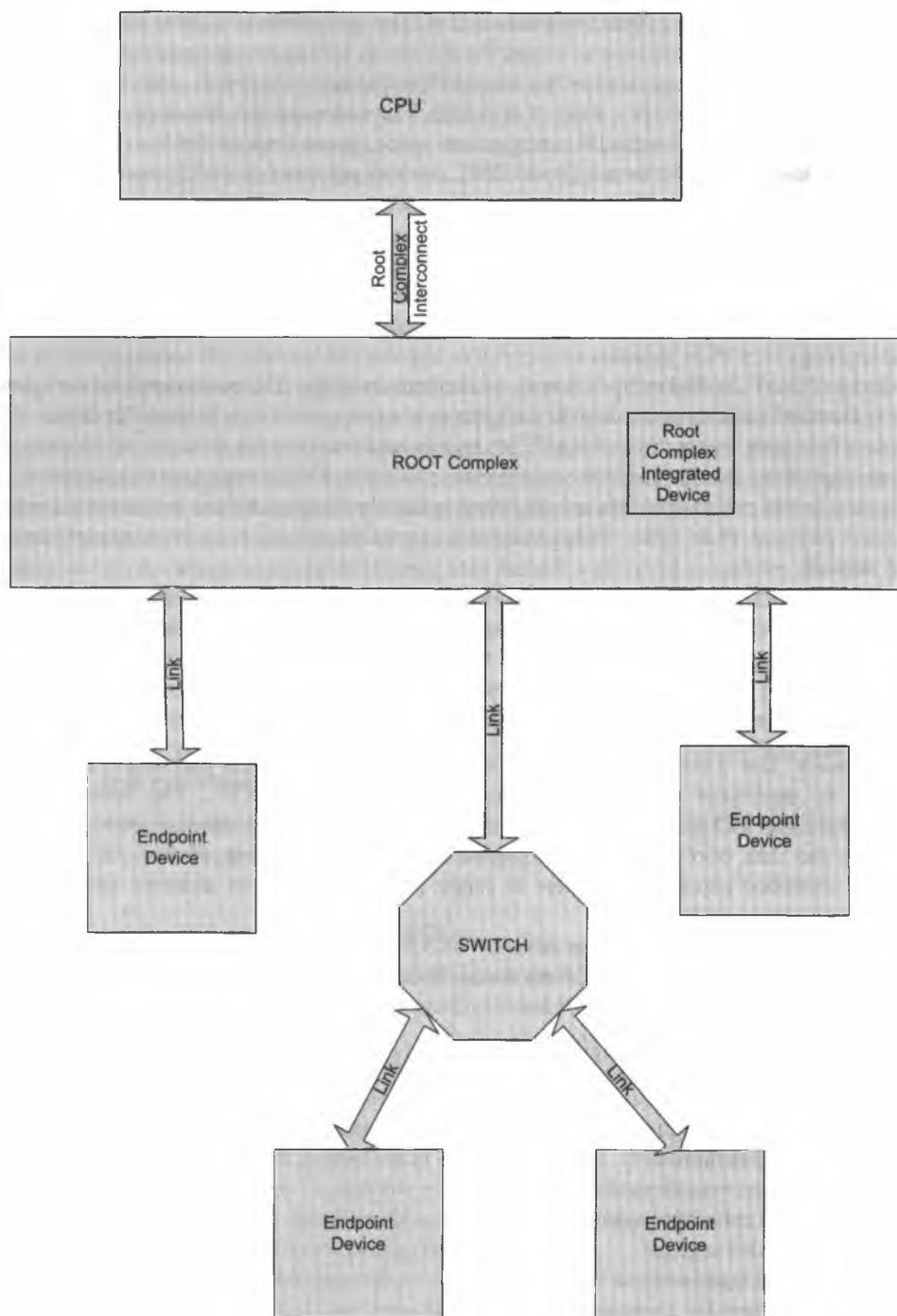


FIGURE 4.15

PCIe Physical Hierarchy.

Each device must have a base configuration header. Figure 4.16 shows a base configuration header. It allows the system software to identify the device type and to assign memory and interrupts to the device.

The vendor ID identifies the manufacturer of the function. The IDs are assigned by the PCI-SIG. The vendor ID assigned to Intel devices is, somewhat humorously, 8086h. The device ID is assigned by the manufacturer and is used to indicate the device type. This device ID is usually used by operating systems to find the appropriate device driver to load and associate with the device. The device revision can be used to adapt the driver for specific versions of the device. The Class code can for some generic devices be used to provide a generic device driver support for a device. In this case the device ID is ignored and the operating system assigns the device driver based on class code. An example where a class code for a generic device is used is a simple VGA adaptor.

PCI devices are composed of two different register spaces, the configuration space and device registers. The configuration space is primarily used as part of the enumeration and allocating the driver to the device. The configuration space contains a number of base address registers. These base address registers are programmable decoders that are used to decode access to the actual device registers.

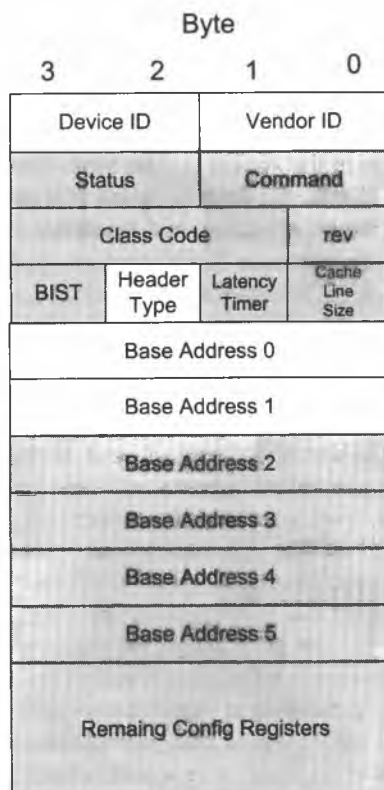


FIGURE 4.16

PCI Configuration Header.

