Compliments of

TEXAS INSTRUMENTS

OMAP™ and DaVinci™ Software

FOR

DUMMIES®

Program OMAP and DaVinci processors quickly and easily with TI software and tools

A Reference for the Rest of Us!®

FREE eTips at dummies.com®

Steve Blonstein
Alan Campbell

Discover the wonderful world of programming the Texas Instruments OMAP and DaVinci processors. OMAP and DaVinci devices contain two unique processors — one general purpose, the other a digital signal processor. Combining these processors inside a single chip makes OMAP and DaVinci programming an interesting challenge. In this book, we explain the high-level concepts required to effectively program these devices. We then let you loose on a development board to create a real video/audio application.

As a companion to this book, we provide a Web site (`www.ti.com/dummiesbook`) where you can find all sorts of goodies to further your OMAP and DaVinci experience. This Web site includes all the software to download to make the video and audio demo run properly on the OMAP development board. You can also visit the TI e-store (`www.ti.com/estore`) where you can purchase additional development boards. Finally, check out (`www.tiexpressdsp.com`), a developer centric wiki site with lots of useful articles and technical documents, many that specifically support OMAP and DaVinci processors.

OMAP™ and DaVinci™ Software FOR DUMMIES®

by Steve Blonstein and
Alan Campbell

Wiley Publishing, Inc.

OMAP™ and DaVinci™ Software For Dummies®

Published by
Wiley Publishing, Inc.
111 River Street
Hoboken, NJ 07030-5774

Published by Wiley Publishing, Inc., Indianapolis, Indiana

ISBN: 978-0-470-39522-6

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

About the Authors

Steve Blonstein is a technical director in the Software Development Organization (SDO) of Texas Instruments. He has spent the last 11 years at TI working on a multitude of programs and projects, making it easier to develop and maintain software on TI processors. Steve was part of the original eXpressDSP initiative that enabled a generation of DSP developers to be more efficient and creative programmers. Now, with the arrival of OMAP and DaVinci class processors, Steve is again part of a team enabling new applications to be developed on these amazingly sophisticated and capable processors.

When away from work, Steve likes to fly his plane around California and spend time with his wife, Andrea, and three children, Samantha, Danielle, and Nicholas, in their home in Palo Alto.

Alan Campbell is the SDO Applications Manager at Texas Instruments, and has 14 years of experience in real-time and DSP applications. He is responsible for applications support of Foundational Tooling, the Integrated Development Environment, and Target Content. Alan's passion is to make all things OMAP and DaVinci easy to use.

In his (limited!) free time, Alan likes to have fun with his twins Erin and Olly, alongside his wife Pauline, in Houston, Texas.

Publisher's Acknowledgments

We're proud of this book; please send us your comments through our Dummies online registration form located at http://dummies.custhelp.com. For other comments, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For details on how to create a custom *For Dummies* book for your business or organization, contact bizdev@wiley.com. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

Some of the people who helped bring this book to market include the following:

Acquisitions, Editorial, and Media Development

Development Editor: Keith Underdahl

Senior Project Editor: Zoë Wykes

Technical Editors: Chris Ring, Katie Roberts-Hoffman, Aravindhan K, Venugopala Madumbu

Editorial Manager: Rev Mengle

Business Development Representative: Kimberly Shelly

Custom Publishing Project Specialist: Michael Sullivan

Cartoons: Rich Tennant (www.the5thwave.com)

Production

Senior Project Coordinator: Kristie Rees

Layout and Graphics: Stacie Brooks, Reuben W. Davis, Shawn Frazier, Sarah Philippart

Proofreaders: Joanne Keaton, Caitie Kelly, Amanda Steiner

Indexer: Johnna VanHoose Dinse

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Director, Acquisitions

Mary C. Corder, Editorial Director

Publishing and Editorial for Consumer Dummies

Diane Graves Steele, Vice President and Publisher, Consumer Dummies

Composition Services

Gerry Fahey, Vice President of Production Services

Debbie Stailey, Director of Composition Services

Table of Contents

Introduction

The creation and consumption of cool gadgets has become a way of life for many designers, engineers, and billions of consumers. These gadgets, whether for music, video, Web, navigation, games, or communications, keep getting ever more sophisticated, *and* cleverer and smaller.

Buried inside these electronic gadgets are high-tech embedded digital processors. Depending on the device, the processor may be responsible for decompressing and displaying video, storing music, running and displaying a Web browser, taking inputs from a touch-screen, or playing amusing ringtones. The tasks handled by these embedded processors usually fall into one of two categories:

- General purpose processing tasks such as data transfer or running graphical user interfaces
- Digital signal-processing tasks such as compressing and decompressing video, audio, and speech streams

Traditionally, different physical devices would be employed to perform these somewhat diverse functions. Today, however, all of this functionality can be packed into a single device.

Texas Instruments has been a pioneer in the area of processor integration with its family of System on Chip (SOC) devices branded *OMAP* and *DaVinci*. These devices make clever use of a combination of general-purpose processors, specialized digital-signal processors, and dedicated hardware accelerators to make digital gadgets literally sing and dance. Even better, devices based on OMAP and DaVinci are really stingy on power consumption, so smaller, lighter batteries can last a long time.

But — you just knew there had to be a but — to make any OMAP or DaVinci device perform the way you want requires a *lot* of software. Fortunately, there's good news, and *really* good news. The first piece of good news is that software *is* soft. In other words, the functionality of a gadget is only limited by the creativity of its programmers.

The really good news is that Texas Instruments provides much of the embedded software and tools required to get OMAP and DaVinci developers up and running fast. Developers can focus on cool applications and not lose valuable time on tasks that don't add real value.

Who Should Read This Book

Any designer or manager about to embark on a project that might involve an OMAP or a DaVinci processor should read this book. If you're a software programmer, you'll discover the various embedded software components and tools that are offered by Texas Instruments, enabling you to get to the good stuff sooner rather than later. If you're worried about being forced into the world of grungy low-level software, you'll appreciate TI's solid software infrastructure that's described in this book.

How to Use This Book

OMAP and DaVinci Software For Dummies is organized so that you can either read it front to back, jump to interesting or relevant sections, or skip straight to the hands-on stuff. If you have little or no experience with software on Texas Instruments devices, we recommend that you become familiar with all the concepts described in Part I before embarking on your development journey in Part II. You wouldn't want to drive off in a rental car at night without knowing how to turn on the headlights, and you wouldn't want to start developing software on a new OMAP or DaVinci device before understanding how TI's system works.

The embedded software and tooling available from Texas Instruments is represented in Figure 1. This book introduces you to the components in the figure and shows you how to use these components to get real-world applications up and running.

How This Book Is Organized

This book is broken into three parts, each offering a different view of the embedded software and tools that run on OMAP and DaVinci devices.

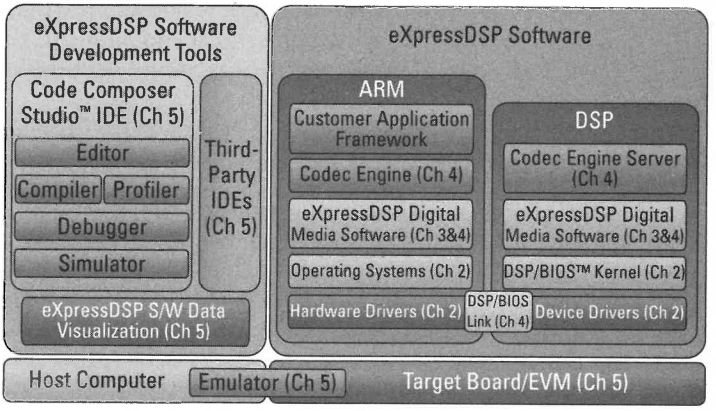


Figure 1: TI offers all these components to speed up your development.

Part I: Understanding the Embedded Software and Tools for OMAP and DaVinci Processors

In Chapter 1, we show you why heterogeneous multi-core devices like OMAP and DaVinci processors have risen to such prominence. Chapter 2 covers the operating systems (Linux, for example) and kernels (DSP/BIOS) required to make OMAP and DaVinci processors run efficiently. Chapter 3 digs into standards (XDAIS, XDM, RTSC) that TI has developed to make it easier for developers to quickly integrate multiple signal processing codecs into the final system. Chapter 4 looks at the framework software, Codec Engine, which forms a convenient housing for XDAIS-compliant algorithms. We also discuss the inter-processor communication protocol (DSP/BIOS Link) used to communicate between the two cores on the SOC. Finally in this part, Chapter 5 explores the development tool choices for both ARM and DSP.

Part II: Building Something Real — Now!

In Part II you work hands-on with one of TI's hardware evaluation platforms (OMAP35xx EVM). You build a demo

application where you can literally see and hear an OMAP processor in action. You actually get a chance to run an MPEG4 video decoder and an AAC audio decoder.

Chapter 6 helps you build a "Hello World" program on the OMAP35xx EVM hardware. Chapter 7 explores the TI MPEG4 video decoder that is used in later chapters. Chapter 8 leads you through packaging a codec and preserving its performance when implemented in a framework. Chapter 9 is where you create DSP server executables. In Chapter 10, you get real codecs running in a test bench. You get to see video and hear audio being processed on the OMAP processor. There's nothing to it!

Part III: The Part of Tens

A standard feature of all *For Dummies* books, each Part of Tens chapter provides ten useful bits of information. In Chapter 11, we cover (almost) ten top codec recommendations that Texas Instruments publishes for codec developers to follow. In Chapter 12, we give you ten great resources for helping you out during your OMAP and DaVinci development.

Icons Used in This Book

We use icons throughout this book to call attention to material worth noting in a special way. Here's a list of the icons you'll see and a description of what each icon means.

Some points bear repeating, and others bear remembering. When you see this icon, take special note of what you're about to read.

This icon indicates technical information that is probably most interesting to programmers, but you never know when you may need to talk to one.

If you see a Tip icon, pay attention — this is handy, real-world advice.

Where to Go from Here

Simply turn the page.

Part I

Understanding the Embedded Software and Tools for OMAP and DaVinci Processors

"We should cast a circle, invoke the elements, and direct the energy. If that doesn't work, we'll read the manual."

In this part . . .

This part explores the factors bringing multi-core processors to the forefront of the programming world, focusing on OMAP and DaVinci Processors from Texas Instruments. With OMAP and DaVinci Processors, you get both a general purpose processor and a digital signal processor. The following chapters provide a high-level overview of a software architecture that gets the most from OMAP and DaVinci devices. We explore operating system choices, embedded software component standards, and TI's software framework — Codec Engine. We also give you options for choosing development tools for both processors.

Chapter 1

OMAP and DaVinci Processors — Hybrids of the Programming World

In This Chapter

- Exploring hybrid System-on-Chip (SOC) trends
- Working with multi-core SOCs
- Using homogeneous and heterogeneous programming models

Are you driving around in one of those cool hybrid cars yet? If not, you're probably cursing at the price of gas because your car only gets 15 to 30 miles per gallon. Our European friends probably think American fuel prices still sound like a bargain, but since this is *OMAP and DaVinci Software For Dummies* and not *Global Energy Policy For Dummies*, in this book we stick to the subject of what hybrids have to do with programming.

"What exactly do hybrids have to do with programming?"

Great question!

In this chapter, we discuss the unique factors that have caused the hybrid SOC (System on Chip) trend to develop. First, we consider the problems encountered by circuit designers as chip clock frequencies headed towards infinity (and literal meltdowns). Next, we explore the multi-core trends that exploited the ability to pack hundreds of millions of transistors on a single chip. Finally, we look at why, for

the types of applications described in this book, it's best to have both a general purpose processor (GPP) and a special purpose digital signal processor (DSP).

A Hybrid Is As a Hybrid Does

Hybrid cars combine together electric and gasoline drive trains to increase fuel efficiency. Hybrids also boost performance, because the combined drive train produces better torque and acceleration than either motor can deliver on its own. You can even improve stopping distances by leveraging the regenerative braking principle. Finally, don't forget that a hybrid car emits a much smaller amount of pollution for each mile traveled.

A remarkably similar trend has occurred during the past decade in the embedded silicon and programming world. SOC silicon processors have gained widespread acceptance because they use less power, offer better performance, and emit less heat pollution into electronic devices.

High Frequency Meltdowns

Automobile buffs clearly understand that simply doubling the horsepower in a car doesn't double the vehicle's top speed. Annoying things like road friction and wind resistance conspire to make the math much less favorable than one would hope.

It wasn't so long ago that most people assumed that the clock speeds on silicon (Si) devices would continue to get higher unabated. People were touting 5GHz processors as being "just around the corner." Apparently, that car spun off the road somewhere because 5GHz chips still aren't available, and they don't seem to be anywhere on the horizon.

So why aren't there 5GHz chips? Designers ran head-on into two related problems:

- **Fuel consumption:** Fuel (or should we say *electrical power*) consumption becomes too prohibitive at higher frequencies.
- **Heat dissipation:** As electrical power consumption grows, adequate heat dissipation becomes impractical in all but the most powerful server-type computers.

The challenge to Si designers is this: How do Si designers boost chip performance while alleviating the power and heat problems? Since they can't just increase chip frequencies, how can they make better use of lower frequencies? The solution is the multi-core device.

Multiplying Performance with Multi-Cores

The people who realized that 5GHz processors wouldn't be practical in common embedded applications needed to come up with practical alternatives. As transistors got smaller, it became possible to pack hundreds of millions of transistors (even a billion!) on a single chip. So silicon chip designers concluded that it would be more practical to place several processors, each running at a lower clock speed, on a single chip. Hence, the multi-core processor was born.

Currently, the two common approaches to designing multi-core architectures are

- **Homogeneous core approach.** Essentially, the silicon architect places two or more identical (homogeneous) cores on a single chip. Devices are already available with four, six, or even eight cores. This homogeneous approach is generally the simplest multi-core architecture, but it has some serious limitations that we discuss later in the chapter.
- **Heterogeneous core approach.** The principle of the heterogeneous methodology is for silicon architects to place two or more different cores on the same chip. Huh? Why would you make things any more complicated than they already are on a chip with hundreds of millions of transistors? The reasons are very similar to the gasoline/electric combination in a hybrid car. The two motors are tuned to be most efficient at different tasks and can even be combined to produce a killer combination. Figure 1-1 shows a Texas Instruments OMAP core that utilizes a heterogeneous design approach. One core serves as the GPP while another is the DSP.

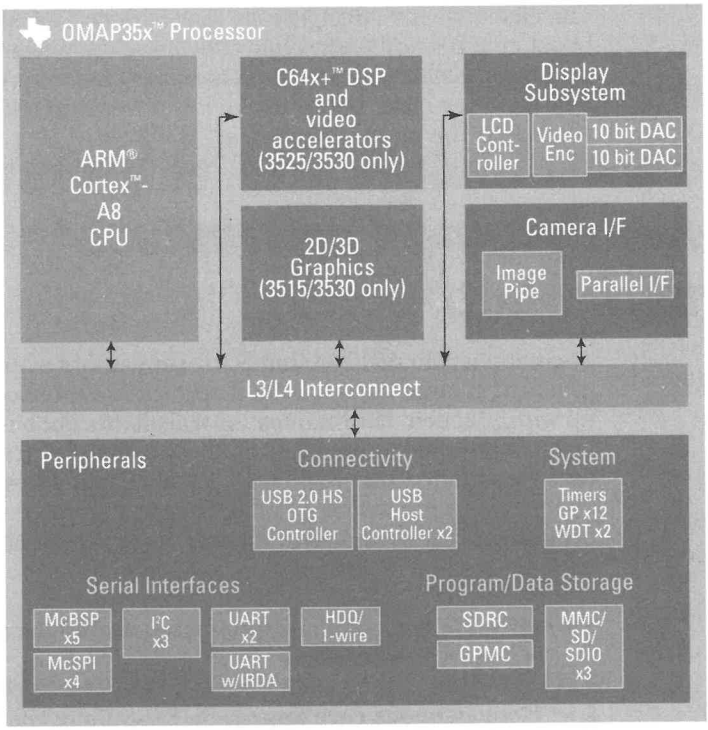


Figure 1-1: OMAP processors use a heterogeneous design approach.

We could debate the pros and cons of the homogeneous and heterogeneous approaches all day, but that it isn't going to get any software code running on either architecture. Ultimately, someone (you?) has to get a software application running on a multi-core device.

Homogeneous multi-core programming models

The homogeneous core approach works well in a variety of programming scenarios.

One scenario involves running the same task over and over again, with some number of identical tasks running on one core and more identical tasks running on the other core(s). Suppose, for example, that you need to run a voice codec

application. One core might be able to handle eight channels of voice simultaneously. This means that with two cores you can handle 16 channels, four cores can handle 32 channels, and so on. Of course, this presumes that you have the I/O bandwidth to get the voice data on and off the multi-core device in real-time.

Another scenario that might work well on a homogeneous multi-core architecture is where independent tasks split conveniently across the cores with little or no need for the tasks to communicate with each other. An example of this would be the previously mentioned multi-channel voice application, but with an additional need to simultaneously run a video codec. Hypothetically, say that the video codec consumes about eight times the compute cycles as a single voice channel. Voilà — you run the eight voice channels on one core and the video channel on the other core.

Homogenous multi-core processors work best when you need to process a lot of identical tasks, or when dissimilar tasks can be easily separated among multiple cores.

Heterogeneous multi-core programming models

Suppose you need a device to handle voice and video media, plus a sophisticated graphical user interface. Oh, and it's desirable to reuse a whole bunch of high-level OS application code that was developed for your previous product.

General purpose processors that are good at handling high-level operating systems and lots of I/O tasks aren't very efficient or capable at handling voice, audio, and video signal processing. Meanwhile, a digital signal processor is *very* capable and efficient at running video, audio, and speech processing, but a DSP doesn't support general-purpose high-level OS applications efficiently.