There are six 32-bit or three 64-bit base address registers. The base address register contains some read-only low-order bits set to zero. When you write all ones to the base address register and read back the value, the enumeration software establishes the size of memory the enumeration software must allocate. For example, if the software writes 0xFFFFFFFF to a base address register but reads back 0xFFFF000, then the device is requesting a space of 4 kB. The enumeration software allocates all the memory for the devices “requested” by the enumeration process and writes to all the device base address registers. The lower byte of the base address register has some special meaning; it indicates whether the base address is to allocate memory or I/O space, although the vast majority of devices now limit the registers to memory-mapped only. It also includes an prefetchability attribute—it indicates if the memory space for the device is prefetchable. This is an important classification for I/O devices. Registers on a device must not be prefetchable of a speculative read by the processor; this has a side effect on the hardware. Consider a processor speculative read of a hardware FIFO. In this case the value is read, but it may not be used by the software: the data are lost. This bit has no special significance for Intel hardware since the Intel CPU does not do speculative prefetch to MMIO space, which is normally mapped into the uncachable region.

When the base address register is written, all memory transactions generated to that bus address range are claimed by the device. The addresses are assigned to be in noncacheable memory-mapped I/O space by the bus enumeration software at startup. The registers allocated via the bar address registers are specific to the device and not standardized in any way by the PCI-SIG.

The addresses used on the bus are known as bus addresses. On Intel platforms there is a one-to-one correspondence between the host physical address and the address used on the bus. There is no translation of the outbound address in the system (unless virtualization is being used in the system).

In Linux the `lspci` command details the devices for all PCI devices in the system. It shows the device types, the memory and interrupts allocated, and the device driver attached to the device.

```
>lspci -v (sample of results)
00:03.0 Multimedia video controller:
  Intel Corporation Device 8182 (rev 01)
  Subsystem: Intel Corporation Device 8186
  Flags:bus master, fast devsel, latency 0, IRQ 11
Memory at d0300000
  (32-bit, non-prefetchable) [size=512K]
  I/O ports at f000 [size=8]
Memory at b0000000
  (32-bit, non-prefetchable) [size=256M]
Memory at d0380000
  (32-bit, non-prefetchable) [size=256K]
02:0a.2 Serial controller:
  Intel Corporation Device 8812
  (prog-if 02 [16550])
  Flags:bus master, fast devsel, latency 0, IRQ 19
  I/O ports at e040 [size=8]
Memory at d0149000
  (32-bit, non-prefetchable) [size=16]
Kernel driver in use: serial
02:00.1 Ethernet controller:
```

```

Intel Corporation Device 8802
Flags:bus master, fast devsel, latency 0, IRQ 25
Memory at d0158000
      (32-bit, non-prefetchable) [size=512]
Kernel driver in use: ioh_gbe

```

The interrupts vectors are also assigned during the enumeration process. Each PCIe device provides configuration registers to support a feature known as MSI/MSI-X. There are two key register types: the first is an address register and the second is a data register. The address is a special address in the system, and it is the targeted interrupt controller for which the interrupt is being sent to. Each processor hardware thread has an individual target address, so MSI/MSI-X interrupts can be steered to a specific hardware processor thread (mechanism for any processor to handle the thread are also defined). The data register is the vector for the interrupt that is assigned to the device. The device can request multiple interrupt vectors, and the enumeration process attempts to satisfy all device requests with individual interrupt vectors. In cases where there is oversubscription of interrupt vectors versus device requirements, (i.e. the system cannot allocate all the vectors requested by the device), some vectors must be shared among different functions within a device. Having many separate vectors for a device can improve the performance of an interrupt handling routing, as it does not have to read device status registers to find out the cause of the interrupt.

Not all operating systems support MSI-X; fallback legacy interrupt support mechanisms provide emulated wired interrupts.

Bus Mastering

The ability for a PCI device to read and write payloads (descriptions of work, data to be transmitted, and so on) is very important, especially for high-performance interfaces or devices. The ability for a device to read or write host memory is known as *bus mastering*. A device can generate read and write transactions on the PCIe bus. These transactions are routed to the root complex and then to the system memory controller. The bus address used in an Intel system corresponds to host physical addresses in the system (unless virtualization is being used). The device has access to the entire physical memory space of the host. It may directly access memory with high performance. A common PCI communication pattern is to provide circular buffers of work to send to a device in host memory. The device contains head and tail pointers for the host memory resident circular buffer, and these pointers are mapped via the base address registers. The circular buffer contains the description of the work requested of the device (for example, “transmit the packet pointed to by the descriptor”). Let’s discuss the case of a network interface card. The software first sets up the descriptor with a pointer to the packet to be transmitted. Then the tail pointer is updated in the device. This indicates to the device that there is something to do. The device reads the descriptor and finds a pointer to a packet to be transmitted. The device then directly accesses the packet from host memory to internal FIFOs for transmission on the line. A similar process occurs in reverse for reception of packets. The device driver can also update the circular buffer multiple times before writing the tail pointer to send a batch of work to the device. The pointers written to the device (such as the pointer to packet) must all be specified as PCIe bus addresses. In the case of Intel architecture systems this is the same as physical addresses, but not all systems are like that so you should use bus to physical address conversion routines provided by your operating system. If you are using virtual memory, you must convert from virtual to bus address when populating any address in a device that it will subsequently access directly.

Transaction Ordering

When a device has the capability of directly accessing host memory, it becomes very important to understand the memory transaction ordering rules for these host memory accesses. Consider a device that has received a packet from the network. The device issues a number of memory writes to update the packet in memory and raises an interrupt to the processor. Depending on the bus and SOC topology, the interrupts may follow a different path than the memory writes. On PCI there are very well-defined rules for transaction ordering. The interrupt indication (when MSI and MSI-X is used) follows the same logical ordered path as the memory writes, and the interrupt cannot pass the memory writes until they are globally visible to the processor. That means that when the driver eventually receives the interrupt you can be sure that the memory has been updated with the correct values from the device (or if not actually in memory, the right data values are returned if the processor performs a read of the address). On many SOC devices this is not as rigorously defined, and ordering of transactions and interrupts is loosely defined. In such cases, the software may operate on inconsistent memory contents partially updated by the device; in many cases, reading the device ensures that the memory is fully up to date.