Applications that combine diverse requirements are where OMAP and DaVinci devices really shine. These devices embrace the heterogeneous multi-core programming model by combining GPP and DSP cores on a single chip. OMAP and DaVinci devices include

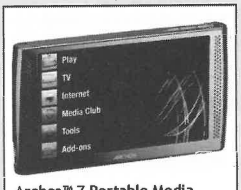
- **An ARM Ltd general purpose processor.** The ARM processor is ideally suited to running a common operating system like Linux, and it can handle most of the input/output (I/O) and housekeeping functions such as a graphical user interface (GUI).
- **A high performance C64x+ DSP.** The DSP handles all the really intense signal processing functions required in today's intense multimedia applications.

Rocket power: Accelerating hardware performance

Hybrid cars are undoubtedly fuel efficient, but hasn't everyone secretly wanted to tune their cars for more performance? How about bolting a rocket motor with afterburners to the back of the car? That'll boost the performance quite a bit.

In the silicon world, the equivalent of a rocket motor is referred to as a *hardware accelerator*. A heterogeneous multi-core device needs to perform certain functions a zillion times at extremely high speeds. These functions are predictable, repeatable, and deterministic. It's generally a waste to consume either general processor cycles or even specialized DSP cycles on such tasks. It's much more efficient, both in terms of Si area and power dissipation, to build dedicated hardware circuits to perform these tasks. An example of such an accelerator is the video and image co-processor (VICP). Such accelerators often perform two dimensional image transforms and motion estimation for coding movement between video frames. Many OMAP and DaVinci devices like the ones shown here leverage the hardware accelerator concept to efficiently boost performance.

AVM IP Set-Top Box



Archos™ 7 Portable Media Player

RED Embedded Design VPHS405 Wifi Videophone

Chapter 2

Using the Right Operating Systems

In This Chapter

- Understanding the DSP scheduler
- Using Linux — a real operating system for the general purpose processor
- Choosing between community and commercial Linux
- Working with applications that run on the two processors

A hybrid car contains two different types of motors. One motor is an internal combustion engine that must be fueled by gasoline (petrol for our European friends), and the other is an electric motor that must be powered by electricity. Each motor must be fed from the correct energy source; an internal combustion engine won't run on electricity, and if you pour gasoline into an electric motor, you'll probably start a fire.

A similar but less obvious problem presents itself when it comes to powering an OMAP or DaVinci hybrid SOC (System on Chip). As we discuss in Chapter 1, these SOCs include both a GPP (general purpose processor) and a DSP (digital signal processor). Something has to schedule tasks effectively on both of these processors. For the DSP, the task scheduler is a lightweight scheduler called DSP/BIOS. For the GPP, the SOC leverages the wonderful world of Linux.

In this chapter, we explore DSP/BIOS and show you how the DSP/BIOS attributes make it ideal for OMAP and DaVinci devices. We also look at how the Linux operating system is ideal for running the general purpose processor. Finally, we take a brief look at applications that run on both the DSP and GPP.

BIOS is one of those often misused acronyms. Rather than focusing on the definition of the four letters, here's how we define it: DSP/BIOS is a royalty-free real-time multi-tasking kernel (mini-operating system) created for the TMS320 family of DSPs from Texas Instruments.

DSP/BIOS — Real, Real-Time

Considering the "DSP" part of DSP/BIOS, guess where this operating system runs? Good guess — it's created for the DSP part of TI's heterogeneous multi-core devices. DSP/BIOS is included at no charge with the standard tools distribution from TI, and it has no run-time royalties. DSP/BIOS isn't suitable for every use, but it is ideal for real-time DSP task scheduling, thanks to three key attributes:

- Scalability
- Speed
- Low latency

These three traits make DSP/BIOS a *real* real-time task scheduler, preventing unacceptable performance problems in your applications. The next few sections explore the scalability, speed, and low interrupt latency of DSP/BIOS.

Some folks might get into lengthy philosophical discussions about whether DSP/BIOS is truly an operating system or just a scheduling kernel. Having flunked Philosophy 101, we'll skip that debate, but we can declare that DSP/BIOS is an ideal real-time scheduler for the kinds of tasks that the DSP is designed to run.

Scalability

If small is beautiful, then DSP/BIOS is definitely *the* Beauty. (We'll try to avoid any Beast references!) A small (and beautiful) scheduling kernel helps you leverage on-chip resources for the stuff that really matters — like video, imaging, speech, and audio algorithms.

It's really important for the scheduling kernel on the DSP to be scalable so that it uses just what's needed to get the job done. Since it doesn't link in a whole bunch of code that'll never get used, DSP/BIOS can run flat-out when needed. In many cases — including examples shown in Part II — DSP/BIOS can actually reside in slower external memory outside the SOC, leaving maximum internal chip resources for the signal processing functions. Table 2-1 shows the scalable DSP/BIOS memory configuration for a typical DSP configuration.

To put the size in perspective, DSP/BIOS fills just tens of kilobytes (yes, *kilobytes*). Compare that to "typical" operating systems that tend to be hundreds of kilobytes or (in most cases) several Beastly Mbytes! (Uh oh — there's that Beast we wanted to avoid mentioning.) Refer to TI document SPRA772A at `www.ti.com` for detailed memory size information.

Table 2-1 Typical DSP Memory Configuration

Data or Code Section	*DSP/BIOS 5.3x Size (C64x Bytes)*
Code	15648
Initialized Data	291
Uninitialized Data	8616
C-Initialization	3460
TOTAL	28015

Speed

A small DSP scheduling kernel isn't much good if it consumes too many *MIPS* (million instructions per second) on the processor core. Every cycle taken by the scheduler is one less cycle available for what the DSP is *supposed* to be doing — namely, signal processing algorithms. DSP/BIOS shines because all of its modules are highly optimized for the DSP core. It's also ideally suited for handling streaming data, which is exactly what the typical DSP will see in an audio/video application. Table 2-2 shows the high execution speeds for certain key DSP/BIOS modules.

Table 2-2 DSP/BIOS Latency

Speed/Latency Benchmark	*DSP/BIOS 5.3x (C64x cycles)*
HWI_Enable	12
HWI_dispatcher	125
Hardware interrupt to SWI	184
SWI post, including context switch	117
Hardware interrupt to blocked task	584
TSK_create: no context switch	849
TSK_yield	226
Post a semaphore, no waiting task	28
Post a semaphore, context switch	256

Low latency

A DSP operating system must be *deterministic,* which means that the system produces the same results every time with no random wishy-washy performance ambiguities. If an operating system claims a task switch time of one millisecond (1ms), is that a 1ms average or a best case? What's the worst case? What if the worst case is 50ms and you're operating a video system where something really important has to happen every 33.3ms? Oops.

Latency — the delay between when an operation is initiated and it starts to take effect — is one thing that separates so-called real-time operating systems (like DSP/BIOS) from larger beast-like operating systems. As you can see in Table 2-2, DSP/BIOS has very low latency and it's deterministic, so it gets to the next task in a timely fashion — *every time*!

Linux — a Real Operating System for OMAP and DaVinci Devices

On both OMAP and DaVinci devices, the General Purpose Processor part of the device is a member of the ARM family of

cores. These cores typically have a multitude of peripherals connected to the core such as serial ports, USB ports, Ethernet ports, multimedia cards, and more. Several aspects of Linux make it the ideal choice for the general purpose processor.

In the following sections, we discuss the popularity of Linux, and the interesting business model aspects of using Linux. Finally, we look at the applications that the Linux community has created.

Linux has become so widespread that variants of it are used in different types of applications. So-called Enterprise Linux runs on "heavy-iron" server farms that are running search applications and other heavy-duty tasks. Another variant, Embedded Linux, is usually configured (stripped-down) to run in lighter-weight applications like the ones we describe throughout this book. ***Note:*** Even though we use Embedded Linux, we call it simply Linux — for the sake of brevity.

Linux wins popularity contests

Linux has become very popular, so it has a lot of supporters and followers. These dedicated supporters — both paid and pro bono (donated) — ensure regular updates and advances to the Linux operating system. Many different types of processors are supported, with ARM being a popular one. Software drivers for a wide variety of peripherals and ports are also readily available. In recent years, Linux has become a lot more proficient at running embedded applications as opposed to just enterprise server-type applications, where Linux first made inroads.

The Linux price is right

You can't argue with the price of Linux: It's free! The open source model and the Free Software Foundation have helped to make Linux ubiquitous. But it's not necessarily all roses — there are real costs associated with developing and maintaining a Linux application. Here we spend a minute or two exploring the realities of the Linux world, which actually divides into two continents: community Linux and commercial Linux.

Picking community Linux

If there ever was a software "Wild West," some would argue that the public community "git tree" version of Linux would be it. Community Linux is usually based on the absolutely latest and greatest version of the Linux kernel. Common problems are incompatibilities between kernel versions and dependencies of non-kernel components (for example, software drivers) on certain kernel versions.

Unless you're a Brit, the word *git* probably doesn't mean much to you. *Git* is an English word for a silly or worthless person. But don't be confused when you visit the public git tree to download the latest version of the software. Although free, the software is definitely not "silly" or "worthless." In the Linux programming world, Git is a version control system and a git tree is where you go to see the version history and status of a given Linux kernel or other project.

How a given community Linux version is maintained varies greatly based on the *owner* of that git tree. Basically, there are no guarantees. Individuals relying on a community version of Linux should not only be comfortable in the world of Linux, but should also be able to take a few knocks as they discover and become victim to not-completely-baked code. Both OMAP and DaVinci Linux have community git trees that you can find simply by searching the Web.

- OMAP git tree:
  `http://source.mvista.com/git/gitweb.cgi?p=linux-omap-2.6.git;a=summary`
- DaVinci git tree:
  `http://source.mvista.com/git/gitweb.cgi?p=linux-davinci-2.6.git;a=summary`

Public Linux git trees usually have owners. An owner is often a single person who is responsible for posting updates and patches to a publicly available version of the operating system. This person must have the respect of the Linux community (a somewhat vague concept in itself) and be viewed as somewhat neutral in the decision-making process for the merits of what does and doesn't get posted to the git tree.

Paying for commercial Linux

Several companies exist to tame the Wild West of Linux development. The value-add from these companies is that they take a snapshot of a recent version of Linux and then test it a lot, fix bugs, and make the version robust and bullet-proof. The obvious upside to this approach is that the final product-ized version of Linux is probably quite a bit more stable than the latest public version. But there are two key downsides:

- **Commercial Linux providers are companies that need money to stay in business.** (Funny how that works!) They extract your money either for development tools and/or for support and services.
- **Updates are a step behind community versions.** By the time a commercial Linux version is product-ized, the snapshot of the public version on which it's based has moved on. So, commercial versions are generally behind the leading edge.

montavista

TI has partnered with MontaVista Software as the commercial Linux provider of choice for OMAP and DaVinci processors. For more on these offerings, see Chapter 5.

Using Applications

Anytime an operating system becomes popular, application developers are likely to follow. Many new applications are open source and readily available at no cost to the user. A good example is GStreamer, an OS-independent multimedia framework that has been ported to Linux and supports multiple multimedia applications.

As we discuss in more detail in Chapter 4, Codec Engine is the primary application running on DSP/BIOS on the DSP for OMAP and DaVinci processors. Codec Engine is a unique software framework designed, maintained, and supported by TI itself to specifically run all the cool signal-processing algorithms that run so efficiently on the DSP part of the hybrid SOC.

Green Hills Software, Inc.

Although this book focuses on the Linux OS for the ARM general purpose processor, other operating systems are also supported on OMAP and DaVinci devices. Two examples are Microsoft WinCE and Greenhills Integrity.

Connecting It All Together

In order for the two hybrid processors (the DSP and the GPP) to work together effectively, you need a few more building blocks.

Typical Linux application developers don't want to have to learn the nitty-gritty details of the DSP and DSP/BIOS scheduler. This means that the DSP functionality needs to be abstracted in a simple and efficient way. It turns out that the Codec Engine framework application software (see Chapter 4) running on the DSP has a companion piece running on the ARM general purpose processor. These two framework pieces communicate with each other in a predetermined way via another piece of software called DSP/BIOS Link (again, see Chapter 4). Thus, the application developer doesn't have to worry about the DSP details.

Meanwhile, another group of developers is building the really clever signal-processing algorithms that run on the DSP. TI refers to these signal-processing algorithms as eXpressDSP Digital Media Software.

In order for these algorithms to play fairly on the DSP, they all need to follow certain rules and guidelines so that they integrate easily and correctly into the final system. The rules and guidelines apply to the algorithm component developers. They're also useful for the algorithm consumer who wants to check and verify that the components are indeed good citizens before integrating them into the final system. If the algorithms don't follow the rules, chaos is quite likely to ensue. And in the software world, chaos equates to inexplicable crashes that defy discovery or explanation and just end up wasting valuable evenings and weekends — nothing any of us really want to do.

Chapter 3 explores a set of TI created standards that set the rules of the playground so that you can keep those precious weekends and evenings to yourselves.

Chapter 3

Digital Media Software: Standardizing How Codecs Work Together

In This Chapter

- Reviewing codec standards
- Verifying codecs with XDAIS
- Working with standard interfaces for classes of codecs through XDM
- Taking a look at RTSC and why consistent packaging is important

Modern multimedia-rich electronic devices use digital signal processing for playing, storing, sending, and retrieving various kinds of media. To make this all happen, designers use sophisticated software codecs that compress, decompress, transmit, and play back video, audio, still images, and speech data. To fully enable OMAP or DaVinci devices, designers choose one or more algorithms to run on each device. These algorithms are called eXpressDSP Digital Media Software.

In this chapter, we explore three OMAP and DaVinci codec standards. We start with the granddaddy of them all — XDAIS (eXpressDSP Algorithm Interoperability Standard). Next, we look at XDM (eXpressDSP Digital Media), an extension to XDAIS that provides standard interfaces for specific classes of codecs. Finally, we take a look at RTSC (pronounced "RITSY" — imagine that, like having our own cereal! . . . more on that in a minute), which is a collection of tools and utilities for Real Time Software Components. These standards enable developers

who know little about the inner workings of sophisticated video/audio algorithms to easily integrate multiple codecs into their systems.

Taking a Look at TI Standards

Literally hundreds of different algorithms are available from both TI and various third parties, and new algorithms are being created all the time. Some algorithms implement existing industry standards (for example, MPEG4, H.264, and so on) or meet new standards. Sometimes algorithms are proprietary because the creators believe they have some extra "special sauce" that differentiates their design from the pack. Unfortunately, where there is great flexibility, there is also potential danger in the form of rogue software that looks good on paper but causes problems in real systems. This leads to the need for additional standards.

Standards are important in nearly all aspects of life. Consider, for a moment, one of those breakfast cereal multi-packs that contain eight cute little single-serving boxes — you know, the little boxes the kids love to fight over. As you look at the single-serving boxes, consider the various standards used in the product. Each box has the same physical size (even if the contents are different), they all open in the same manner, and they all contain a standardized nutrition label. (Yup, way too much sugar!) The eight little boxes also bundle together perfectly in a shrink-wrapped package, ready to be snatched off the shelf by sugar-deprived children. Standards make these single-serving boxes successful products.

Standards are also necessary for eXpressDSP Digital Media Software, even if we don't add excessive amounts of sugar. Texas Instruments has implemented a set of standards that makes using multimedia codecs on either OMAP or DaVinci SOCs as painless as possible.

XDAIS — Ensuring Codecs Play Fairly

The eXpressDSP Algorithm Interoperability Standard (XDAIS, pronounced *X-Dayus*) was introduced in 1999. XDAIS is an

extensive set of rules to make codecs behave appropriately, allowing seamless system integration. XDAIS brought order to what was rapidly becoming a chaotic world of commercial off-the-shelf (COTS) codec software that was exploding in popularity with the advent of the TMS320C55x and TMS320C6x DSPs.

Before XDAIS, codec developers pretty much had a lawless free-for-all. Problems came when unfortunate system integrators had to take one or more codecs and make them work together in a single system. Countless nights and weekends were lost to the resulting insanity. Some codec development behaviors that were bad included the following:

- Codecs couldn't be relocated in system memory, forcing system integrators to have codecs hog certain places in the system.
- Some codecs made calls directly into the underlying hardware, possibly clashing with other processes already using that hardware.
- Some codecs weren't re-entrant. This prevents proper multi-channel operation of codecs.
- Interrupts were disabled for excessively long periods, making it so that other processes couldn't get a word in edgeways.
- "Creative" marketing numbers for performance and memory usage were often best-case numbers as opposed to real performance measures.

XDAIS has undergone several revisions since its introduction. The revisions have largely been supersets of previous versions of the standard. The TI document covering XDAIS is called SPRU352 and can be found at `www.ti.com`. Following the number, you see a letter, indicating the revision version. That letter should be version G or later.

XDAIS breaks down into three distinct categories: rules, guidelines, and generic interfaces. The next few sections take a brief look at each group.

XDAIS rules

Needless to say, the XDAIS rules are mandatory. The rules must be followed in order for an algorithm to declare

"compliance" with the standard. Currently, there are 46 rules. Although this number might seem large, a lot of the rules are based on common sense. Other rules may seem arbitrary, but they force consistency. For example, Rule 25 requires that all C6x-based algorithms are delivered in a little-endian format. More on this later.

The good news is that for an algorithm consumer (rather than the original creator), it's not critical to know the number of rules and exactly what they do. What *is* important is knowing that the rules have been followed so that the algorithm easily integrates into the system, operates as expected, and delivers performance that matches the documentation. Some of the most important XDAIS rules include

- **Rule 2: All algorithms must be reentrant within a preemptive environment**. This ensures that algorithms support multiple instances.
- **Rule 4: All algorithm code must be fully relocatable**. This ensures that system integrators are free to place codecs where they see fit, not where algorithm creators might force them to go.
- **Rule 6: Algorithms must never directly access any peripheral device**. This solves the problem of rogue algorithms directly accessing the peripheral hardware while other processes may already be using the peripheral.
- **Rule 12: All algorithms must implement the IALG interface.** IALG is a memory resource management interface that all algorithms deploy to allow system integrators to distribute memory resources as they deem appropriate, as opposed to algorithms just "grabbing" whatever they feel like. We cover IALG in more detail later in this chapter.
- **Rule 23: All algorithms must characterize their worst-case interrupt latency for every operation.** Rules 19–24 are so-called performance characterization rules that force algorithm creators to document resource usage and performance numbers.
- **Rule 25: All C6x algorithms must be supplied in little-endian format**. This is one of those potentially arbitrary rules. What about big-endian format? By picking at least one format for everyone to deploy, system integrators are guaranteed to have at least one set of little-endian

algorithms as opposed to a mix (up!) — which is potentially troublesome.

XDAIS guidelines

XDAIS has rules, and it also has guidelines. What's the difference? XDAIS guidelines are sort of like those dietary guidelines that "suggest" limiting how much sugar your kids consume. The guidelines are practical most of the time, but you'll always find exceptions in which you can break them (like while on vacation, for example). For the software system integrator, the impact of not following a guideline should be minor, not catastrophic. Some important XDAIS guidelines include

- **Guideline 5: Algorithms should keep stack size requirements to a minimum.** System integrators want to know how much total stack size their final systems will require. Generally they want to keep sizes manageable so they don't have to plan for ridiculous resource-hogging worst-case scenarios. Guideline 5 suggests that each codec do its part to help with the overall stack size goal.
- **Guideline 12: All C6x algorithms should be supplied in both little and big-endian formats.** Here's a guideline that is really a companion to Rule 25 discussed earlier. Although Rule 25 requires the algorithm developer to supply all algorithms in at least little-endian format, it's actually better if the developer provides both a little-endian and a big-endian implementation of the algorithm, hence this guideline.

XDAIS interfaces

The XDAIS standard mandates the use of one or more interfaces that standardize the way codecs request and are supplied with resources. The principle behind these interfaces is to ensure fairness in the assignment of critical resources. So, rather than "selfish" algorithms that just grab all the good stuff, the XDAIS interfaces standardize how algorithms request resources. System integrators then make either design-time or run-time decisions about who gets what. The XDAIS standard defines three core interfaces:

- **IALG.** This interface is required by all algorithms and handles each algorithm's memory requirements.
- **IDMA3.** This version replaces earlier versions called IDMA and IDMA2. IDMA3 is required if the algorithm needs certain types of DMA resources.
- **IRES.** This interface is required if the algorithm needs other resources like hardware accelerators. (Check out Chapter 2 for some good stuff about a rocket motor strapped to the back of the hybrid car.)

XDAIS compliance

Standards are all well and good, but at some point you need to be assured that what you're buying actually adheres to the standards. In our cereal analogy discussed earlier in this chapter, various federal agencies test for compliance with nutrition labeling, food safety, and packaging safety standards. TI offers a tool called QualiTI that is specifically designed for testing algorithms for standards compliance. QualiTI helps codec creators check their handiwork, and lets codec users verify compliance before getting into serious system integration.

The QualiTI tool is available as part of the XDAIS Developer's Kit, which you can download (for free!) from `www.ti.com`. We give this tool a tryout in Chapter 7.

XDM — Standard Interfaces for Common Classes of Codecs

XDAIS handles the rules, guidelines, and interfaces common to *all* codecs. But a limitation appears because XDAIS knows nothing about the specific nature of interfaces used in video, imaging, speech, and audio codecs. As a result, most codec developers used to produce their own unique interfaces. This was fine until system integrators wanted to exchange one video codec for another or simply swap brand A for brand B.

With different codec interfaces, extra work was required to integrate the replacement codec into the system. XDM extends the original XDAIS standard to address this specific issue. XDM specifies encoder and decoder interfaces for four classes of algorithms: video, audio, speech, and imaging. XDM interfaces are simple, lightweight, and extensible where required by codec developers.

The main benefit of deploying XDM interfaces on certain classes of codecs is interchangeability. For example, replacing one vendor's XDM MPEG4 video codec with another vendor's version should be easy. The application only has to know about the XDM interface (which doesn't change) and doesn't have to worry about the specifics of a particular video codec in question.

As you work with codecs, you may see references to the acronym VISA. This acronym has nothing to do with credit cards or getting permission to visit foreign countries. In the XDM world, VISA is the name of the application level interface that's used to call the four standard classes of XDM codecs: Video, Imaging, Speech, and Audio. We recommend that you don't leave home without this acronym. You might not be able to pay for all those multi-pack cereals that the kids want!

The XDM standard specifies eight different generic code interfaces. They are

- IVIDENCx, for video encoders
- IVIDDECx, for video decoders
- IAUDENCx, for audio encoders
- IAUDDECx, for audio decoders
- ISPHENCx, for speech encoders
- ISPHDECx, for speech decoders
- IIMGENCx, for image encoders
- IIMGDECx, for image decoders

If you want to extend XDM, refer to the article entitled "Extending data structures in XDM" found on `http://wiki.davincidsp.com`, but proceed with caution. Problems occur when interfaces are extended for too many

"marginal" reasons. The net result is that many codecs aren't really standard anymore. The onus is on codec developers to make judicious use of extensibility and for consumers to insist on and check that extensibility isn't being abused.

Like all good things that mature over time, some additional interfaces have been introduced, such as IVIDDEC2 and IVIDENC1. These interfaces provide improvements to the original interfaces such as enabling advanced buffer management on the DaVinci High Definition platforms. The advice for codec creators is to implement the most recent interfaces since these often provide capabilities that can be used without having to make custom extensions. See the preceding warning for more on custom extensions.

RTSC — Standardized Packaging for All Codecs

Earlier in this chapter, we talk about little cereal boxes and how we probably just take for granted that the boxes and cereal conform to established standards. Until recently, we could *not* have said the same about signal-processing codecs. There weren't any standards, so it was pretty much a free-for-all when it came time to package codecs for delivery. The success of codec integration owed as much to chance and good fortune as to skill and developer knowledge.

Real Time Software Components (RTSC) is an open-source initiative that was initially driven by TI to help standardize embedded components. Component software concepts have been around for years, first in the enterprise computing world and then rapidly spreading to the desktop world. The whole concept relies on the reuse of software components by multiple consumers. To make component reuse possible requires standards and methodologies that foster the easy transfer of components between producers and consumers.

Why RTSC?

For various reasons, component technology was late in coming to the embedded programming world. There are probably a few key reasons for this. One reason is the overhead associated with component models — and the words *overhead* and *embedded programming* usually don't mix well. Embedded systems are all about performance and lightweight designs, so *overhead* is a dirty word. RTSC is ideal for embedded programming because it tackles the issue of overhead, er, head-on, so to speak.

The second reason that component technology has been slow to catch on is the fractured nature of the embedded world. No single large company is driving standards the same way that IBM, Microsoft, and Sun have done in the enterprise and desktop worlds. By taking RTSC open source, RTSC should gain broad acceptance across the industry without the need for a single 800-pound gorilla to drive its acceptance. There's a lot to RTSC that we don't cover in this book, but we do talk about packaging because that's the last part of the puzzle for delivering a "good" codec.

To find out more about RTSC, visit `www.eclipse.org/dsdp/rtsc`. Eclipse is an open-source industry consortium in which TI participates. RTSC is a project in the Eclipse community.

RTSC rules and guidelines

RTSC consists of rules and guidelines that packages (for example, codec packages) should follow in order to be considered RTSC-compliant. As with XDAIS, the consumer of a package doesn't have to worry too much about specific rules because most of that burden is placed on the component producer. However, the rules and guidelines have been written so that it's easy for tools such as the RTSC Package Wizard to automate much of what has to be done by the producer. (We discuss the RTSC Package Wizard in Chapter 7.)

The basic principles that drive RTSC packaging requirements are delivery (standard look/feel, documentation, and so on), configurability (versioning and so on), and assembly (modularity and so on).

Here are examples of some of the more important rules:

- **Package Naming:**
  - Every package must have a unique name comprised of lowercase letters (for example, ti.sdo.codecs.mpeg4dec). We recommend starting each package with your company/organization name.
  - Each package must reside in a directory structure that matches its package name, such as /alan/workdir/dummiesbook/ti/sdo/codecs/mpeg4dec.
- **Package Files:** All package files must exist in the package's base directory or sub-directory, such as impeg4dec.h in the mpeg4dec directory.
- **Package Basic Rule:** All filename references within a package should use `/ not \` so packages can exist in both Windows and Linux host environments.
- **Package Compatibility:** Packages must specify which other packages they depend upon, as well as which versions of those packages are required.
- **Package Documentation:** Documentation should be included as part of the package and must be in PDF format, except that online release notes may be in HTML format.
- **Source Packages:**
  - Source packages should be 100 percent development host independent.
  - Configurable source packages must not require referenced packages to be rebuilt.

Chapter 4

Multimedia Framework Products — Revving the Codec Engine

In This Chapter

- Exploring Multimedia Framework Products
- Understanding XDAIS Framework Components and Codec Engine
- Connecting multiple processors with DSP/BIOS Link
- Picking the right multimedia codecs for your application

Texas Instruments has been busy building a software infrastructure that represents the programming equivalent of plumbing, electricity, and wood beams. The elements of this infrastructure are called Multimedia Framework Products (MFP). Although most developers could probably create this stuff on their own, there's not much value in reinventing the same items over and over again. TI maintains and enhances these products so that you don't have to.

In this chapter, we introduce you to the MFPs available from TI. We also introduce additional foundational software — DSP/BIOS Link — for inter-processor communications.

Multimedia Framework Products

Multimedia Framework Products provide the software infrastructure you need as you develop your own applications for OMAP and DaVinci–based systems. MFPs break down into two distinct categories:

- **Framework Components (FC)** are off-the-shelf modules designed specifically to help instantiate (which is just a fancy one-word way of saying "create an instance of something") and run XDAIS/XDM codecs (refer to Chapter 3 for more on XDAIS and XDM standards). Think of Framework Components as the plumbing, electrical systems, and wood beams used to construct the software house that your application will leverage. These Framework Components are freely available to any developer. You can get started at `www.ti.com/dummiesbook`.
- **Codec Engine** is more like a model home. It makes extensive use of the Framework Components (the virtual plumbing, electrical systems, and wood beams) as well as other infrastructure components that we discuss in Chapters 2 and 3. Even though TI provides a fairly complete model home, it's still up to the user (that's you) to decorate the place. In the OMAP and DaVinci world, this means picking off-the-shelf codecs, adding proprietary algorithms, and writing high level applications.

Multimedia Framework Products replace a previous range of software frameworks called *Reference Frameworks.* If you've seen references to things like RF3 or RF5, these acronyms refer to different levels of Reference Frameworks. Many of the key concepts from the original Reference Frameworks are now incorporated in MFP. A benefit of the new MFP approach is that the individual Framework Components are separately available — something that wasn't the case in previous generations of Reference Frameworks.

Taking inventory of XDAIS framework components

In Chapter 3, we talk about the eXpressDSP Algorithm Interoperability Standard (XDAIS). This standard consists of a set of rules, guidelines, and interfaces that all eXpressDSP Multimedia Software (codecs) must implement to be considered compliant with the standard. As we note in Chapter 3, XDAIS rules are mandatory whereas XDAIS guidelines are optional (though strongly recommended, of course). Codecs must follow XDAIS rules and guidelines to allow the system integrator, either at design-time or run-time, to determine who gets what in terms of valuable on-chip resources.

Interfacing to standardized codecs requires some basic software components that every application developer must use in order to instantiate, query, and properly run the codec in OMAP and DaVinci environments. TI provides several key prebuilt components to jump-start the task of putting together an integrated system:

- **DSKT2** (pronounced *D-Socket 2*). This Framework Component exercises the XDAIS IALG interface (see Chapter 3). The IALG interface is responsible for determining what system memory resources are required by the codec. To ensure fair play, codecs are not allowed to just grab resources at will, but must adhere to the IALG interface. DSKT2 establishes resource requirements, allocates memory as the system sees fit, and then instantiates the codec so it operates properly.
- **DMAN3** (pronounced *D-MAN3* — that was hard!). This component uses the XDAIS IDMA3 interface (see Chapter 3) to establish which DMA resources are required by each codec in the system. Just like memory resources handled by IALG, codecs are not allowed to simply access DMA resources before going through an approved negotiation.

  Okay, someone probably wants to know how we got to DSKT2 and DMAN3? What happened to DSKT, DMAN and DMAN2? Did they quit or get fired? More advanced DSP cores such as the C64+ DSP core, found on both OMAP and DaVinci devices, feature advanced DMA features called QDMA and EDMA 3.0. In order to leverage these new capabilities, DMAN and DMAN2 had to be put out to pasture, and the new, more capable DMAN3 was brought in to take full advantage of the new C64+ features.

- **ACPY3** (pronounced *A-COPY3*). This component is designed specifically for usage by codecs in which huge amounts of data must be copied from one memory to another. ACPY3 leverages fast DMA-based memory copies. These DMA resources are acquired through the DMAN3 interface. ACPY3 has been highly optimized for the C64+ DSP core found on both OMAP and DaVinci devices. And before you ask what happened to ACPY and ACPY2, check the same unemployment lines where you'll find DMAN and DMAN2!

Starting the Codec Engine

There's a lot to be said for moving into a house that has already been built. There are some advantages to having a fully custom home built, but time and cost are usually not among them. With some luck you may find a house that's about right for your needs, and then when you move in, you can put in the finishing touches. TI offers a pre-built, ready-to-use multimedia software framework called *Codec Engine* (CE). It's ideally suited to run on both OMAP and DaVinci platforms.

Although most OMAP and DaVinci devices consist of both an ARM and a DSP processor, some exceptions do exist in which parts may have only a DSP or an ARM core. The good news is that your application — if it's based on Codec Engine — will port easily between these different Silicon platforms.

The Codec Engine builds on everything that we've already discussed in this book. It builds on the DSP/BIOS kernel running on the DSP core, and it uses the XDAIS Framework Components which in turn instantiate and run XDAIS/XDM compliant codecs. The Codec Engine uses the DSP/BIOS Link discussed later in this chapter for inter-processor communications, and it leverages the concepts from Real Time Software Components (RTSC) discussed in Chapter 3. Figure 4-1 shows the component blocks of Codec Engine.

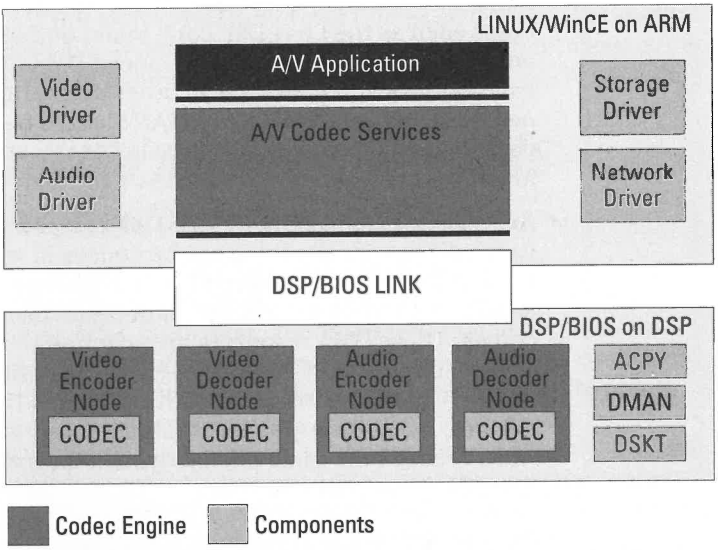


Figure 4-1: Codec Engine builds on other components available from TI.

Codec Engine is currently supported on Linux, WinCE, and Green Hills Integrity. Other operating systems will likely be added in the future.

VISA — Video, Imaging, Speech, Audio

Key to the ultimate abstraction in Codec Engine are the VISA APIs. VISA stands for *Video, Imaging, Speech, Audio.* These APIs allow the high level application developer to instantiate and run various codecs without knowing the specifics of the codec interface, other than the class of codec to which it belongs. The Codec Engine infrastructure allows a codec to be easily substituted with another one from the same class with very little effect on the application developer. Such seamless substitution is what abstraction is all about!

Each VISA codec is either an encoder or a decoder. There are separate APIs for encoding and decoding, giving rise to a total of eight API classes:

1. VIDENC
2. VIDDEC
3. IMGENC
4. IMGDEC
5. SPHENC
6. SPHDEC
7. AUDENC
8. AUDDEC

Like other APIs, some of these have already been updated to a later version like VIDENC2. In a new design, always use the highest version number. Newer APIs for Analytics and Transcoding have also recently been added.

Within each class mentioned here, there are four APIs:

1. xxx_create()
2. xxx_control()
3. xxx_process()
4. xxx_delete()

These four interfaces provide a standard way for each codec class to be used. Initially the system ***creates*** the codec by preparing and loading program and data memories in the appropriate places and readying the codec to operate. The codec will then receive one or more blocks of data to ***process*** and pass the results back to the client. Usually then, ***control*** parameters can be passed to alter the operation of the codec. Finally, when the codec is no longer needed, it can be ***deleted***, freeing processor and memory resources for other processes.

But wait — there's more! If you call in the next 30 minutes, you'll get access to one of the coolest features offered by the Codec Engine: abstraction. Okay, kidding about the phone call, but in the following section, we do explain why the abstraction capabilities of Codec Engine are so cool.

Abstracting performance with Codec Engine

As we mention in Chapter 1, hybrid cars have two types of engines: One is powered by electricity, and the other by gasoline. Imagine a hybrid with two accelerators on the floor — one for each engine — and that it's up to the driver to figure out how much pressure to use on each pedal to make the car accelerate and cruise at the right speed. Confusing! That particular hybrid probably wouldn't sell very well. Of course, real hybrid cars don't have two accelerators. They have one accelerator pedal, and the systems built into the cars employ the concept of *abstraction* to figure out how and when to deploy either or both engines.

In the land of OMAP and DaVinci processors, not only are there two heterogeneous cores, but there are also two parts to the Codec Engine. One part resides on the ARM processor, and the other resides on the DSP. This enables the ultimate performance abstraction. A high level applications developer running Linux on the ARM processor (for example), can create an instance and run very powerful codecs without having to worry about how and where the codec actually runs. In an ARM/DSP system in which the codecs require lots of processor cycles, the codec processing will likely be done on the DSP and attached hardware accelerators. However, when running a codec that requires limited processor cycles, it might be possible to run the codec on just the ARM core (also true when there is only an ARM and no DSP core).