UNIVERSAL SERIAL BUS

The Universal Serial Bus (USB) is a low-voltage differential pair serial bus that was designed to connect computer peripherals to PC platforms. The USB is standardized by the USB Implementers Forum, Inc. (www.usb.org). It has been designed to support real-time data transfer of video, audio, and data. The USB is composed of a tiered star physical topology. At the top of the hierarchy is the host root hub. There is a single USB host in the USB hierarchy; the USB host contains the USB host controller. In some cases, the host controller has an integrated hub to expand the number of connectors on the host controller. There are two types of USB device, hubs and functions. Hubs provide additional fan-out for the bus, and functions provide a capability such as a mass storage or mouse.

Each physical link consists of four wires. A pair of differential signal wires, D+ and D-, are used to carry the data to and from the device. The wires are effectively half duplex. In addition to the data wires, there is a 5 V power supply and a ground pin. The data is carried using an NRZI line encoding, and a special sync sequence is sent at the start of all the packets to allow the receiver to recover a clock. This scheme removes the need for a separate clock wire.

As the USB standard has grown in its use, the trend has been to miniaturize the USB connector standard, from the traditional PC/laptop USB connector to the micro USB connector found on many cell phones.

The USB bus is hot-pluggable, which means that the device can be added or removed from the system without any electrical considerations. When a device is inserted into the platform, a change in the electrical characteristics is detected by the host. This triggers the discovery process.

As we mentioned, the USB provides a 5 V power supply. It is provided for use by the peripherals being attached to the platform. The current supply is relatively low, from 100 mA to 500 mA. This is adequate for peripherals such as mice, touch screens, and pen drives, but not for a peripheral such as a printer.

Different speeds have been standardized; these allow for quite significant data transfer capabilities over the bus:

- Low speed: 1.5 Mbps. This is standardized under the USB 1.0/1.1 specification, and included in the USB 2.0.
- Full speed: 12.0 Mbps. USB 1.0/1.1 and 2.0.
- High speed: 480 Mbps. This data rate is only available under the USB 2.0 standard.

The speed of the bus should not be confused with sustainable throughput of the bus after all the bus and protocol overheads have been taken into account. For example, on a bulk endpoint there is a maximum payload size of 512 bytes. A reasonable sustained user payload throughput of about 400 Mb/s or 50 MB/s is available for a high-speed device when overheads are included. Actual devices may be lower due to its own hardware limitations.

Compatibility is a key feature of the USB standard; a USB device compliant with an older version of the standard can be inserted into a platform supporting a newer standard and operate at its originally supported speed. The bus limits the transactions seen by a device to ensure this compatibility: high-speed transactions are only distributed to high-speed devices, full-speed transactions are distributed to full-speed devices and low-speed transactions are distributed to full- and low-speed devices.

There are two sides to a USB link; one is known as the host side and the other as the device side. The host port is the primary driver of the USB bus. There can only be one host port in a USB hierarchy, as shown in Figure 4.17. There can be up to 127 devices on a Universal Serial Bus. This fan-out is supported by a hierarchy of simple hub devices.

There are many cases where an embedded system may want to behave as a USB device in one use case and as a USB host in another use case. For example, when a mobile phone is connected to a PC to

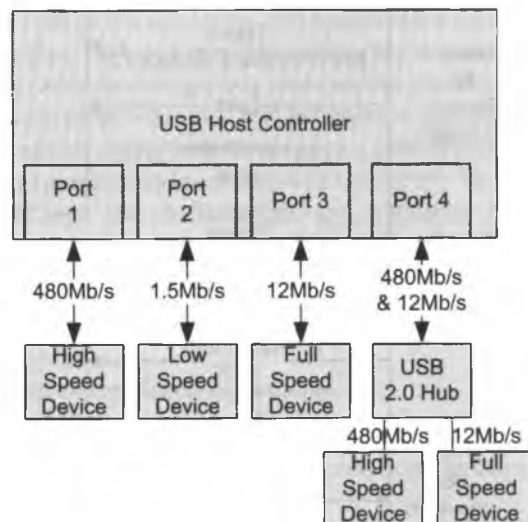


FIGURE 4.17

USB Hierarchy Examples.

change and synchronize the contacts, the PC acts as the USB host and the mobile phone acts as a USB device. However, when the phone is connected to a peripheral such as a printer or audio speakers via USB, the mobile phone now acts as a USB host and the peripheral attached acts as a USB device. A good example of this can be found in the current Android-based handsets that support the Android@ Home APIs. In order to support this, a mode known as USB On-The-Go (USB OTG) has been developed. It is a supplement to the USB 2.0 specification. A USB controller that supports USB OTG can act as a device or host. This also allows you to connect two devices of the same type, where one becomes the host and the other becomes the device; the roles are defined by the cable.

Any time a device is inserted into the bus it is detected in real time, or the host is powered on and a bus enumeration occurs. This allows the host software to assign each device in the USB hierarchy a device address. The bus protocol itself is robust with a built-in error handling/fault recovery mechanism.

The USB specification outlines the data flow between host and devices, as well as scheduling of traffic to and from the devices, as illustrated in Figure 4.18. The protocol supports isochronous and bulk data transfers across the bus.

The USB is a polled bus; the host controller initiates all data transfers between the host and devices.

Most bus transactions consist of three separate packets transmitted on the bus. The host controller starts by generating a token packet; this packet indicates the type and direction of the transaction. It also contains a device and endpoint address. The endpoint is a logical channel identifier at the device. There can be up to 15 endpoints within a device. After the token has been received by the device, the data transaction is generated either by the host or by the device depending on the direction specified. Once the data transaction is complete, the handshake packet is generated. The ACK packet is

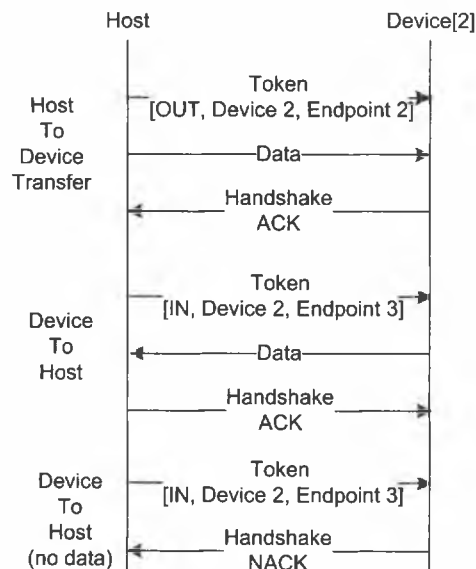


FIGURE 4.18

Transaction Phases for USB Transfer to and from a Device.

generated by the device of the data flow from the host to device (OUT transfer) or by the host to indicate that the host has received an error-free data packet from the device. IN and OUT are defined from the host controller view; an OUT token indicates that data will be sent from the host to the device, and an IN token solicits data from the device to the host.

The USB supports data and control exchange among the USB host and devices across a set of logical pipes. A pipe is a logical association between a logical endpoint and the host device. These pipes can be unidirectional or bidirectional. Pipes are used to present a logical connection between a software element running on the host and an endpoint in a device. From the software view this is a direct connection and hides the details of the bus hierarchy.

In order to support many different device types and use cases on the USB, four fundamental types of data transfer have been defined:

- *Control Transfers.* These are used to configure the device when it is initially discovered in on the bus. Control transfers are usually used to set up the other endpoints (e.g., a bulk endpoint) on the device.
- *Bulk Data Transfers.* These are used for large-scale transfer of data to and from the device. They are used when no special latency constraints exist in the transfers to or from the device. The data exchange is reliable. The bandwidth occupied by bulk transfers can vary depending on the other bus activities; these transfers are the lowest priority.
- *Interrupt Data Transfers.* These are used for timely delivery of data to and from the device. These can be used for events such as mouse movements, or they can be used from a device that wishes to indicate that data are available for a bulk transfer (avoiding constant polling of the a bulk endpoint).
- *Isochronous Data Transfers.* These transfers are allocated a guaranteed bandwidth and delivery latency constraint. They are continuous and occur in real-time. Dedicated bandwidth is set aside on the bus for isochronous data. They are usually used for real-time transfers of audio and video.

Transfer types cannot be mixed on the same pipe. The bandwidth is allocated among pipes. Depending on the configuration of the pipe, bulk data transfers can occupy any remaining unassigned bandwidth.

Devices are assigned an address during a bus enumeration process. This process occurs when the host controller is first initialized or when devices are inserted (hot-plugged). All devices support at least one endpoint: endpoint zero. Endpoint zero contains a control pipe. The control pipe on endpoint zero provides a standardized mechanism to interact with the device. The control endpoint must support standard requests: `set_address`, `set_configuration`, `get_descriptor` for device and configuration descriptors. The standard information that can be obtained for all USB devices is

- Vendor identification
- Device class
- Power management capabilities
- Endpoint description with configuration information

In addition to this standard information, further information that is specific to the class can be provided. After that, vendors can provide additional information, but there is no standardization of this information.

The information provided by the control endpoint is provided as a series of descriptors. The command below shows a subset of the data provided by the Linux `lsusb` command. It shows the device, configuration, interface, and endpoint descriptors for a USB pen drive.

```

>lsusb -v
Bus 001 Device 002: ID0781:5406 SanDisk Corp. Cruzer Micro U3
Device Descriptor:
  bcdUSB             2.00
  bDeviceClass       0 (Defined at Interface level)
  bDeviceSubClass    0
  bDeviceProtocol    0
  bMaxPacketSize0    64
  idVendor           0x0781 SanDisk Corp.
  idProduct          0x5406 Cruzer Micro U3
Configuration Descriptor:
  bmAttributes       0x80   (Bus Powered)
  MaxPower           200mA
Interface Descriptor:
  NumEndpoints       2
  InterfaceClass      8 Mass Storage
  InterfaceSubClass   6 SCSI
  InterfaceProtocol   80 Bulk (Zip)
Endpoint Descriptor:
  EndpointAddress    0x81 EP 1 IN
  Transfer Type      Bulk
  Synch Type         None
  Usage Type         Data
  MaxPacketSize      0x0200 1x 512 bytes
Endpoint Descriptor:
  EndpointAddress    0x01 EP 1 OUT
  Transfer Type      Bulk
  Synch Type         None
  Usage Type         Data
  MaxPacketSize      0x0200 1x 512 bytes

```

The pen drive is one of the simplest USB devices; it only used two bulk endpoints. A similar command issued for an Apple iPhone™ 4 shows far more extensive use of the USB capabilities. Table 4.6

Table 4.6 Apple iPhone 4 Configuration and Endpoints

Configuration	Endpoints/Type	Usage
PTP	2 bulk endpoints (IN /OUT) 1 interrupt endpoint	Still image capture using Picture Transfer Protocol
iPod USB Interface	1 isochronous (IN) 1 interrupt	Audio control - Streaming audio - HID
PTP + Apple Mobile Device	2 bulk + interrupt 2 bulk (IN/OUT)	PTP Vendor-specified protocol
PTP + Apple Mobile Device + Apple USB Ethernet	2 bulk + interrupt 2 bulk (IN/OUT) 2 bulk (IN/OUT)	PTP Vendor-specified protocol Vendor-specified protocol

is a summary of the endpoints on such a device as listed by `lsusb`. There are four different configurations supported with the device we queried. Only one of these configurations is active at a time.