As long as the application developer can provide a steady stream of data for the codec to operate on, and as long as there's enough available processing *MIPS* (millions of instructions per second) to run the codec, then Codec Engine takes care of pretty much everything else. Rather than worrying about which processor to address and detailed things such as inter-processor communication, address translation, and cache management, the application developer can focus on

the application itself and can just assume that the plumbing isn't going to leak, the electricity will stay on, and the wood beams will stay termite free.

Reviewing the Codec Engine process

In this section, we walk through the various steps and procedures that have to occur to make abstraction possible in Codec Engine.

- **Step 1: Cool algorithms.** Codec algorithms can either be off-the-shelf or homemade, but either way they need to implement XDAIS and XDM (refer to Chapter 3) for everything to integrate nicely. As we mention in Chapter 3, there are tools (like QualiTI) that help verify compliance with the standards, and you'd be wise to use them. You can't move forward if your algorithms won't function correctly in the integrated system.
- **Step 2: Codec Engine Server.** In order for the Codec Engine to support the notion of remote codecs (that is, codecs running on another core), there needs to be a server. This server combines the core codec along with the other infrastructure pieces (DSP/BIOS, Framework Components, DSP/BIOS Link, and so on) and ultimately produces an executable that is callable from another core. It's possible to combine more than one codec together into various combinations. However, it's critical that someone evaluates system resource requirements (MIPS, memory, DMA, and so on) to make sure that the codec combination can co-exist and run as required by the application. It's also possible to build multiple servers for a given application, for example if the application has different profiles (such as a "player" server and a "recorder" server).

  You may see a combination of codecs referred to as a Codec Combo. TI spent a lot of time coming up with that name!
- **Step 3: Codec Engine Integration.** A system integrator will create one or more engine configurations. These configurations will include the names of the engines, the codec or codecs included in those engines, where the codecs will run (usually the DSP, but there can be exceptions), and the name of the server image if the engine includes remote codecs.

- **Step 4: Codec Engine Application.** The application leverages Codec Engine APIs provided by Codec Engine to create and delete engine instances; create, delete, and interact with codecs; acquire data buffers for the operation of the codecs; and more. The Codec Engine itself doesn't perform I/O; that's left to the top-level application to implement.

DSP/BIOS Link

As you put together a software system on a multi-core SOC, you must consider the "minor" issue of how the processors are going to communicate with one another. Traditionally, developers made in-house inter-processor communication (IPC) schemes to solve this problem. About ten years ago, TI recognized this trend and produced an inter-processor communication component aptly called DSP/BIOS Link. Sometimes you see it referred to as BIOS Link, DSP Link, or simply Link. Figure 4-2 illustrates the basic component blocks of DSP/BIOS Link.

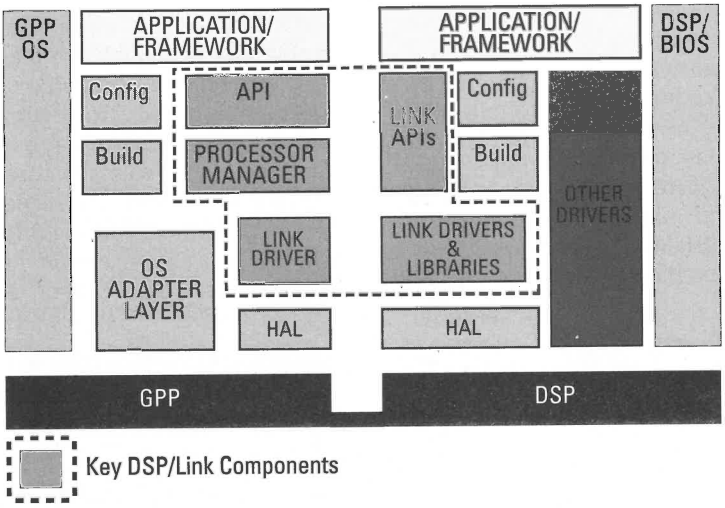


Figure 4-2: DSP/BIOS Link provides a standardized link between processors.

Why use DSP/BIOS Link?

DSP/BIOS Link is like the plumbing in your house; its function is critical, but nobody gives it much thought as long as everything is working correctly. You won't earn any bonus points for spending (wasting?) time building new inter-processor communication schemes when DSP/BIOS Link is available and ready to use. The primary advantages of using DSP/BIOS Link are that it:

- Serves as a generic API interface to applications, thus providing portability of those applications.
- Provides a hardware abstraction layer between the applications and the underlying hardware, also enabling portability between hardware platforms.
- Multiple RTOS ports of DSP/BIOS Link (as well as a porting kit) are available, meaning less repetitive work when moving a Link-based application from one OS to another.
- Like its cousin — DSP/BIOS —is scalable so that only the modules required by the application are linked in.

What does DSP/BIOS Link offer?

Your home's plumbing system provides several key services, such as supplying fresh water where needed and draining away waste water safely and completely. DSP/BIOS Link provides important services as well, and they're divided into three groups:

- **Basic processor control.** This service allows the ARM processor to attach itself to the DSP, loads the DSP with code, starts and stops the DSP, and detaches itself from the DSP.
- **Inter-processor communication protocols.** These provide for data transfer between the processor cores.
- **Inter-processor communication building blocks.** The basic processor control and inter-processor communication protocols build upon the inter-processor communication building blocks. The building blocks are made available separately for framework writers to develop their own protocols.

Here's a more detailed list of services provided by the DSP/BIOS Link:

- PROC: Basic Processor Control
  - GPP process attaches to DSP
  - GPP loads DSP with DSP executable
  - GPP starts DSP
  - GPP stops DSP
  - GPP process detaches from DSP
- Inter-processor Communication (IPC) Protocols
  - MSGQ — message queue
  - CHNL — data streaming based on an issue-reclaim model
  - RingIO — data streaming based on a circular ring buffer
- Inter-processor Communication Building Blocks
  - POOL — memory manager
  - NOTIFY — interrupt abstraction and de-multiplexing for event notification
  - MPCS — Multi-Processor Critical Section for mutually exclusive access to shared objects
  - MPLIST — Multi-Processor doubly linked circular list
  - PROC_read/PROC_write — read from or write to DSP memory

Picking the Right Multimedia Codecs

The really great thing about the OMAP and DaVinci family of devices is the almost infinite number of combinations of codecs and algorithms that can be run either consecutively or simultaneously. In this section, we explore what's available, different sources of the codecs, and things to look for when you go shopping.

Codec categories

The four most significant classes of codecs are video, audio, speech, and imaging. You may see references to VISA (an application API that addresses these four key classes of codec). It's important to note that there are other newer categories such as video analytics. There's also a myriad of niche-type signal-processing algorithms and an almost endless list of possibilities for buying and creating proprietary algorithms.

Here's a partial list of popular codecs available to run on OMAP and DaVinci class processors.

- **Video:** MPEG2, MPEG4, H.263, H.264, WMV9, DivX
- **Imaging:** JPEG
- **Speech:** G.711, G.723, G.726, G.729
- **Audio:** MP3, WMA8, WMA9, AAC, MPEG1 L2,

For updated complete lists visit `www.ti.com/digitalmediasoftware`.

Places to acquire codecs

You can acquire codecs in several ways. Two key sources are TI itself, or a member of the TI Developer Network often referred to as an Authorized Software Provider (ASP). When picking between suppliers, consider the following factors:

- Cost, both up-front and any run-time licensing costs.
- Licensing terms and indemnification issues.
- Whether demo or watermarked versions of the codec are available for evaluation purposes.
- Performance, both subjective and objective. Note that many of these algorithms have been highly optimized for the underlying DSP core and hardware accelerators. Simply taking standard C code and using the standard C-compiler may produce a functional codec, but performance is likely to be lower than what is possible with various optimization techniques.

- Quality, both subjective and objective.
- Support and maintenance.
- Reputation and history of the supplier.

Assuming that you can come to business and licensing terms for the codec(s) that you're interested in acquiring, there are still some key technical things to look for while shopping and some very important questions to ask the supplier:

- Is the codec XDAIS/XDM compliant? If it isn't, it's critical to understand exactly why and how it isn't. Do this before proceeding so that you don't spend the rest of your living years in an integration nightmare.
- Performance numbers. Verify under what conditions the algorithm was tested in order to achieve the claimed performance numbers. Over the years, we've seen all kinds of "amazing" claims that weren't quite true!
- Has the codec been tested and benchmarked in a Codec Engine–type environment, preferably on a test platform on which you can also verify the performance?
- Has the codec been tested in a real system? By this, we mean does the codec operate properly when all the other sub-systems are running at full-speed and the processor(s) is close or actually fully loaded.

Chapter 5

Picking the Right Development Tools

In This Chapter

- Considering TI Evaluation Modules
- Delving into the Digital Video Software Development Kit
- Selecting OS tools for the OMAP and DaVinci ARM and DSP processors
- Exploring ARM-DSP Interactions

We realize that a lot of the chapters in this book start out by discussing hybrid cars, overly sweet cereals, and leak-free plumbing. Or, is it overly sweet cars, leak-free cereals, and hybrid plumbing? Hmm . . . In any case, in this chapter, we introduce a convenient software bundle called the Digital Video Software Development Kit (DVSDK). DVSDKs contain all the software components, introduced in Chapters 2, 3, and 4, to allow demo applications to run right out of the box. But, at some point you'll want to add your own differentiated content to the project. Read on.

In the upcoming sections, we introduce the OMAP hardware Evaluation Module (EVM), the tools available for programming and debugging the ARM processor, the DSP processor, and some specialized tools for seeing what's going on between the two processors.

In Part II of the book, you actually get to see and hear a demo application consisting of an MPEG4 video decoder and an AAC audio decoder.

Introducing TI Evaluation Modules

Evaluation Modules (EVM) may be a fancy term tossed about by TI, but they do exactly what the name implies: they're ideal platforms for evaluating hardware and getting your initial software up and running. EVMs aren't designed to be pretty, but they are functional. Typically, if the device supports a particular type of I/O, then that peripheral is "pinned out" to the edge of the board so that users can experiment with the interface. The boards aren't cramped, so there's room to poke around with oscilloscopes and logic analyzers.

Here's what you'll find included in an OMAP EVM bundle:

- Hardware:
  - OMAP35xx Processor Target Board
  - Touch Screen LCD display and stylus
  - Power Cords (US & EU)
  - Serial Cables
  - Power Supply
- Connectivity:
  - Daughter card connections to most peripheral interfaces
  - Ethernet, USB 2.0, SDIO, I2C, JTAG, Keypad
  - S-Video output
  - SD/MMC
- Software:
  - OMAP3530 Linux Support Package (LSP)
  - Sourcery G++ evaluation tools from CodeSourcery

TI usually makes the board layout design available so customers can copy some of the trickier techniques required to connect high speed memories or I/O directly to their OMAP and DaVinci devices. Part II of this book gives you a tour of the OMAP 3530 EVM, similar to the one shown in Figure 5-1.

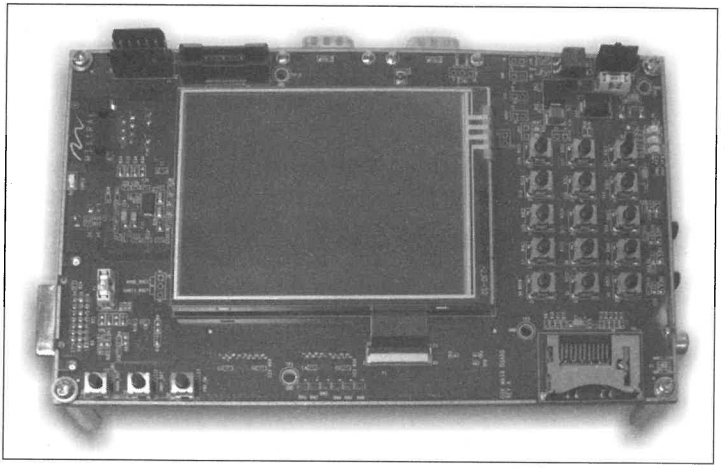


Figure 5-1: This is the OMAP35xx EVM.

Digital Video Software Development Kits (DVSDK)

Evaluation modules (see the previous section) help you evaluate hardware. The software equivalent of an EVM is called the *Digital Video Software Development Kit* (DVSDK). It's a complete software bundle that allows you to quickly get applications up and running on the EVM we describe earlier. We introduce most DVSDK elements in Chapters 2, 3, and 4 of this book. The DVSDK typically includes

- **Linux Support Package (LSP):** Kernel base port plus peripheral driver set (see Chapter 2)
- **XDAIS developers kit:** XDAIS standard including the XDM digital media interface extensions (visit Chapter 3)
- **Evaluation codecs:** These run on the DSP and hardware accelerators (See Chapter 4)
- **Codec Engine:** Framework for creating and interacting with multimedia codecs (see Chapter 4)
- **Linux utilities:** A set of useful utilities to enhance the base Linux PSP
- **"Decode" Demo applications:** Illustrates usage of the Linux drivers and codecs

- **File-based "Encode" example:** Shows basic usage of video encoders
- **A/V data:** Contains A/V clips and other data files needed by the demo applications
- **Digital Video Test Bench (DVTB):** A simple test fixture for testing basic codec operation (see Chapter 10)
- **DSP/BIOS:** A ready-to-run DSP scheduler (covered in Chapter 2)
- **DSP/BIOS Link:** A component for interprocessor communication (see Chapter 4)
- **Documentation:** Fun stuff to read

Unlike the EVM, which you're not likely to ship in a product, you can reuse and ship many DVSDK components in a final product.

Picking ARM Processor OS Tools

At some point, you need to start development and debugging of your application on the ARM processor. Your choice of development and debugging tools is often driven by the embedded operating system you choose for the ARM processor. Currently, the most common operating system choice for OMAP and DaVinci processors is Linux. But, support for other operating systems and tools is also available. The next couple of sections outline the development and debugging tools available for various operating systems.

Choosing tools for Linux applications

In Chapter 2 we discuss the pros and cons or choosing between a community open source version of Linux versus commercially available Linux versions. OMAP and DaVinci devices generally have support from both camps. In this book, we focus primarily on a community open source version of the Linux kernel (in Part II we show an open source implementation on an OMAP 35xx system). Our preferred set of support infrastructure tools is Sourcery G++, provided by CodeSourcery.

Sourcery G++ from CodeSourcery is a complete software development environment based on the GNU Toolchain. Sourcery G++ includes the GNU C and C++ compilers, the Eclipse IDE, and other tools. Sourcery G++ for ARM supports the OMAP family of processors, runs on GNU/Linux and Windows, and targets GNU/Linux, uClinux or EABI (bare board) system A command line only version of these tools is available for free. Visit `www.ti.com/dummiesbook` for more information.

Commercial Linux development and debugging tool support is provided by MontaVista. The MontaVista Pro product includes a fully tested open-source Linux kernel ported to OMAP and DaVinci. It also provides a fully integrated development tool chain for the particular kernel version you select.

You pay a premium for a commercial package, but there are benefits. The commercial provider does a lot of integration and testing that you may have to do yourself with a community version. Additionally, commercial products are often several releases behind the latest git version.

Selecting tools for other operating systems

No one says you have to run embedded Linux on OMAP and DaVinci devices. Alternative operating system tools fall into two categories. The first category provides a base port of the OS to the ARM processor, and also a port of Codec Engine and DSP/Link to the operating system. This gives you easy access to all the cool stuff we describe in Chapters 2, 3, and 4. There are currently two operating systems that fall into this first category:

- Microsoft WinCE (supported by BSquare) integrated with the tool suite Microsoft Platform Builder. For more information, visit `www.bsquare.com` and `www.microsoft.com`.
- Integrity from Greenhills Software integrated with the tool suite Greenhills Multi. For more information, go to `www.ghs.com`.

The second category of alternate operating systems provides just a base port of the OS to the ARM processor, but no direct support for Codec Engine and DSP/Link. Because this second category is so dynamic, it would be hard to list all possible operating systems here. Instead, we suggest that you contact your favorite operating system company and ask them about support for OMAP and DaVinci devices. Perhaps you can convince them to do a port of Codec Engine and DSP/BIOS Link to their OS and move themselves into the first, better-supported category!

Tools for the OMAP and DaVinci DSP Processor

TI provides multimedia codecs and a complete software infrastructure on the DSP. This means you don't have to directly program or debug the OMAP and DaVinci DSP. However, there will be certain applications and developers who want to, or need to, get more deeply involved with DSP development. If you're one of them, you'll need a set of DSP development and debug tools. Code Composer Studio (CCS) is an Integrated Development Environment (IDE) provided by TI to help with DSP development. For more details, visit `www.ti.com`.

Even though CCS is ideally suited for DSP development and debugging, you can also use it for ARM development and debugging on OMAP and DaVinci devices.

Because DSP development and debugging is different than on an ARM processor, CCS has been optimized specifically for DSP program development and debugging. The basic feature set of Code Composer Studio includes

- **IDE:** Integrated editor, project manager
- **Debugger:** Debugs the DSP and provides data visualization, cache visibility, and robust host-to-target connection
- **Real-Time Debug:** Gives non-intrusive memory access and handles interrupts while halted
- **Advanced Event Triggering:** Provides watchpoints, event sequences, and non-intrusive counters

- **Simulation:** Includes cycle accurate simulation and code coverage
- **Code-Generation Tools:** Offer industry-leading performance and program-level optimization
- **Profiling:** Profiles functions and loops, and measures cache activity and pipeline stalls

Unlike the ARM development tools described earlier in this chapter, CCS focuses heavily on an emulation-based debugging approach. Emulation leverages on-chip hardware specifically designed to assist with debugging. Among other things the built-in emulation hardware can scan register sets, memory locations, and other states, and then report back to the development host for display and analysis. Usually, an *emulator* (a small hardware "box") connects between the target board and the development host, serving as an interface between the two. This kind of "peeking" and "poking" around the chip can often be performed while the chip is running the application, meaning that debugging has little or no effect on the application itself.

What's Going on with ARM-DSP Interactions?