For a particular configuration, a device's descriptors contain a list of interface classes (and subclasses). The interface classes are mapped to endpoints with the required capabilities. The interface classes have been standardized for a number of common use cases. If an interface conforms to the standard, then the device does not require a special device driver to operate with the host—it can in most cases rely on the OS class driver for the capability. If the device does not support the standard class, the platform will require a special driver to interact with the device. The following is a list of classes that are supported:

- Bluetooth
- Chip/smart card interface
- Hub
- Human interface device
- Mass storage
- Printing
- Scanning/imaging (PTP)
- Media transfer (MTP)
- USB audio
- Modem
- Video

The USB organization standardizes the class specifications. The standard covers items such as the required descriptors and endpoints, and, most importantly, it specifies the format for the payloads that go across the endpoints.

Many concurrent users of the bus are carried over the various pipes between each of the devices and the host controller. Each of the pipes has specific bandwidth requirements and latency bounds. The host controller controls the allocation of bandwidth across the bus. It manages when the host and each individual pipe are allowed to transmit data. The maximum size of a packet that a device can transmit or receive is provided in the endpoint descriptor. The USB bandwidth is allocated on the bases of 1-ms frames, with microframes of 125 μ s. Each microframe can contain multiple USB transactions. The USB host controller software schedules transactions to be performed during each frame by building up a list of transfer descriptors. Linked lists of transfer descriptors called a frame list are fetched by the host controller for each USB frame. The host controller makes sure that that the microframes are packed with transactions, but no transaction can cross the microframe boundary.

Programming Interface

The programming interface for USB host controllers has been standardized. These standards define the register maps for the controller and the descriptor formats for the data when they are transmitted and received on the bus. Three standards are in existence:

- UHCI – Universal Host Controller Interface standardized by Intel and used for USB 1.x controllers.
- OHCI – Open Host Controller Interface for USB 1.x. This standard was developed by Compaq, Microsoft, and National Semiconductor (ftp://ftp.compaq.com/pub/supportinformation/papers/hcir1_0a.pdf). The OHCI specification expects more capabilities in the hardware over that expected by a UHCI controller.

- EHCI – Extended Host Controller Interface. This is used for USB 2.0 device controllers on Intel platforms (<http://www.intel.com/technology/usb/ehcispec.htm>).
- USB OTG – As we mentioned, there is no standard host controller interface standard for the device model.

Most devices support both 1.1 and 2.0 devices by instantiating both a UHCI and EHCI controller. This is often done for expedience; the two controllers can coexist and a multiplexing function can be placed between the devices and each controller.

Depending on the source of the IP in the embedded SOC device, the register map and descriptor format may not follow these standards.

Figure 4.19 shows the components of a software stack on top of the EHCI controller.

The EHCI host controller manages the transmission and reception of frames on the bus. The Enhanced USB host controller contains two sets of software accessible hardware registers, memory-mapped registers, and optional PCI configuration registers. The PCI configuration registers are required if the host controller is implemented as a PCI device. The PCI registers contain the PCI header and some additional configuration registers required to allocate the appropriate system resources during enumeration. The PCI registers are not used by the USB driver after enumeration.

The memory-mapped registers are divided into two sections:

- *Capability registers* are a set of read-only capability registers. The capability registers specify the limits, restrictions, and capabilities of a host controller implementation. These values are used as parameters to the host controller driver.

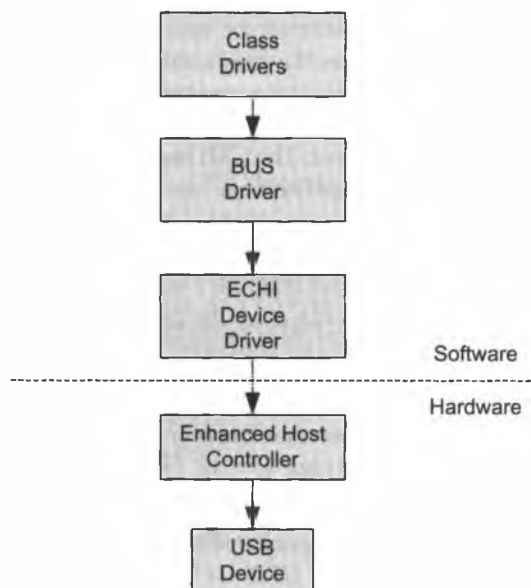


FIGURE 4.19

Software Stack on ECHI Controller.

- *Operational registers* (which are read/write). The operational registers are used by system software to interact with the operational state of the host controller. The operation registers contain pointers to lists in host memory that are used to define the traffic to be transmitted and received.

Figure 4.20 shows the register structure for a PCI EHCI controller.

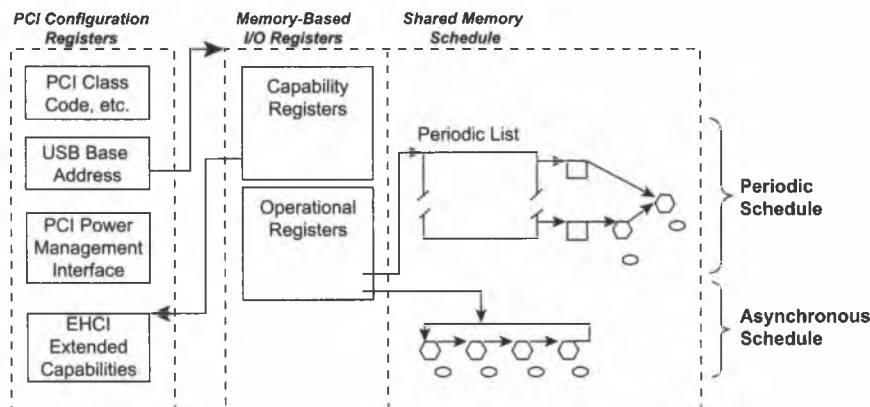


FIGURE 4.20

EHCI Register Classification.

Source: <http://www.intel.com/technology/usb/download/ehci-r10.pdf>

The EHCI provides support for two categories of transfer types: asynchronous and periodic. Periodic transfer types include both isochronous and interrupt. Asynchronous transfer types include control and bulk. Figure 4.20 illustrates that the EHCI schedule interface provides separate schedules for each category of transfer type. The periodic schedule is based on a time-oriented frame list that represents a sliding window of time of host controller work items. All isochronous and interrupt transfers are serviced via the periodic schedule. The asynchronous schedule is a simple circular list of schedule work items that provides a round-robin service opportunity for all asynchronous transfers.

Note

Wake on USB: The EHCI specification often describes the physical location of the registers within the device. Different parts of the EHCI controller are powered by different supply voltages within the SOC. As a result, some sets of registers retain their values during different CPU power states. These features are provided to support wakeup events from a USB device. Consider the case where your laptop is in a sleep state (suspended); moving the mouse can trigger a wakeup event to the platform, but in order to do so the EHCI controller must remain powered.

The EHCI controller generates interrupts to the processor based on a number of events. One such interrupt is generated at a point in the processing of the asynchronous lists. This is configured when the list is generated by the driver to provide enough advanced notice for the driver to add more work to the

list. Another interrupt is generated when the period frame list rolls over; this again is to ensure that the driver supplies data for subsequent frames.

As we mentioned, there are two key lists managed by the driver and consumed by the host controller: the periodic frame list (for isochronous and interrupt traffic) and the asynchronous list (control and bulk transfers). We'll discuss the asynchronous list first. The list is only processed when the host controller reaches the end of the periodic list, the periodic list is disabled, or the periodic list is empty. That is, it has the lowest priority of all traffic and will only be sent on the bus when there is no other work to do. The asynchronous list is pointed to by a control register: AsyncListAddress. The list is a circular list of queue heads. The controller cycles through each element in the list (in a round-robin fashion), processing each queue head as it goes. One queue head is used to manage the data stream for one endpoint. The queue head consists of a queue element transfer descriptor. The queue head structure contains static endpoint characteristics and capabilities. It also contains a working area from where individual bus transactions for an endpoint are executed. Each queue element transfer descriptor represents one or more bus transactions, which is defined in the context of this specification as a transfer. A descriptor has an array of buffer pointers, which is used to reference the data buffer for a transfer. Figure 4.21 shows how the elements described above are linked together.

The USB UHCI device driver provides services to the individual class drivers, which make use of the bus drive to transmit and receive data.

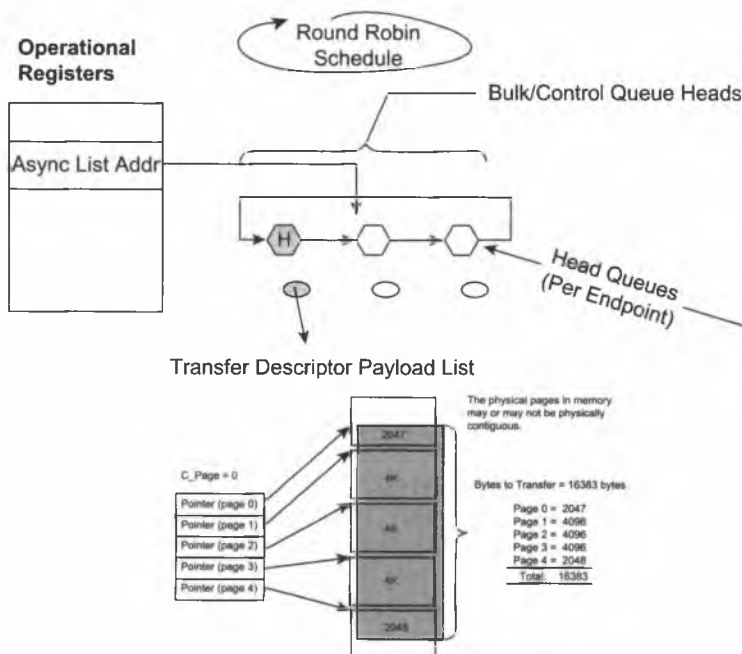


FIGURE 4.21

Async List Address, Head Queues, and Payload Description.

Source: EHCI Intel Spec.

Linux Driver

- The Linux USB driver is split into a core and set of class drivers. The core driver provides capabilities to read and write to an endpoint (among other things). The core driver provides a very simple mechanism to access the bus. The steps are as follows (<http://lxr.linux.no/linux+v2.6.35.7/drivers/usb/usb-skeleton.c>):
- Allocate a USB Request Block (URB) using `usb_alloc_urb()`.
- Allocate a DMA buffer data.
- Copy data into the allocated DMA buffer.
- Initialize the URB with device, endpoint, and data buffer pointers, and a function pointer to a callback routine using the call `usb_fill_bulk_urb()`.
- Submit the URB using `usb_submit_urb()`. The interface to the USB host controller is asynchronous, which means that the call `usb_submit_urb()` returns straightaway and does not block to caller.
- The USB driver will call the callback function associated with the URB once the transmission of the data associated with the URB is complete.

That describes the basic mechanism of sending data on the bus, but it is agnostic as to the payload being sent or received. The USB class driver for the device actually formats the payload that sent on the endpoint. For example, a USB pen drive uses the USB mass storage class device driver. Reference source code for the USB mass storage driver can be found at [Linux/drivers/usb/storage/](#).

DEVICE INTERCONNECT—LOW PERFORMANCE

There is a wide range of low-speed, low-pin-count, low-performance interface types. These are used to control sensors, flash devices, simple LCD displays, and the like. In this section we discuss the following interface types:

- Inter-Integrated Circuit (I²C) bus
- System Management Bus (SMB)
- Serial Peripheral Interface (SPI) bus
- Inter IC Sound (I²S)
- Universal Asynchronous Receiver/Transmitter (UART)
- High-Speed Serial

Inter-Integrated Circuit Bus

One of the most prevalent of these buses is known as the Inter-Integrated Circuit bus, or I²C bus. It was invented by Philips (now NXP) in the 1980s. The specification can be found at http://www.nxp.com/documents/user_manual/UM10204.pdf.

The bus standard allows for multiple masters on the bus (however, in many use cases it runs as a single master with multiple slaves attached). It is a simple two-wire bus consisting of a serial clock line (SCL) and a serial data line (SDA). Figure 4.22 shows the interconnection between devices on an I²C bus.