In the good old days of discrete processor designs, it was relatively simple to find out what was going on between two different processors in a system. Now that both processor cores are packed into a single device, it's much harder to see what's happening. Fortunately, TI provides a solution. It's called *Data Visualization Tools* (DVT), which operate independent of any other tools; you can think of DVT as a software logic analyzer. DVT relies on two key concepts:

- **"Breadcrumbs."** In Chapters 2, 3, and 4, we introduce components such as DSP/BIOS, DSP/BIOS Link, and Codec Engine. Scattered throughout these components is the ability to drop "breadcrumbs" along the execution path. These breadcrumbs record events on the ARM, on the DSP, or between the two. These informational breadcrumbs are swept up by the software infrastructure and passed to the host development environment, most likely

via an emulation channel (introduced in the previous section). All this dropping and sweeping of breadcrumbs happens in the background so it doesn't disturb the operation of the system, even when it's running at full speed.

- **Visualization.** Execution information stored in the breadcrumbs isn't very useful if you can't see it. The host development PC provides dedicated visualization windows to display this information to the developer in a useful, often graphical way. Figure 5-2 shows an example. The visualizations are great tools for watching messages go back and forth between the processors. You can also see who's interrupting who, who's waiting for what, or who's grabbing what resources.

Even if the system isn't actually broken, DVT provides a great way to tune your system for maximum performance by discovering and removing bottlenecks that could throttle (in the wrong direction) performance.

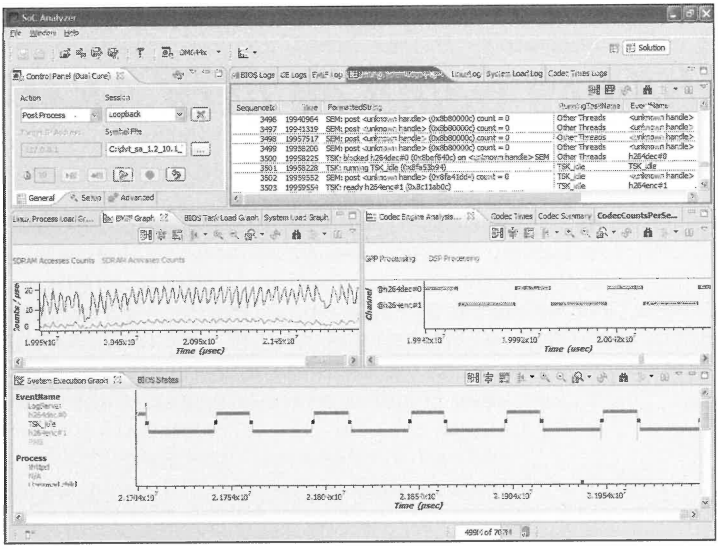


Figure 5-2: DVT visualizations help you see what's going on.

Part II

Building Something Real — Now!

In this part . . .

Okay, it's time to build something real — so boot up your computer and get ready to start.

In this part, we lead you on a short journey from getting "Hello World" to run on the TI development board to a final application that can decode and display video while also playing audio. Your kids will be so impressed that you not only know what a multimedia player is, but you also built one between lunch and dinner.

Chapter 6

Meet the Board!

In This Chapter

- Saying hello to the OMAP3530 EVM
- Building an OMAP or DaVinci program
- Working with the decode demo

It's time for you to meet the board. "Meet the board"? You're probably thinking we mean the folks upstairs in grey suits who never quite understand people like us engineers. But, we're not talking about *that* kind of board! We're talking about the OMAP 3530 board, the Evaluation Module you work with throughout Part II of this book.

If you're currently staring at a different board such as the DM6446 or DM6467 EVM, don't despair. The DVSDK is a consistent bundle of components — you'll find Codec Engine (CE), RTSC Tooling, codecs, drivers, and a standard set of demos. The programming model and design flow applies equally to all GPP (for example, ARM) and DSP platforms.

In this chapter, you start doing real work. In fact, if you follow along through the remaining chapters in this Part, you'll end up with an MPEG4 or H.264 video decoder running alongside MP3 or AAC audio decoders. In other chapters of this book, you see how to build demos from bare deck codecs by leveraging various tools, helper wizards, and sample applications. We hope these chapters will make you a DVSDK guru capable of building multimedia applications at the board's whim. (This time we *do* mean the folks upstairs in grey suits!)

Welcome to the OMAP3 EVM

Chapter 5 introduces you to what's on the OMAP 3530 board. Now, take a look at Figure 6-1 to see which parts you use in this book.

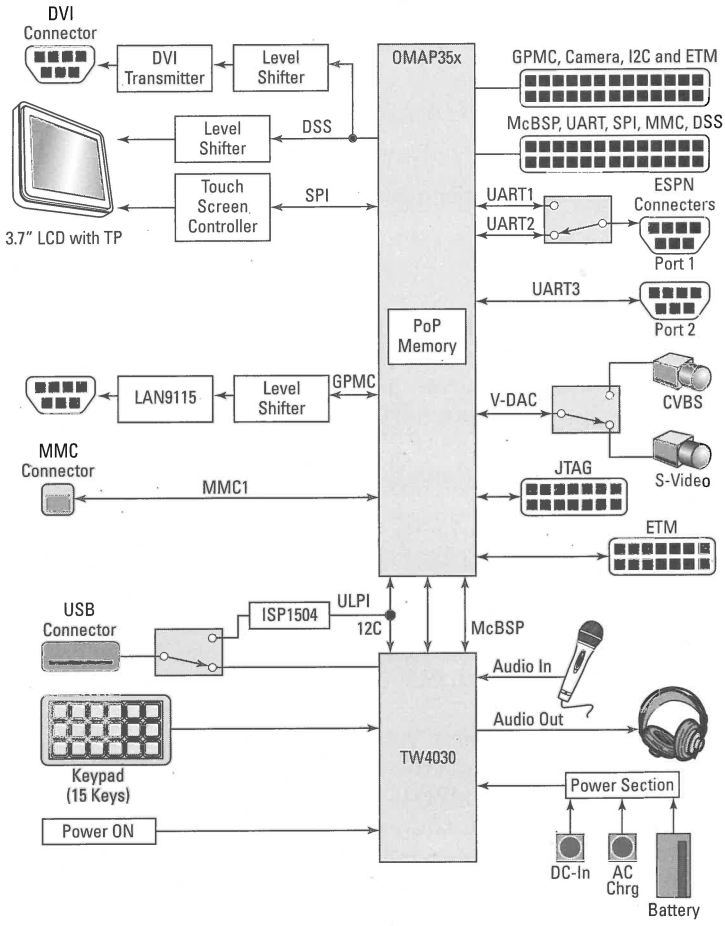


Figure 6-1: These are the major hardware components of the OMAP 3530 EVM.

In this book, we focus on creating video and audio applications. That means you output decoded video streams to the onboard LCD and optionally a flat panel LCD computer display over Digital Visual Interface (DVI). You also use audio inputs and outputs via a McBSP-based codec Platform Support Package (PSP) driver. You may also play with the keypad to start, stop, and toggle between video streams to decode.

The OMAP 3530 EVM is a powerful beast. It offers

- 600Mhz Cortex ARM A8 for applications programming
- 360Mhz c64+ DSP for multimedia acceleration
- Graphics support primarily via the Open GL ES 2.0 API

Texas Instruments baselines off the Linux 2.6 kernel and contributes back to the community "git tree" (refer to Chapter 2). All of the PSP drivers run on the ARM Cortex-A8 core under Linux. These drivers are provided in source form to facilitate porting to your hardware. The advantages of following the latest kernel tree are numerous; you can pick up new features others have contributed, and it's easier to apply patches. Wherever possible, the PSP drivers implement the standard Linux interfaces, such as Advanced Linux Sound Architecture (ALSA) for audio and Video 4 Linux 2 (V4L2) on the display subsystem.

"Hello World"

Your first adventure in OMAP and DaVinci programming is to build and run a program that simply says, "Hello world." This may sound strange or simplistic if you've read about all the advanced features we describe in this book, but you have to start somewhere.

This chapter assumes you've installed the software from each of the CDs in the DVSDK. Before you start building a program, you should browse through TI's "Getting Started Guide (SPRUFZ7)" that came in the box. That means you have the Linux source and binaries for the OMAP35x platform and all necessary tools, such as the CodeSourcery GNU build

environment. The target file system should boot the pre-built Linux kernel image using your preferred method, such as TFTP boot with NFS file system. The supplied serial cable should be connected from UART 1 to your PC, and a terminal emulator such as Teraterm (Windows) or minicom (Linux) should show a successful boot sequence.

To get started, move to your Linux host system (under VMWare or a true Linux box), and perform the following steps.

1. **Make sure you're logged in under the username *user*, not *root*.**
2. **Create a directory for the earth-shattering "Hello World" program by typing**

```
[>] mkdir -p ~/workdir/filesys/opt/hello
```

3. **Change to the new directory by typing**

```
[>] cd ~/workdir/filesys/opt/hello
```

4. **Create a file called hello.c with the following contents:**

```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("This is OMAP3530 saying Hello
World!\n");
    return 0;
}
```

5. **Build the program by typing**

```
[>] arm-none-linux-gnueabi-gcc hello.c -o
hello
```

This command assumes that you've already added the CodeSourcery tools to your path as per TI's "Getting Started Guide (GSG)."

6. **Now flip across to your OMAP3530 target terminal emulator.**

You're not on the Linux host at this point. You're on a terminal within the target file system. In our examples here, we use a $ sign instead of a > to denote the target system as opposed to the host.

7. **Change to the sample program directory by typing**

```
[$] cd /opt/hello
```

8. **Type** [$] ./hello **to run the program.**

When you run the program, you should see output that looks like this:

```
This is OMAP3530 saying Hello World!
```

Running the Decode Demo

If you want to experience the true power of the OMAP 3530 platform, you can do so using the decode demo. Each DVSDK platform supplies at least three demonstration applications:

- **decode:** Audio and video decode typically displays to a monitor or EVM LCD display.
- **encode:** Audio and video encode input from a microphone and camera.
- **loopback:** This application encodes and decodes audio and video.

Each of these applications uses the multimedia codecs, Codec Engine, PSP drivers, RTSC tooling, and other key content provided in the DVSDK.

In this section, we show you how to get the decode demo running. In Chapter 10, we show you how to rebuild the demos to leverage a new DSP server executable that can be built by following Chapters 7 through 9.

The demos can be run either standalone or via the command line. In standalone mode, you don't need a connection to a Linux host system. All interaction (Start, Stop, Pause, and so on) is done using the keypad interface. For example, key S17 is designated as "Play" by the GSG.

We use command line mode for now because that's what you use when you build and run other multimedia applications in this book. Follow these steps:

1. **Make sure your OMAP3530 LCD display is turned on and that speakers or headphones are connected via the P9 connector on the board.**

   The demos, the partner DSP executable, and a bunch of audio and video clips should already be on the target file system.

2. **In Teraterm or minicom, execute**

```
[$] cd /opt/dvsdk
```

   Before you run the demos from the command line, various Linux kernel modules must be loaded. These should already be loaded via the boot script.

3. **To confirm that the correct Linux kernel modules are loaded, type**

```
[$] lsmod
```

   At a minimum, you should see dsplink and the CMEM Contiguous Memory Allocator modules loaded.

4. **Inspect the supported command line options by calling up the Help menu:**

```
[$] ./decode -h
```

5. **Pick one encoded video and one encoded audio clip to decode:**

```
[$] ./decode -v
./data/videos/davincieffect_ntsc_1.264
-a ./data/sounds/davincieffect.mp3
```

You should see a decoded H.264 Base Profile DaVinci Effect clip on the EVM LCD display, accompanied by MP3 audio output through your speakers. The terminal also reports some useful statistics such as bit-rate and CPU load percentage of both the ARM and DSP.

Now that you know how the DVSDK contents work, you're ready to start building your own multimedia applications. In Chapters 7 through 9, we walk you through the complete flow of designing and building ARM plus DSP systems.

Chapter 7

Making Codecs Play Nice with Rules and Guidelines

In This Chapter

- Checking codecs using the QualiTI tool
- Reviewing common XDAIS rule violations
- Working with XDM classes

When a car manufacturer develops a new type of vehicle — a hybrid-powered car, for example — it has to make sure that the new design will be compatible with existing highway infrastructures. The car can't be too wide for traffic lanes and should be capable of maintaining normal highway speeds. Likewise, the new car must comply with government safety regulations; if the manufacturer follows the rules and guidelines of proper automotive design, the car won't be a hazard to other road users and buyers will find it useful.

Car designers aren't the only engineers who have to follow rules and guidelines. Codec producers must play by the rules too! Without rules and guidelines, programming chaos ensues and systems may crash when someone else adds a new codec or feature. This chapter shows you how to spot-check codecs for integration success. Also in this chapter, we pre-check an MPEG4 decoder, which you get to run on our OMAP3530 EVM in later chapters.

Keeping Codec Producers Honest with the QualiTI Tool

What ensures that you don't go barreling down the road at 100 mph in your shiny new hybrid? Two things: The police department, and your own self-preservation instincts.

What ensures that your shiny new codec doesn't barrel destructively through the devices in which it's installed? Most police officers are too big to fit inside silicon devices, and your job-preservation instincts can't guarantee against every mistake. But, there is a handy tool called QualiTI that checks your codecs for XDAIS standard compliance. QualiTI is included with the XDAIS developers kit (refer to Chapter 3).

A sample codec to validate

To show you how QualiTI works, we use it to inspect the codec described in Table 7-1. The TI video codes are either watermarked with the TI logo or incorporate a timeout feature to distinguish evaluation from production releases.

Table 7-1	Sample Codec
Codec Name	M4H3DEC (mpeg4 decode)
Vendor	TI (Texas Instruments)
Version Number	2.00.003
Validated on Platform	OMAP3530 EVM
XDAIS compliant?	Yes
XDM Interface version	IVIDDEC2

The small amount of data in Table 7-1 actually says quite a bit. Key pieces of information include

- **Codec Name** and **Vendor**. The name "M4H3DEC" and vendor "TI" are prefixes on symbol names to ensure that it doesn't clash with other codecs. We've seen systems

with as many as 26 codecs, and if everyone called their main processing function "doIt," we'd have a slight problem! See the section "Rules 8, 9, and 10: Namespace compliance" later in the chapter.

- **Validated on Platform**. Specifying OMAP3530 EVM indicates that the performance and footprint-size numbers of the codec were validated on that platform.
- **XDM Interface Version**. The algorithm has implemented the IVIDDEC2 interface.

Getting set to run QualiTI

The data in Table 7-1 claims that our MPEG4 decoder is XDAIS-compliant. How can you be sure? Run the QualiTI tool. QualiTI is available as part of XDAIS 6.10 or greater, and you can get it at:

```
www.ti.com/dummiesbook ->XDAIS Developers Kit &
QualiTI
```

After you've installed QualiTI on your computer, you can find it in the following directory tree:

```
xdais_6_20/packages/ti/xdais/qualiti
```

The "packages" concept is something you see more of throughout this Part. All Target Content and many development tools sit in packages. Packages are TI's embedded equivalent of supermarket cereal packs because they help keep things organized.

QualiTI has a couple of prerequisites that must be installed before you can run it. They are

- XDCtools v3.00 or greater
- Code Generation Utility Scripts v1.20 or greater

You can find both at `www.ti.com/dummiesbook`.

These scripts and executables do the grunt work of checking codecs for rules compliance. For example, the sectti utility script parses the object files in a decoder, checking for problematic section names. It also provides a neat-and-tidy footprint report.

Running QualiTI

You're now ready to run QualiTI. Here's how:

1. **Launch the QualiTI program.**

   As described in the previous section, you can find QualiTI in the directory:

   ```
   xdais_6_20/packages/ti/xdais/qualiti
   ```

2. **Tweak the startup file xdais_6_20/startqti.bat (or startqti.sh on Linux) to point to your XDC tools installation, and then execute it.**

3. **Enter the module name and vendor, as shown in Figure 7-1.**

   In Figure 7-1, you're checking the same codec mentioned earlier in this chapter.

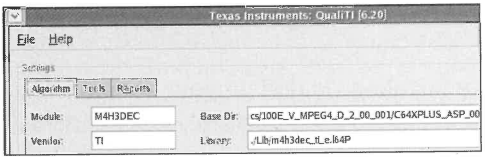


Figure 7-1: Enter the module name and vendor.

4. **Point QualiTI to the base directory of the codec and the library archive itself.**

5. **Switch to the Tools tab and set the paths to the prerequisite tools, as shown in Figure 7-2.**

   As described in the previous section, the XDC Tools and Code Generation Utility Scripts are prerequisites for QualiTI.

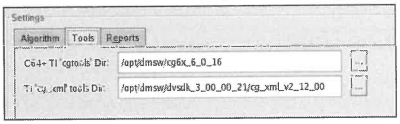


Figure 7-2: Enter the paths to the codec's directory and library archive.

6. **Click Run and dive for cover!**

Okay, we're just kidding about running for cover. Ideally, you should soon see the "Test passed" message under Status Details, as shown in Figure 7-3. If you see this message, it's time to celebrate with a coffee break. But, if the codec doesn't pass, cancel the coffee break and review the list of broken rules. The Status column under Rules indicates which rules were broken.

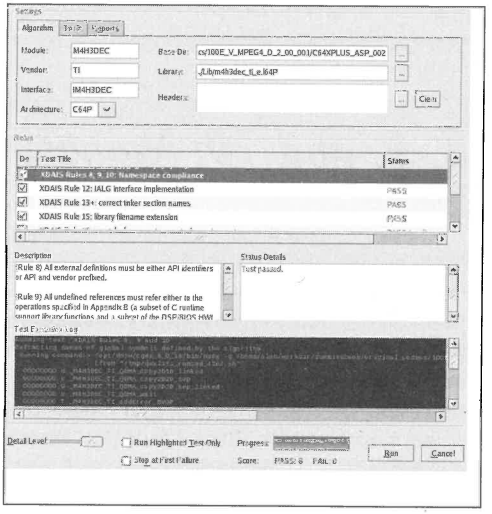


Figure 7-3: The codec passed QualiTI's test!