Each wire (SDA and SCL) is bidirectional. They are driven by open collector gates on each of the devices. Open collector gates allow any device to bring a pin to zero. You may have noticed the pull-up

USB memory stick is inserted, power is turned on, and ... nothing. No flashing LEDs, no indicators of any kind. The team was expecting a celebration, but got nothing but inert disappointment. What's wrong? Is the problem serious or trivial? How will you know?

Of course, in this fictional scenario and in every other real-world platform debug effort, there are an extraordinary number of possible causes. Was the power supply plugged in? Was memory inserted and properly seated? Is some component configured for debugging rather than operation? Is there a short on the circuit board? The fully enumerated list would be a very long one, indeed. Rather than working with a flat checklist, however, we will follow a process, one that is designed to systematically identify what works and what is faulty or misconfigured.

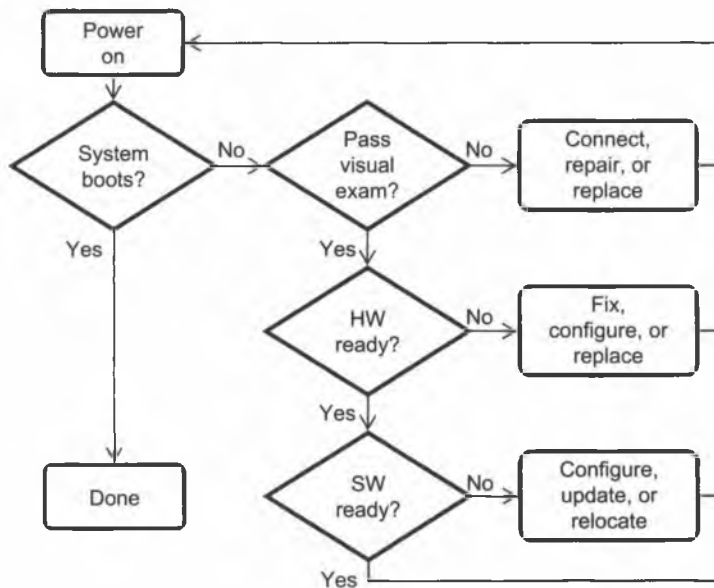
Before jumping in, it will be constructive to consider your mindset and perspective as you debug. Debugging is an open-ended activity, in which neither the ultimate explanation nor the time that will be needed to find it is known in advance. What is certain, however, is that you must ultimately find the problem. Given this, for most it is highly productive to adopt a positive, inquisitive mindset as you begin debugging. Not only will this make you feel better as you work, but it will make you better at what you are trying to do. Debugging, whether at the platform level or the application level, requires a careful, systematic effort to reduce possible causes until the problem is identified. Debugging in frustration or in haste is not conducive to careful, systematic work. So, before you start debugging, make sure you begin with the right frame of mind: be calm, systematic, and curious.

A PROCESS FOR DEBUGGING A NEW PLATFORM

The modify-compile-debug pattern so familiar to programmers is of little use when your platform will not boot. This traditional practice is a productive one, because with each change there is a high probability that any problems observed in the newly compiled program will be related to the most recent changes. When a system will not boot, while there is a possibility that only a small number of changes have been made that help to locate the source of the failure (that is, "all we changed was the power supply!"), but it is more likely that trouble with a brand new embedded system cannot *a priori* be confined to a small set of locations.

To tackle the search systematically, you can rely upon a structure process like the one presented in this chapter. The process can be characterized in three high-level steps:

1. Visual examination. Look at the system. Verify that external cords, cables, and I/O devices are attached. Remove the cover and examine the circuit board for damage and properly seated components.
2. Hardware evaluation. Are the hardware components, such as power supplies, voltage regulators, and integrated circuits, in their proper state and operating normally? As we will see, this step is subdivided into a number of specific, ordered substeps to uncover possible hardware faults or misconfigurations.
3. Software evaluation. Are the BIOS and OS kernel properly configured and able to boot? Are you aware of all known bugs from the most recent data sheet? Once the hardware has been determined to be operating properly, the next step is to verify that the systems software is properly configured and is being invoked correctly. This step, too, is subdivided into further substeps.

**FIGURE 17.1**

Platform Debug Process.

These three steps are illustrated in an overall platform debug flowchart in Figure 17.1. We will soon discuss these debug steps in detail, but before doing so we must first review some tools and chipset features essential to platform debug.

DEBUG TOOLS AND CHIPSET FEATURES

Debugging at the platform level requires inspection of system state that is not typically visible to the user or programmer. Consequently, a variety of tools may be needed in the course of a debugging session. In this section, we will discuss the classes of tools that are typically employed in the platform debug process. These include both hardware probes and analyzers and software tools.

Additionally, we will discuss some processor and chipset features that provide access to internal chip state for the purposes of debug and test that would otherwise be inaccessible.

Observing the output characteristics of hardware devices implies the measurement of fundamental electrical characteristics such as voltage and current. While voltmeters and multimeters can offer such measurements, it is typically necessary to observe these characteristics as they change through time.

To do this, hardware probes are required. Such probes come in several varieties and device specializations. Regardless of the power levels and time domains in which your system operates, hardware probes have been produced to meet your needs. Often, however, extremes of both very high and very low power and frequency come at a steep price. For most classes of embedded systems,

however, reasonably affordable, USB-based measurement devices can be obtained in the cost range of hundreds of dollars (USD). At the high end, some probes can cost hundreds of thousands of dollars. Most university and digital design laboratories will be stocked with these devices.

Oscilloscopes

Oscilloscopes are a ubiquitous form of hardware probe that primarily provides voltage levels as a function of time for one or more inputs. For example, leads can be attached to CPU pins to watch the change in voltage level as power is applied to a system. This simple example illustrates the type of observation required to verify that the various power levels required by a CPU are being provided and are arriving in the correct order.

Logic Analyzers

Logic analyzers are a related form of hardware probe whose purpose is to capture hardware protocol-level information being communicated across one or more connections. While oscilloscopes report analog values, logic analyzers are designed to work with digital systems, where information and system state are expressed at discrete voltage levels.

The user of the probe has complete control over how the logic analyzer is used. The user chooses the points of observation within the system under test, provides names for the input signals, and then expresses when information should be captured in terms of the values observed among the input signals. In this way, a logic analyzer can be programmed to observe a precise subset of system state under specific system conditions. Logic analyzers, like oscilloscopes, come in a variety of forms, from stand-alone units to more modest USB-based ones.

As an example, a logic analyzer can be used to capture a trace of memory addresses that are transmitted between a CPU and DRAM. For a given application, a logic analyzer must be capable of capturing the required number of inputs (for example, one for each pin under observation) and must operate at an adequate frequency to observe the state transitions on the pins.

Bus Analyzers

Important computer I/O bus types, such as PCI Express™, are popular enough to warrant their own logic analyzer specialization. A PCI Express bus analyzer can capture and analyze PCI traffic in a much more straightforward way than a comparably capable logic analyzer would. Logic analyzers are generally more expensive, but also may be widely applicable.

Power-On Self-Test (POST) Cards

In IA-32-based devices, power-on self-test (POST) routines run immediately after power is applied to the system. These routines are defined by the BIOS vendor and are the part of the pre-boot sequence intended to catalog system resources and identify easily recognizable classes of system errors.

PC users may be familiar with the so-called “beep codes” that can be used to diagnose some hardware problems during system boot. For example, booting a PC with heavy book on the keyboard

(which presses down multiple keys continuously) will, in many machines, result in a sustained series of beeps as the system boots.

In addition to reporting errors through the system speaker, most BIOS vendors report error numbers through I/O port 0x80, which is mapped to a PCI location that can be observed by a PCI card. These cards, sometimes called POST cards or port 80 cards, can be used to collect a finer grain of BIOS-defined error conditions.

JTAG Adapters

The Joint Test Action Group, or JTAG, standard defines a process for testing boards and controlling processor execution. A CPU or chipset supporting JTAG must support a minimum set of commands to reset the system, read and write system state, and read and write test data. The most basic usage is to send test vectors into a system via the JTAG interface. Such an interaction can be used to test logic correctness that cannot be definitively verified in pre-silicon simulation.

In addition to system test, the JTAG interface and command set can be used to control the CPU and its software. In this context, JTAG can be used as a systems-level debugger, enabling breakpoints, stepping, and source code correlation at the systems software level. JTAG can also be used to initialize devices or program a boot loader into ROM for systems that are powering up for the first time and do not have code in ROM to control the boot sequence. As a debug interface, JTAG can be invaluable because it is available regardless of the soft state of the system.

A JTAG adapter connects to the system under test via the JTAG interface and through a debug machine through some other interface, such as USB.

Most chip vendors provide their own extended version of JTAG. Intel processors, including Intel Atom, feature extended debug ports (XDP), which implement JTAG and some proprietary extensions.

DEBUG PROCESS DETAILS

Given the preceding high-level overview and discussion of tools, we are now ready to consider the steps of the platform debug process in detail.

Visual Examination

The first step of the process, visual examination, is at once the most obvious and the least familiar of the steps in the platform debug process. The step is unfamiliar because most software-level debugging can be carried out in ignorance of the physical state of the computing device. New embedded systems, and new digital systems of all kinds, must be methodically examined for physical correctness and proper component inventory.

Depending on the enclosure, removing covers to get at the underlying circuit board may be a challenge in and of itself. Sometimes rare screw heads are used, requiring rare screwdrivers; other times, no screws are used at all and the proper sequence of squeezes, pressures, and pries must be applied to separate pieces of an enclosure.

Assuming that the circuit board can be uncovered, the next step is to take a physical inventory of proper components while noting signs of physical damage. It is certainly possible for assemblers to

Develop today's embedded computing systems with examples featuring the Intel Atom processor

"I watch with fascination the emerging phenomenon of Internet of Things—a world of intelligent systems working seamlessly and in real time with the cloud infrastructure to collect, organize, and analyze massive data and derive knowledge... The authors of this book not only bring many years of experience in embedded systems, networking, Intel Architecture, and systems engineering, but also represent the visionary technical leadership we will require if we were to successfully take on the challenges of the changing landscape in embedded systems."

—PRANAV MEHTA, Sr. Principal Engineer & CTO, Intel

Unlike embedded microcontrollers that perform only limited functions for industrial applications, modern embedded systems are connected and media-rich, with networking capabilities and graphic user interfaces. **Modern Embedded Computing** will teach you the platform architecture of today's embedded computing systems that drive pervasive, connected devices such as media phones, networking equipment, in vehicle infotainment units and home automation systems.

Examples feature the Intel Atom processor, widely used in devices such as Internet-enabled TVs, tablets, net books, and many embedded applications. Beginning with a discussion of embedded platform architecture and Intel Atom-specific architecture, modular chapters cover system boot-up, operating systems, power optimization, graphics and multimedia, connectivity, and platform

tuning. Both an embedded computing textbook and a resource for professional system designers who need to update their skills, the book provides a comprehensive view of how to develop a framework for embedded systems-on-chips.

- Learn embedded systems design with the Intel Atom Processor, based on the dominant PC chip architecture. Examples use Atom and offer comparisons to other platforms
- Design embedded processors for systems that support gaming, in-vehicle infotainment, medical records retrieval, point-of-sale purchasing, networking, digital storage, and many more retail, consumer and industrial applications
- Explore companion lab materials online that offer hands-on embedded design experience



Peter Barry

Principal Engineer, Intelligent Systems Group,
Intel Corporation



Patrick Crowley

Associate Professor, Computer Science & Engineering,
Washington University in St. Louis

MK

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER

mkp.com

INTEL EXHIBIT 1012 Page 50
IPR2023-00787

Computer Architecture / Embedded Computing

ISBN: 978-0-12-391490-3



Cover image © Getty Images