QualiTI provides a useful sanity check on your codecs. However, you may notice that it doesn't check *all* the XDAIS rules. In fact, as of XDAIS 6.20, it doesn't actually *run* the codec's algorithm, hence it can't check interrupt latency or execution time numbers. Nevertheless, you can avoid most of the common system integration pitfalls if the algorithms in your system pass QualiTI.

QualiTI automatically generates an HTML report. You want to make sure that the codec vendor ships this report with its codec packages to prove that its algorithm is easy to integrate.

Diving Deep into a Few XDAIS Rules

If the codec fails QualiTI, the tool will list which specific XDAIS rules were broken. You need to know what the rules mean if you want to fix the codec. In the next few sections, we dive deep into a few of the rules and show you how QualiTI catches problems.

Rules 8, 9, and 10: Namespace compliance

What does namespace compliance mean? In a nutshell, *namespace compliance* refers to globally exposed symbols that won't clash with the next 25 algorithms you might add to your system. As described earlier in this chapter, the name "doIt" is a problem, but "M4H3DEC_TI_doIt" is not because it conforms to namespace rules. On the other hand, if "doIt" is a local (static) function, there can be lots of function names like that in the system — the XDAIS rules only care about global symbols.

Rule 13+: Correct linker section names

First question: What does the + in Rule 13+ stand for? Answer: The plus sign signifies that TI has stretched XDAIS to cover common integration pitfalls. The original Rule 13 simply required algorithm writers to mark each IALG function with a tag to enable individual linker placement. A function may have looked something like this:

```
#pragma CODE_SECTION(FIR_TI_alloc,
".text:algAlloc")
```

Marking functions in this manner was important in the days of ultra small DSPs with tiny internal memories and no cache. It allowed non-critical stuff to be pushed off to slower external memory, leaving more room for the critical functions of "doIt" (sorry, "M4H3DEC_TI_doIt"). But OMAP and DaVinci

processors all have advanced program (and data) caches, so they won't grind to a halt if Rule 13 isn't followed. The processor might, however, stall in the pit lane if you have ambiguous section names like ".tables". Thankfully, QualiTI informs you of such problems, as shown in Figure 7-4.

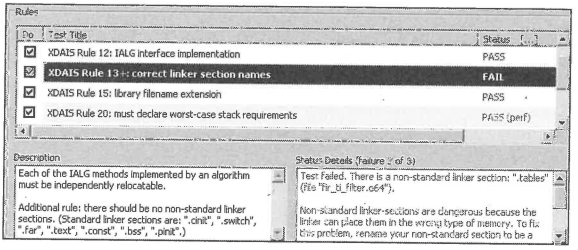


Figure 7-4: Uh-oh! Somebody broke Rule 13+.

In the sample rogue FIR filter algorithm shown in Figure 7-4, QualiTI found a nonstandard section named ".tables", so it kindly tells you *why* this name is a problem. If the section isn't explicitly placed by its section name, the linker has no clue whether the section is code, un-initialized data, or constant data. The linker may arbitrarily place the function in the wrong type of memory (that is, Program-only). Such misplacement is a frequent source of crashes and is the reason for the + in Rule 13.

Rules 21, 22: Must characterize static data and program memory requirements

The cool thing about QualiTI is that it practically does some of your job for you. QualiTI runs a bunch of scripts and produces a footprint report looking something like this:

```
================================================
REPORT FOR LIBRARY:
D:/TI_Docs/dummiesBook/mpeg4dec/100E_V_MPEG4_D_1
_10/DM644x_SP_001/./Lib/m4h3dec_ti.164P

================================================
```

```
REPORT FOR FILE: common.o
Name : Size (dec) Size (hex) Type
----------------------------------
.text : 3648 0x00000e40 CODE
.const : 15 0x0000000f DATA

REPORT FOR FILE: conceal.o
Name : Size (dec) Size (hex) Type
----------------------------------
.text : 7616 0x00001dc0 CODE
.switch : 16 0x00000010 DATA
...
----------------------
Totals by section type
----------------------
Uninitialized Data :    3736      0x00000e98
Initialized Data :      23088     0x00005a30
Code :                  154272    0x00025aa0
```

Armed with the footprint report data, plus the stack, scratch, and persistent data requirements, the system integrator can start carving up the memory between codecs.

XDM and VISA Semantics

Recall that VISA stands for Video, Imaging, Speech, and Audio. Specifically, we care about the interfaces (IAUDDEC, IVIDDEC2, and so on) it defines. Within these interface header files, various parameters, which have been generally agreed as key fields for that class, are defined. Many codecs don't need to extend these parameters, but several others do. The MPEG4 decoder used here doesn't need extensions, but TI's H.264 encoder on DM6467 (for example) needs extended parameters to tweak the Quantization Parameter (QP) values. The application sets these as follows:

```
/* Use extended dynamic parameters to allow
tweaking of the QP value */
 IH264VENC_DynamicParams extDynParams = {
 Venc1_DynamicParams_DEFAULT,
 0, /* ChromaQPOffset */
 18, /* QPISlice */
 20, /* QPSlice */
 ...
 };
```

```
....

extDynParams.videncDynamicParams.size =
sizeof(IH264VENC_DynamicParams);
...

/* Create the video encoder */
hVe1 = Venc1_create(hEngine, "h264enc",
&params,
       (VIDENC1_DynamicParams *)
      &extDynParams);
```

This code is okay. Dynamic parameters have been applied to obtain the maximum encoder quality, yet the *logic* of the code remains unchanged.

However, extending inArgs and outArgs of XDM interfaces has an enormous effect on application code since it basically nullifies the plug-and-play functionality of the codec. Custom application logic becomes necessary for codecs that extend inArgs and outArgs. If, for example, you have an AAC encoder that extends the standard AUDENC interface, you invariably end up with something like this: -:

```
#ifdef AACENC_VENDOR
// do custom code for AACENC from VENDOR
#else
// do standard processing for AUDENC codec
#endif
```

That's ugly because you have to maintain code specific to a particular vendor's codec implementation. Plug and play is gone.

Newer XDM interfaces (v1.x and higher) were introduced specifically to alleviate problems like this. Extending interfaces via DynamicParams is fine, but doing so via inArgs and outArgs requires custom application logic and is strongly discouraged.

Chapter 8

Making a Standard Box for Codecs

In This Chapter

- Appreciating the need for standardized RTSC packaging
- Simplifying package creation with the RTSC Codec Packaging tool
- Preserving all important codec performance

Have you ever wondered why shipping containers are all the same shape and size? Shipping containers are a standard-sized box. That way the folks who load them onto ships know how to deal with the containers and can keep the docks running efficiently. No matter what is being transported inside the containers, they all work the same in ships, trains, and trucks. Without standardization, docks and other transportation hubs would have a difficult time stacking the boxes.

Now imagine a customer with 26 codecs. What if each codec was packaged differently? How would you know where the key header files were? If the codec wanted to tell users how much stack size it requires, you might end up with 26 different ways to read it. System integrators want to add new codecs without thinking too much about them. A standardized box — a codec shipping container, if you will — is what's needed. This chapter shows you how to use the RTSC Codec Packaging tool to ensure that codecs can be easily plugged into the framework.

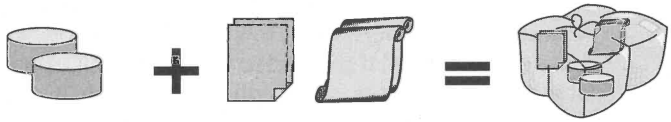
Why Bother with RTSC Packaging?

What's a RTSC package (also called an XDC package) and what goes inside it? Figure 8-1 shows the basic contents of a RTSC package. According to the official glossary, RTSC is defined as follows:

> *A RTSC package is a named collection of files that form a unit of versioning, update, and delivery from a producer to a consumer. Each package is embodied as a specially named directory (and its contents) within a file system. Packages are the focal point for managing content throughout its lifecycle. All packages are built, tested, released, and deployed as a unit.*

What does all that mean in plain English? It basically translates as, "Add a few files around your codec library to ensure the codec fits nicely in the Codec Engine framework." To find out which files you need to add, check out the next section for a list of RTSC package files.

With one or two codecs, the value of packaging may not seem apparent at first. But, as you add more content, possibly from different vendors, the benefits of a standard box become clear during system integration.



Original target content: libraries and header files

XDC files: additional information on using the content (specifications and scripts)

XDC package: an *active container* for the embedded target content

Figure 8-1: The basic elements of a RTSC package.

What's in the package?

Take a look at the MPEG4 decoder package shown in Figure 8-2. This package is shipped in the DVSDK and contains the following files:

- **app:** This contains an example application using TI's MPEG4 decoder.
- **ce:** This stands for Codec Engine. It adds a bunch of meta information to let the framework make smarter decisions. We show you how it works when we run Step 5 of the wizard.
- **docs:** You can probably guess what's in here!
- **lib:** This is the heart of the RTSC package because it contains the codec library.
- **package.bld:** This is like the key to the RTSC build system. Written in XDCscript — a superset of industry-standard JavaScript — it gives you fine-grain control over how you build your code.
- **im4h3dec.h:** There's nothing clever here, it's just the header file interface to the MPEG4 decoder.
- **package.xs:** This item is a bit more clever. It enables programmatic selection of libraries. For example, the getLibs() function may return either the production or evaluation version of the library, depending on the profile selected by the system integrator.

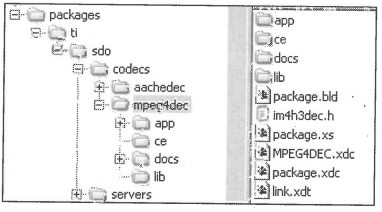


Figure 8-2: TI's MPEG4 decoder RTSC package.

- **MPEG4DEC.xdc:** This typically comprises a bunch of configuration and memory placement parameters, such as codeSection and dataSection.
- **package.xdc:** This item is small but important. This is what it looks like:

```
package ti.sdo.codecs.mpeg4dec {
    module MPEG4DEC;
}
```

  This code basically says, "I am a package. My name is MPEG4DEC, and I live in the ti.sdo.codecs.mpeg4dec namespace."
- **link.xdt:** This item is very clever! It specifies a component's contribution to the linking process. We talk more about this during the section on preserving codec performance.

What are the RTSC package benefits?

Creating the RTSC package files listed in the previous section seems like extra work. So why bother? The standard box brings at least five concrete benefits:

- No need to specify different –I"/path1" –I"/path2" codec preprocessor directives. You can have one package path for multiple codecs.
- Codec-Engine add-on methods, such as getStackSize(), enable specification of stack size (declared by the expert — the codec producer!) for optimal task stack creation.
- The linker template allows contribution of required placement directives on a per-codec basis. This means no messy cutting and pasting from example files.
- The package enables additional tooling, such as the RTSC Codec Package Wizard we use in the next section.
- When you've learned one package, you've learned them all.

Getting Help from the RTSC Codec Packaging Tool

The preceding section shows you the benefits of standardized RTSC packages. Even if you're convinced of the benefits, wouldn't a tool to generate all that stuff be nice? Enter the RTSC Codec Packaging Wizard.

Created with a similar look and feel to the QualiTI tool (described in Chapter 7), you basically point the RTSC Codec Packaging tool to your codec library, enter some basic package information (such as which XDM interface is implemented), and out pops a RTSC codec package. The payoff is easier integration with the Codec Engine.

Getting ready for the wizard

Of course, there are a few prerequisites before you can run the RTSC Codec Packaging tool. Here are the basic steps:

1. **Assemble your DVSDK content.**

   The Codec Packaging Wizard requires XDC tools, XDAIS, Codec Engine, TI c6x Code Generation Tools, and the Codegen Utility Scripts package. Thankfully all these items are included in the DVSDK, as is the wizard itself under dvsdk_3_<version_num>/ceutils_1_<version_num>.

2. **Set your XDCPATH.**

   What does this mean? The XDCPATH tells the XDC tools where to find all the packages (including Codec Engine, XDAIS, and the wizard itself), much like the Java CLASSPATH and Windows/Linux PATH for executables. There are several ways to do this, the simplest of which is to leverage the Makefile provided with the wizard. Since you're likely using the wizard from within the DVSDK, the Makefile takes advantage of the top-level Rules.make, which sets up all the paths for you.

If you're using the wizard standalone, the other ways to set the XDCPATH are specified in the Wizard FAQs in `www.ti.com/dummiesbook`.

3. **Run it!**

   Simply type

   ```
   make genpackage
   ```

   In plain English, this says, "use the part of the Makefile that relates to kicking off the Codec Packaging Wizard."

Running the wizard

The RTSC Codec Packaging Tool is easy to use because it walks you through a series of screens where you enter details about your codec package. When you execute the wizard as described in the preceding section, the first thing you see is a graphical user interface. You start at (surprise!) Step 1, as shown in Figure 8-3. The next few sections walk you through each step.

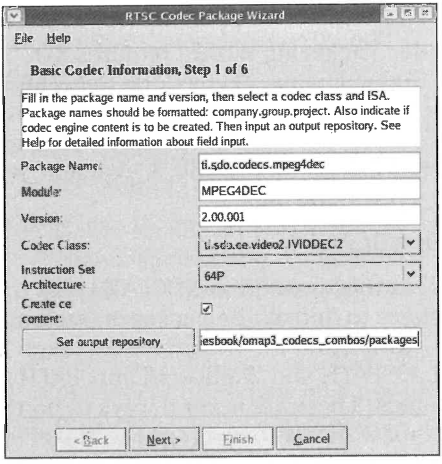


Figure 8-3: Enter basic codec info in Step 1.

You can click Help on the wizard's menu bar at any time to get context sensitive help.

Step 1

Fill in the blanks in Step 1 of the RTSC Codec Packaging Tool wizard. The package name we entered in Figure 8-3 is ti.sdo.codecs.mpeg4dec and the module name MPEG4DEC. This package name follows this convention:

```
companyname.groupname.codecs.codecname
```

It's not mandatory to follow this naming convention, but it helps ensure uniqueness so we strongly recommend it. Other items to enter in Step 1 include

- **Version.** This is simply the codec version. Again, it's not mandatory but it is helpful when looking at codecs.
- **Codec Class.** This field specifies whatever XDM / VISA interface the documentation states it implements. At the time of writing, the MPEG4DEC implements the IVIDDEC2 interface.
- **Instruction Set Architecture.** This field defaults to c64Plus, so no change required.
- **Create CE content.** This subtle checkbox is important because it ensures all the Codec Engine add-on functions are generated. These functions include the getStackSize() method for specifying stack requirements.
- **Set output repository.** This box tells the RTSC Codec Packaging Wizard where to place the output content.

Step 2

Next is Step 2 as shown in Figure 8-4. This step is pretty simple. The "watermark" is simply a boolean variable to indicate whether this is a production or evaluation codec. In Figure 8-4, we use an evaluation codec (as denoted by the "_e" in the name) so the watermark should equal True. Click Next again to move to Step 3.

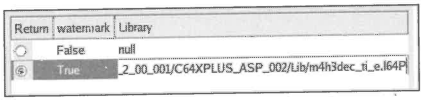


Figure 8-4: Enables the system integrator to select between an evaluation or production codec.

Step 3

Step 3 inquires about any extra directories that may be needed to complete the package. If you have header files, source files, or other items in different directories, specify them as shown in Figure 8-5. Adding the codec header file is important if there are extended parameters you need to expose.

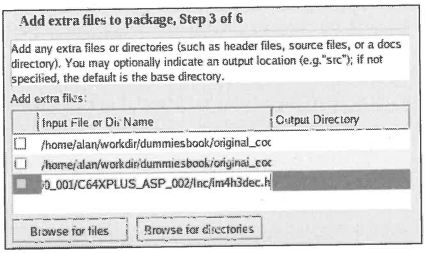


Figure 8-5: Choose extra directories and files to include in the end package.

Step 4

Click Next again in the Packaging Wizard to progress to Step 4, as shown in Figure 8-6. Step 4 brings in the prerequisite tools we highlighted earlier in this chapter. There's nothing new to download since it's all in the DVSDK, however you need to get the paths right.

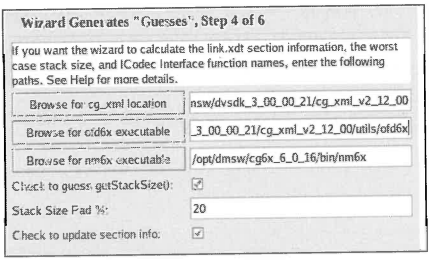


Figure 8-6: Select paths to the tools you'll need in the final steps.

Step 5

In Figure 8-7, you can see that the wizard has magically retrieved the symbol names M4H3DEC_TI_IALG and M4H3DEC_TI_IDMA3. How'd it do that? The wizard finds these names thanks to the structure inherent in the XDAIS standard. All codecs must expose a symbol <MODULE>_<VENDOR>_IALG. This is the entry point to the IALG memory interface functions. Because of this convention, the symbol listing tool nm6x (remember, we specified that in Step 4) can search for the symbol and fill in the blanks for you. The same is also true with M4H3DEC_TI_IDMA3, which is the entry point to the algorithm's requests for DMA resources.

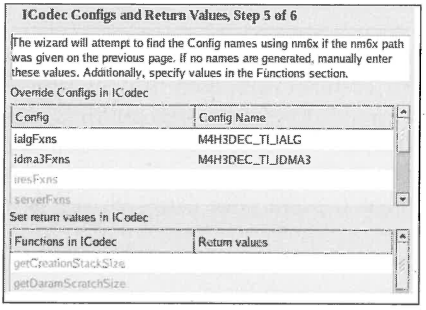


Figure 8-7: The wizard automatically finds the key symbols you need.

Codec Engine uses these entry points to step in to the underlying function tables and call the correct initialization and processing functions. The XDAIS standard and well-defined VISA interfaces make this functionality feasible.

Step 6

When you click Next to go to Step 6, you see a screen like the one shown in Figure 8-8. Fear not. You don't have to type in the code shown in the figure. The wizard generates the code for you via the ofd6x and cg_xml tools. Once again the library object files are parsed, this time to get the linker section names for the functions and data.

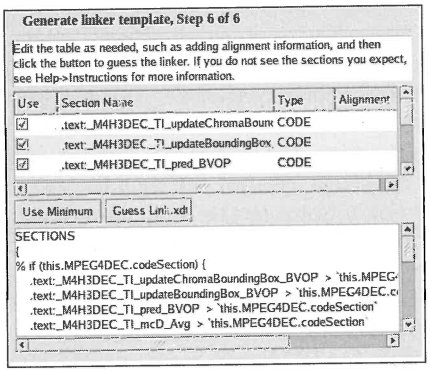


Figure 8-8: You're almost done! Enter code and data placement directives.

A clever trick in the wizard is automatic grouping of code into an MPEG4DEC.codeSection, data into an MPEG4DEC.dataSection, and uninitialized data into an MPEG4DEC.udataSection. Why care? You may not, but the linker does. If you do nothing, the linker might place *data* in a *code-only* memory type, at which point your program starts to veer off the road.

If you have 25 more codecs to package, you'd quickly get bored typing the paths each time. Hence, the RTSC Codec Packaging Wizard lets you save an XML representation of the inputs. You can use the XML as a template for the next codec, or simply import it as a baseline for modifications to existing codecs.

After you complete Step 6, click Finish and you're done. You now have a codec package that will fit nicely in Codec Engine.

Preserving All-Important Codec Performance

We can optimize the wizard output a little to gain precious cycles. How so?

RTSC provides a unique feature that can be described as: "Stick your special sauce codec memory placement in this file, and the tools will make sure it gets honored at system integration time." We've observed an H.264 encoder running on the DM6446 platform and found that memory placement special sauce can give a 10 percent performance boost. Consider the following code:

```
% if (this.MPEG4DEC.codeSection) {
GROUP :     {
    .text:_M4H3DEC_TI_decodeNextFrame
    .text:_M4H3DEC_TI_getCurRefBufs
    .text:_M4H3DEC_TI_initErrConcealObj
    ...
    } > `this.MPEG4DEC.codeSection`,align =
0x10000
%}
```

The codec vendor who wrote this code has labored to provide the optimal placement grouping of functions to yield the best possible c64+ Level 1 Program cache performance. With video codecs in particular, this can have a big performance impact.

Before RTSC, the codec vendor simply said, "Please copy and paste this special sauce into your system." But now, link.xdt and the XDC tools automate this for you.

To check the result of this placement, look in the generated .xdl file in the upcoming codec server build.

Chapter 9

Generating DSP Server Executables

In This Chapter

- Taking a look at the Codec Engine server generator
- Using configuration to tweak the knobs
- Bundling multiple codecs together in combos

In Chapter 8, we talk about the fact that shipping containers conform to certain standards to ensure they're all the same size and stack easily. But, no matter how standard the containers are, they're pretty useless unless you can find ships to put them on.

Just like shipping containers, neatly packaged codecs also conform to standards. And, also like shipping containers, you need somewhere to put your codecs for them to be useful. In the case of codecs, your ship is a DSP server executable. That's the DSP end-application that your GPP code will start and run.

The DSP server executable contains Codec Engine, DSP/BIOS, Framework components, DSP Link content, and various BIOS utility modules. In "ye olde days," the user would have to put all these components together and make the complete system from scratch. But today, the RTSC Server Wizard gives you a major head-start on packaging. In this chapter, we show you how to use the RTSC Server Wizard to bundle codecs into a Codec Engine-based DSP server application.

Timeout for a Terminology Recap

Figure 9-1 illustrates the hierarchy of codecs, servers, and engines in a system. As you study the diagram, you may notice that the TLA (that's a three letter acronym for "three letter acronym") count is beginning to creep up. Here's a recap of what's in an OMAP or DaVinci system and explanations of what all that stuff means:

- **RTSC Codec Package.** This is the RTSC package for the codec library. It implements an XDM-based interface such as video, imaging, speech, or audio.
- **RTSC Server / Combo Package.** This is a complete DSP application bundling one or more codecs together with DSP/BIOS, CE (Codec Engine), and framework components.
- **Engine.** You need something to invoke your server from the remote GPP side of the system, and that "something" is called the engine. You "open" an Engine to get a handle to that collection, then you can create instances of those codecs given the Engine handle. Quite often you have several engines in a GPP application.
- **DVTB (Digital Video Test Bench).** This is a command line interface that uses engines to get and set codec parameters and execute codecs.
- **Demos.** Demos are real-world GPP end-applications upon which you can build.

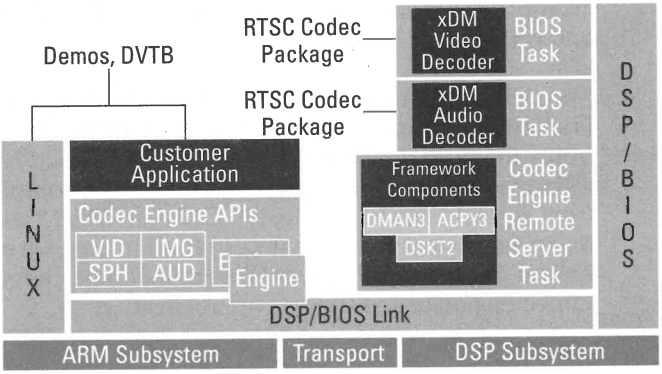


Figure 9-1: This is the hierarchy of codecs, servers, and engines in an OMAP or DaVinci system.

Getting Help from the RTSC Server Packaging Tool

The RTSC Server Packaging Tool is a little simpler than the RTSC Codec Packaging Tool described in Chapter 8 because it has just one window and a couple of boxes to fill in. What the Server Packaging Tool generates, however, is rather useful. The next couple of sections show you how to prepare for and run the RTSC Server Packaging Tool.

Getting ready for the wizard

Before you can build a server package, you need to build a codec package. (We show you how to package a codec in Chapter 8, so pay a visit to that chapter if your codecs still need packaging.)

You can build a codec package in lots of ways. The easiest way is to just type **make** from the ceutils directory location inside the DVSDK.

Sounds simple, doesn't it? Yeah — too simple! The build will fail because you haven't updated your RTSC package path (XDCPATH) to point to your newborn codec. Comment out the previous CODEC_INSTALL_DIR in Rules.make and point it to the new path as follows:

```
# Where the codec servers are installed.
CODEC_INSTALL_DIR=${HOME}/workdir/dummiesbook/om
ap3_codecs_combos
```

Now, you can build it. Type **make** from the ceutils directory and you're done. It looks like this:

```
[ceutils_1_06 >] make
```

You're not actually building the codec source code — you're just building the package "meta-data," which contains add-on wrapper functions such as getStackSize(), getLibs(), and others.

Running the wizard

The Server Packaging Wizard is bundled along with the Codec Packaging Wizard, so there's no need to spend half a day hunting for prerequisite tools. To kick-start and run the Server Packaging Wizard, follow these steps:

1. **Start the wizard by typing**

```
[ceutils_1_06 >] make genserver
```

After a few seconds, you see a dialog box like the one shown in Figure 9-2.

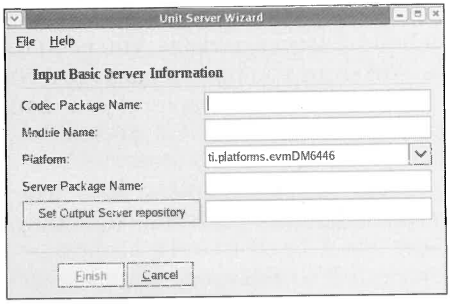


Figure 9-2: This is the server wizard in action.

2. **Type ti.sdo.codecs.mpeg4dec for the codec package name.**

It would be nice if you could just enter mpeg4dec for the codec name, but there could be lots of mpeg4 decoders out there from several vendors. Therefore, the name needs to be unique and identifiable.

3. **Choose ti.platforms.evm3530 in the Platform menu.**

As with the codec package name, the platform must be unique. A name like omap3530 wouldn't be sufficient, because Vendor X might offer a blue 3530 EVM, while Vendor Y sells a pink version. The wizard automatically fills in the rest of the fields for you, as shown in Figure 9-3.

A sure sign that you got it right is when the wizard fills in the output server repository automatically. The wizard magically (it *is* a wizard, after all!) peeks at your entries, and if it finds a package match, the wizard defaults to the repository containing the codec.

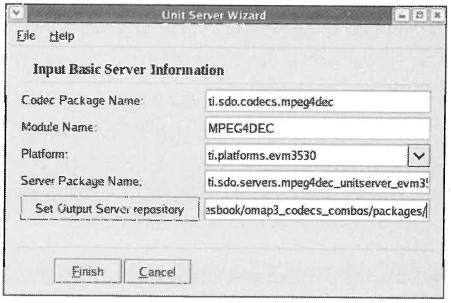


Figure 9-3: The server wizard points to the mpeg4 decoder on your OMAP3 EVM.

4. **Click Finish.**

   The wizard asks if you want to save the data as an XML file. You can pass on this option, since there's not much information to remember anyway.

Finding the server package

When you run the Server Packaging Wizard as described in the preceding section, the wizard generates several files. Simply open an Explorer window and point it to the server's folder. You see a subfolder containing the new server. Open it (see Figure 9-4).

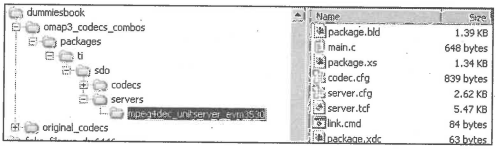


Figure 9-4: Files and directories are automatically generated by the wizard.

As you review the server package, pay attention to:

- **servers/mpeg4dec_unitserver_evm3530 directory.** The wizard generates the unit server in the same package repository as the codecs. This means you have one less package to specify on the path.
- **package.xs, package.xdc.** These files tell the build system, "Hey, I'm a package." The server's package.xs might often have extra Javascript functions to validate correct usage.
- **package.bld.** This file enables the RTSC build. It invokes the configuration system on server.tcf and the two cfg files, it specifies compilation of main.c, and then links it all together to generate a DSP executable.

The main.c file simply initializes Codec Engine and sets up a Tracing mechanism for debug. (But, let's hope you don't need to debug!)

Configuring the package

After running the wizard, the only file you *have* to manually edit is codec.cfg. When you first open the file, it looks like this:

```
var MPEG4DEC =
xdc.useModule('ti.sdo.codecs.mpeg4dec.ce.MPEG4DE
C');

    // Package Config
    MPEG4DEC.alg.watermark = true;
    MPEG4DEC.alg.codeSection = undefined;
    MPEG4DEC.alg.udataSection = undefined;
    MPEG4DEC.alg.dataSection = undefined;

Server.algs = [
  {name: "mpeg4dec", mod: MPEG4DEC ,
threadAttrs: {
    stackMemId: 0, priority: Server.MINPRI + 2},
groupId : 0,
  }
];
```

In plain English, this code says, "Use the MPEG4DEC module and set its configuration parameters. Assign a priority so that it will play nice with other codecs that may get added later."

As you look at the codec.cfg file, the things you need to edit are pretty obvious; you need to edit codeSection, udataSection, and dataSection. Each of these lines needs to be assigned to a memory section. The wizard doesn't know whether you want to place the first section in fast internal memory and the second section in slower external memory, or vice versa. These are memory layout decisions the system integrator must make.

For now, we just assign everything to external memory and let the cache do the work. Edit codec.cfg as follows:

```
MPEG4DEC.alg.codeSection = 'DDR2';
MPEG4DEC.alg.udataSection = 'DDR2';
MPEG4DEC.alg.dataSection = 'DDR2';
```

Not sure how to assign memory? You can find the memory types in the platform datasheet or the DSP/BIOS configuration server.tcf file.

Building the server executable

Now you're ready to build the server executable. Type the following:

```
[ceutils_1_06 >] make
```

If everything goes well, you will have a DSP executable named mpeg4dec_unitserver_evm3530.x64P. The DSP side of your build is now done. Great job!

Configuro — sticking with a build system you know

The content generated by the Server Packaging Wizard leverages RTSC build, and it should simply work. However, if you're more comfortable with standard makefiles or CCS project files, that's okay too. The Configuro tool from TI enables RTSC configuration in your chosen build flow.

Configuro processes a RTSC configuration script — such as server.cfg — into a set of header files, object and library files, and compiler/linker command line options. You refer to these generated files in your own build flow. Configuro is also integrated into Code Composer Studio.

Bundling Multiple Codecs into Combos

A single codec on its own isn't very exciting. For example, you may want to add an Advanced Audio Codec (AAC) Decoder to play audio alongside some video. The DVSDK decode combo does exactly this, bundling the MPEG4 and H.264 video decoders, JPEG imaging, and MP3 and AAC audio decoders. In this section, we show you how to tweak some knobs to fine-tune the performance of codec combos.

Building combos

We won't cover combo creation in this book, but you can find step-by-step instructions to build a combo from a unit server on our Web site. Visit `www.ti.com/dummiesbook` and click Building a Codec Combo.

Additionally, keep checking the Web site for details on an upcoming Combo Wizard. As the name implies, the server wizard is being upgraded to handle multiple codecs.

Tuning for performance

With a single codec, you don't need to worry about memory sharing or DMA resource contention. But in a combo, the codecs have to share. For example, if you have the H.264 and MPEG4 codecs, you're most likely not running them both at the same time. Since they don't run simultaneously, the codecs can share *scratch* data memory. Scratch memory is the opposite of *persistent* memory. A scratch buffer can be freely trashed after the buffer has processed a frame, whereas a persistent memory buffer's content needs to stick around. Scratch resources are important because fast on-chip memory is limited. Because they seldom run concurrently, video codecs are usually assigned to the same DSP/BIOS task priority and scratch memory groups. For example, a typical combo.cfg might look like this:

```
Server.algs = [
  {name: "mpeg4dec", mod:
MPEG4DEC,threadAttrs: {
   stackMemId: 0, priority: Server.MINPRI +
1}, groupId : 0,
  },
  {name: "h264avcdec", mod: H264VDEC,
threadAttrs: {
   stackMemId: 0, priority: Server.MINPRI +
1}, groupId : 0,
  },
  {name: "mp3dec", mod:
MP3DEC,threadAttrs: {
   stackMemId: 0, priority: Server.MINPRI +
2}, groupId : 1,
  }
];
```

Each video codec in the preceding example has common priorities and groupIds. This says, "MPEG4 decode and H264 decode won't preempt each other so they can share fast on-chip scratch memory." The sharing is achieved via the common groupId configuration parameter.

The audio decoder has a higher priority and a different groupId. That's because audio has a higher frame rate (its processing call repeats more frequently) so it should be allowed to preempt the longer video codecs. That's why audio and video codecs can't share the same scratch memory; the different groupId specifies different scratch buffers.

Chapter 10

How Do I Test This Thing?

In This Chapter

- Using the DVSDK demos
- Tweaking codec knobs on the Digital Video Test Bench
- Making simple applications with the DaVinci and OMAP Multimedia Application Interface

You're excited! Your codecs are XDAIS-compliant and packaged via the RTSC Codec Packaging Wizard. You also have a DSP server executable with Codec Engine, DSP/BIOS, and the neatly boxed codec inside. But how do you know that everything works?

In this chapter, you finish the job of making everything work. Flipping to the other side of TI's OMAP3530 chip — the powerful Cortex ARM A8 — you leverage some DVSDK General Purpose Processor (GPP) Linux applications to run the DSP executable.

You have several applications to start from. You can use the DVSDK demos, the Digital Video Test Bench, or the slim line applications provided with the OMAP and DaVinci Multimedia Application Interface (DMAI) library. In this chapter, we also show you when and how to use each of these tools.

As a grand finale, we introduce you to some bigger OMAP applications that do video and audio synchronization, such as GStreamer. The cool thing about GStreamer is that you don't have to write and maintain much code since a lot of it is "borrowed" from open-source plug-ins.

Using the DVSDK Demos

To be honest, the word *demo* gives the wrong impression. You've probably seen flashy tradeshow demos that reveal lots of corner-cutting and quick fixes when you get your hands on the code. It may take weeks of effort and a gallon of coffee just to get a handle on the code. The DVSDK demos from TI aren't like that. Consumer set-top boxes and portable media players based on the demo code are already in the market. The design of the processing input/output threads does a great job of keeping real-time operation in the presence of even the trickiest video streams.

The DVSDK decode demo — illustrated in Figure 10-1 — uses the Codec Engine (CE) as well as video, audio, and speech algorithms to decode video and sound data from files on your device's Linux file system. The output is sent to Linux device drivers controlling the video and audio peripherals on the OMAP3530 EVM.

The demo decodes audio using the MP3 or AAC audio algorithms, and video via the MPEG2, MPEG4, or H.264 codecs. These algorithms are packaged in a codec server named decodeCombo.x64P. Encoded video and audio data is read from separate elementary streams on the Linux file system, and the decoded data is output to peripherals on the OMAP3530 device.

Okay, enough introduction. We need to get to work plugging your MPEG4 decode server into the demos to see whether everything was done right in the work described in Chapters 6 through 9.

Modifying the demo config file

You don't need to modify the DVSDK demo code, but you do need to modify its configuration file. To do so, follow these steps:

1. **Bring up an editor in the decode demo by typing the following in a terminal window:**

```
[dvsdk_demos_3_<version>/omap3530/decode
>] emacs decode.cfg
```

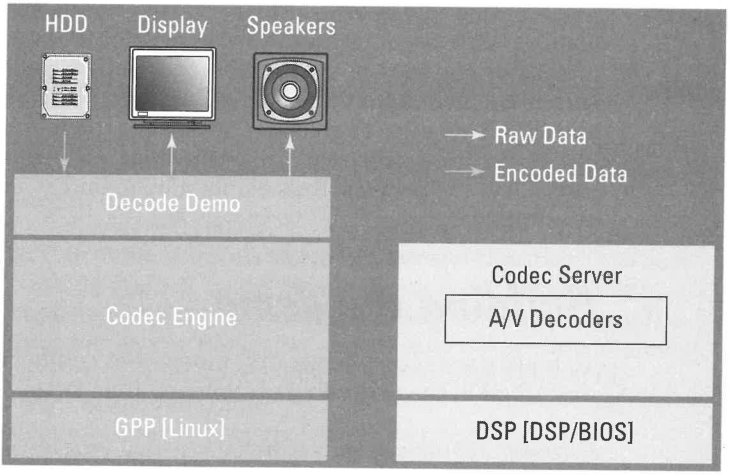


Figure 10-1: Here's a simple overview of the DVSDK decode demo.

2. **Find the lines that look like this:**

```
var engine = Engine.createFromServer(
    "decode",
    "./decodeCombo.x64P",
    "ti.sdo.servers.decode"
    );
```

3. **Replace these lines with this:**

```
var engine = Engine.createFromServer(
    "decode",
"./mpeg4dec_unitserver_evm3530.x64P",
"ti.sdo.servers.mpeg4dec_unitserver_evm
3530"
    );
```

In CE parlance, the first argument to the createFromServer function is the engine name, which is used in the application code's call to Engine_open(). Just leave "decode" in place for this argument. The second argument is the DSP server executable that the engine must load and run. The `ti.sdo.servers.mpeg4dec_unitserver_evm3530` string matches the directory package path containing the server.

The function `createFromServer()` is a handy ease-of-use feature. With previous CE versions (prior to 2.0), the system integrator had to re-specify all the codecs that went into the server *and* match up the memory configuration on the DSP and GPP. This was (a) error-prone and (b) too much typing. `createFromServer` reads all of the meta-data from a generated file in the servers, thus eliminating memory mismatch possibilities.

Building the demo

To build the decode demo, first modify the config file as described in the preceding section and then type

```
[omap3530/decode >] make
```

In the build output, you can see that the unit-server has been processed by the RTSC Configuro Tool (refer to Chapter 9). It should also build the demo C source code without warnings or errors.

Oops. Depending on which version of the demos you have, the build might fail with a message like this:

```
dmai/packages/ti/sdo/dmai/ce/Adec1.c:52:
undefined reference to `AUDDEC1_delete'
```

Remember that the demos expect to run both video *and* audio. If your mpeg4 decode server doesn't have any audio algorithms (which it won't if you followed the examples in Chapters 6 through 9), the GPP code referencing the audio decoder class fails to build. To resolve these references, simply load the audio package by adding the following to decode.cfg:

```
/* link in VISA classes */
xdc.loadPackage("ti.sdo.ce.audio1");
```

This simply tells the linker to include the audio1 libraries to satisfy the linker.

Running your DSP server executable with the demos

The moment of truth has arrived. It's time to run your newly rebuilt decode demo with your MPEG4 decoder DSP server executable.

At this point, it makes sense to have the OMAP3530 DVSDK "Getting Started Guide" handy. If you don't have the hard copy, remember it's available in the DVSDK software in the docs folder. Pay particular attention to the section "Running the demos from the command line" in Chapter 3 of the "Getting Started Guide."

Follow these steps to test your DSP server executable.

1. **Assuming your setup matches the DVSDK, copy the GPP decode demo across to the target file system by typing**

```
[omap3530/decode >] cp decode
/home/user/workdir/filesys/opt/dvsdk/
```

2. **Change directory to the DSP server executable and copy it across as well by typing**

```
 [servers/mpeg4dec_unitserver_evm3530 >]
cp mpeg4dec_unitserver_evm3530.x64P
/home/user/workdir/filesys/opt/dvsdk/
```

3. **Boot up your OMAP EVM and flip to the directory in which you just copied the files.**

You're not on the Linux host anymore. You're on a terminal within the target file system. As noted in Chapter 6, TI uses the $ sign instead of > to denote target versus host.

4. **Type the following:**

```
[opt/dvsdk $] ./decode -v
data/videos/davincieffect_ntsc_1.m4v
```

If all goes well, you'll see the DaVinci effect video clip playing on the OMAP EVM LCD display. You can also run the DaVinci effect MPEG4 clip if you want. Congratulations! If you've made it this far, you've gone through the entire flow of the DVSDK. Take a well-deserved break.

The Digital Video Test Bench

The demos are the right place to start your application design. But what if you're only at the codec evaluation stage? You have a bunch of codecs, and you want to twiddle the knobs on them to check their quality and feature sets. The Digital Video Test Bench (DVTB) is just what you need. Supplied with the DVSDK, the DVTB does the following:

- Provide an easy interface to rapidly test codec servers
- Support encode/decode of all VISA classes either from files or the OMAP3530 audio and video drivers
- Support configuration of the drivers
- Support configuration of all base parameters by the XDM/VISA classes
- Is scriptable

The last two features are the big selling points. The ability to change your MPEG4 decoder's maximum bit rate, or to flip the chroma format from YUV 4:2:2 interleaved to YUV 4:2:0 planar *without changing any code,* makes life easier. If you have a life outside work, you probably want to go home at night instead of testing every parameter combination manually. DVTB supports a simple scripting interface, so you can start a script before you go home. Then, hopefully, when you come back in the morning, you'll find that your codec passed all the tests.

Preparing to run DVTB

You can test DVTB by using a file we created for you called mpeg4dec_unitserver.dvs. It's in the code bundle at `www.ti.com/dummiesbook`. It's also pretty easy to make your own file, which typically looks like this:

```
# Specify the codec combo to be used
setp engine name mpeg4dec_engine

# Specify the decoder to be used
setp viddec2 codec mpeg4dec

# Specify number of frames to decode
setp viddec2 numFrames 1000
```

```
# Trigger the decode+display scenario
func viddec2 -s ./data/videos/
davincieffect_ntsc_1.m4v
```

Once you've downloaded the file, there are a few more prerequisite tasks to finish. Do the following:

1. **Open an editor in the DVTB directory by typing**

```
[dvtb_3_##_### >] emacs omap3530.cfg
```

2. **Specify your server name and its path like this:**

```
var engine = Engine.createFromServer(
    "mpeg4dec_engine",
   "./mpeg4dec_unitserver_evm3530.x64P",
"ti.sdo.servers.mpeg4dec_unitserver_evm353
0"
    );
```

The engine name, in this case mpeg4dec_engine, can be anything. It just needs to match up with the name you specify in the .dvs script.

3. **Build the application by typing**

```
 [dvtb_3_##_### >] make
CONFIGPKG="omap3530"
```

The argument to make is simply the name of the .cfg configuration file. If you have a different config file, just supply that as the CONFIGPKG name instead. When the build is complete, you're ready to run the executable with DVTB.

Running your DSP server executable with DVTB

DVTB produces both debug (dvtb-d) and release (dvtb-r) GPP executables. As you may have guessed, the former prints out helpful debug information as it runs.

Copy both executables (the debug and release) to the same /opt/dvsdk directory in the target file system. Copy the mpeg4dec_unitserver.dvs script across, too. Next, type the following in your terminal window:

```
[opt/dvsdk $] ./dvtb-r -s
mpeg4dec_unitserver.dvs
```

If it all works, you'll see the DaVinci effect video once more.

Making Single-Page Applications with DMAI

The first part of this chapter shows how the DVSDK demos can serve as a production code baseline. The preceding section shows how to use DVTB for codec parameter testing. But how do you make an application that's simple yet portable to a variety of platforms?

Aha! Enter now, DMAI, the DaVinci (and OMAP) Multimedia Applications Interface solution. You can literally fit an entire DMAI sample application on one page. (Okay, it's one page if you cheat by omitting error checks.)

As shown in Figure 10-2, DMAI is a thin utility layer on top of the operating system (Linux or DSP/BIOS) and CE. DMAI assists in quickly writing portable applications on OMAP and DaVinci platforms. DMAI is used by the DVSDK demos.

DMAI does not wrap the Operating System or CE. Instead, the application can choose when to use DMAI and when to use the OS or CE directly.

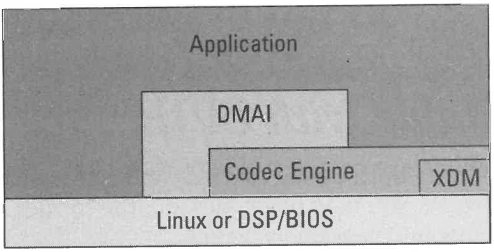


Figure 10-2: DMAI is a thin utility layer on top of the OS and Codec Engine.

Included with DMAI is a collection of modules that try to abstract common peripherals or codec operations, such as Frame copy and Audio decode. Information is passed between modules using a Buffer abstraction. Feel free to pick and choose which modules to use. Since DMAI comes with source code, it can also be used as a reference on how to accomplish certain tasks.

To evaluate your mpeg4 decode server within DMAI, we suggest using the apps/video_decode_io2 example. This is included in the example suite shipped with DMAI. The "io" indicates that file I/O will be used, instead of the video port driver. The "2" denotes the interface version, which in this case is IVIDDEC2.

Because this example is file driven, you won't see any output on the LCD display. To verify that the MPEG4 decode worked properly, you can use analysis tools such as the Elecard YUV Viewer.

Synchronizing audio and video with GStreamer

Here we take a peek at some higher level application software. TI contributes to the open-source *GStreamer* project. This is a multimedia framework capable of serving a variety of applications, such as video editors, streaming media broadcasters, and media players. GStreamer is designed as a pipeline allowing you to easily string together various media-handling components.

The TI contribution helps you to leverage the DVSDK software infrastructure (Codec Engine, Linux LSP, DSP Codecs, and so on) within the GStreamer multimedia framework on DaVinci and OMAP devices. These open-source frameworks increase productivity because code already exists for complex tasks, such as AV synchronization, demuxing/muxing, and network streaming.

TI's latest GStreamer port uses all of the cool stuff described in this chapter. In particular, the createFromServer function helped to simplify GStreamer configuration, and DMAI sped up the ports to various OMAP and DaVinci devices.

Have fun playing AVI files and other fully synchronized audio and video content.

We won't go through the build and install instructions since they're the same as the demos and DVTB described earlier in this chapter. The basic process is

1. **Modify the config file.**
2. **Build the executable.**
3. **Copy the executable(s) to the target file system.**

Okay, maybe video_decode_io2 is a bit more than one page. However, this application also demonstrates the DMAI portability story since it has a prepackaged port for DSP-only applications. For example, you can run the same CE-based application on the low-cost DM6437 c64+ device. That's sometimes useful as debugging a single processor executable is often easier than a dual-processor system. Longstanding problems with file I/O speed on the DSP have been solved by leveraging the new RTDX 2.0 protocol.

Using Pre-Canned Combos

Now, if we're honest, you — as an end-user — probably don't care how beautifully your package was boxed, or which shipping container it was placed in. You just want the finished product on your doorstep and ready to use. The same concept applies to all this QualiTI, RTSC packaging, and server stuff. Ideally, you (the system integrator) simply get a pre-canned DSP combo chock-full of the codecs you care about.

You may think that with so many codecs available, there are infinite codec combinations. However, the number of combos can be narrowed down to some extent by the target markets. For example:

- **Set top boxes.** At the very least, these will need MPEG2 video decoders and AC3 audio decode.
- **Video surveillance.** These systems use MPEG4 encoders in low-end systems, and H.264 encoders at the high-end.

To make life simpler on the OMAP3530, Texas Instruments is providing some prepackaged DSP codec combinations. The DVSDK ships a variety of video, audio, and imaging codecs within the encode and decode combos. Look for more to come in the future.

Part III
The Part of Tens

In this part . . .

The book wouldn't be complete without the *For Dummies* Part of Tens. Much of what makes OMAP and DaVinci processors special is a unique ability to run an unlimited variety of video, imaging, speech, and audio codecs. Chapter 11 provides a codec "must-do" list so software system integrators can use them quickly and efficiently. Chapter 12 gives you a list of additional Web sites and other resources that will enhance the overall OMAP and DaVinci development experience.

Chapter 11

Ten (Almost) Codec Package Requirements

In This Chapter

- Revisiting the importance of XDAIS and XDM APIs
- Using wizards to help get things done (correctly!)
- Documenting performance properly

The central characters throughout *OMAP and DaVinci Software For Dummies* are multimedia codecs. To help both the codec creator and user get on the same page with codec requirements, we've developed a list of ten (well, nine for now) must-dos and must-haves to ensure that codecs properly and easily integrate into complete systems.

- **Ensure XDAIS compliance:** Compliance with XDAIS standards (described in Chapter 3) is a critical prerequisite for any codec. A test tool, QualiTI, is available for download from `www.ti.com/dummiesbook`.
- **Document the footprint and resource usage:** The QualiTI test tool (see Chapter 7) produces memory and performance characteristics. This information should be included in a properly packaged codec.
- **Specify XDM VISA API usage:** Video, Imaging, Speech, and Audio (and now Analytics and Transcoding) codecs should all implement the latest XDM interface and explicitly state which interface they implement. It's not sufficient to simply say "it's XDM." For more on XDM, check out Chapters 3 and 7.

- **Adhere to XDM semantics:** It's not sufficient for a codec to just apply the XDM syntax. For example, using IVIDDEC1, a null termination of the outputID, is required so that the codec behaves properly in a Codec Engine environment. For more on XDM, see Chapters 3 and 7.
- **Deliver proper RTSC packages:** Codecs should be delivered as RTSC packages. To make such delivery easy, you can download a RTSC package wizard (check out Chapter 8) from `www.ti.com/dummiesbook`.
- **Provide codec unit-server examples:** The codec provider should provide a unit-server test example to run in a "real" framework like Codec Engine. A RTSC server wizard is available from `www.ti.com/dummiesbook`. We show you how to use the wizard in Chapter 9.
- **Document coprocessor use:** Codecs should document the use of hardware coprocessors so potential conflicts can be identified. XDAIS provides an IRES (for Interface-Resource) API for handling the use of coprocessors. For more on XDAIS, refer to Chapter 3.
- **Supply known good parameters:** Many XDM interfaces have a large number of parameters. Codec providers should publish a set of known good parameters (or ranges of parameters) so that the codec can run right away. Visit Chapter 7 for more on codec parameters.
- **Preserve codec performance:** To preserve codec performance in a production system, utilize link.xdt as a linker contribution from the codecs. It's a template file that is used by the RTSC tools at the final link to figure out the actual memory assignments on the platform that your system uses. Stop by Chapter 8 for more on link.xdt.

Chapter 12

Ten Super OMAP and DaVinci Resources

In This Chapter

- Exploring key TI Web sites
- Visiting open source Web sites
- Finding other hidden gems

Even with literally a world of information available on the Internet, it's sometimes a bit of a hassle to actually find what you're looking for. How many times have you heard someone say, "It's on the Web!"? It probably *is* on the Web, along with a billion other things cluttering the same space. In this chapter we go beyond telling you, "It's on the Web," and actually point out some key places to start your quest for more details.

- **`www.ti.com/dummiesbook`:** This might seem like we're stating the obvious, but probably the most important site for readers of *OMAP and DaVinci Software For Dummies* is this book's companion Web site. There, you'll find all the required software downloads, updates, and a list of corrections relating to the book.
- **`wiki.davincidsp.com`:** This is a wiki site dedicated to users of all the various software components and tools introduced throughout this book. The site contains technical articles and documents ranging from a single page to complete application notes.
- **`www.tiexpressdsp.com`:** This Web site is similar to the wiki introduced in the preceding item, but is a more general site that covers additional technologies and products not specifically covered in this book.

- **community.ti.com/forums:** These forums are a new way for developers to interact with each other and with TI developers on any number of technical subjects related to OMAP and DaVinci development.
- **source.mvista.com/git/?p=linux-omap-2.6.git;a=summary:** This is the online home of the OMAP git tree. Visit this site for all the latest and greatest OMAP kernel and driver releases, updates, patches, and more.
- **source.mvista.com/git/?p=linux-davinci-2.6.git;a=summary:** This is the online home of the DaVinci git tree. Note that this git tree is different from the OMAP one described in the preceding bullet. Visit this site for all the latest and greatest DaVinci kernel and driver releases, updates, patches, and more.
- **www.eclipse.org/dsdp/rtsc:** Real Time Software Components (RTSC; refer to Chapter 3) is an open source effort being led by TI. It's an initiative to bring consistency to the development, packaging, and delivery of embedded software components. RTSC is now part of Eclipse and has a dedicated project home page on the eclipse.org Web site.
- **focus.ti.com/lit/ug/spru352g/spru352g.pdf:** This PDF is the defacto reference document for the XDAIS standard (refer to Chapter 3). XDAIS has been established for almost ten years as a way to ease codec integration into complex systems. It creates rules and guidelines for codec developers to follow during the development cycle.
- **focus.ti.com/lit/ug/spruec8b/spruec8b.pdf:** This is an additional reference document that describes the multimedia extension to XDAIS referred to as XDM (see Chapter 3). XDM APIs create a standard for several categories of codec classes, easing the process of switching similar codecs.
- **focus.ti.com.cn/cn/lit/ug/sprue67d/sprue67d.pdf**: Last, but by no means least, is the users' guide developed specifically for applications on the Codec Engine (check out Chapter 4). Codec Engine is a convenient, pre-built software framework ideally suited for running codecs on the OMAP and DaVinci processors.

Index

Numbers

• A •

• B •

• C •

• E •

• F •

• M •

• N •

• O •

• P •

• V •

• W •

• X •

Get the most from your

OMAP and DaVinci processor-based products!

Your get-started guide to programming OMAP and DaVinci processors

Explore Texas Instruments OMAP and DaVinci processors to create your next cool product. These programmable devices provide the best of a hybrid world — a general purpose processor *and* a digital signal processor. This guide shows you how to architect your software to make optimal use of the two processors. Become familiar with ready-to-use software provided by TI, as well as tools that speed up development. You get to build a real working video/audio system and still be home for dinner with the kids.

Discover how to:

- *Partition applications across OMAP and DaVinci processors*
- *Utilize an almost unlimited variety of software codecs*
- *Leverage standards for embedded software components*
- *Build a working video and audio example*

THE DUMMIES WAY®

- *Explanations in plain English*
- *"Get in, get out" information*
- *Icons and other navigational aids*
- *Top ten lists*
- *A dash of humor and fun*

Get smart!

@ www.dummies.com

- *Find listings of all our books*
- *Choose from many different subject categories*
- *Sign up for eTips at etips.dummies.com*

ISBN: 978-0-470-39522-6
TI Part #: SPRW184
Not for resale

For Dummies®
A Branded Imprint of
WILEY