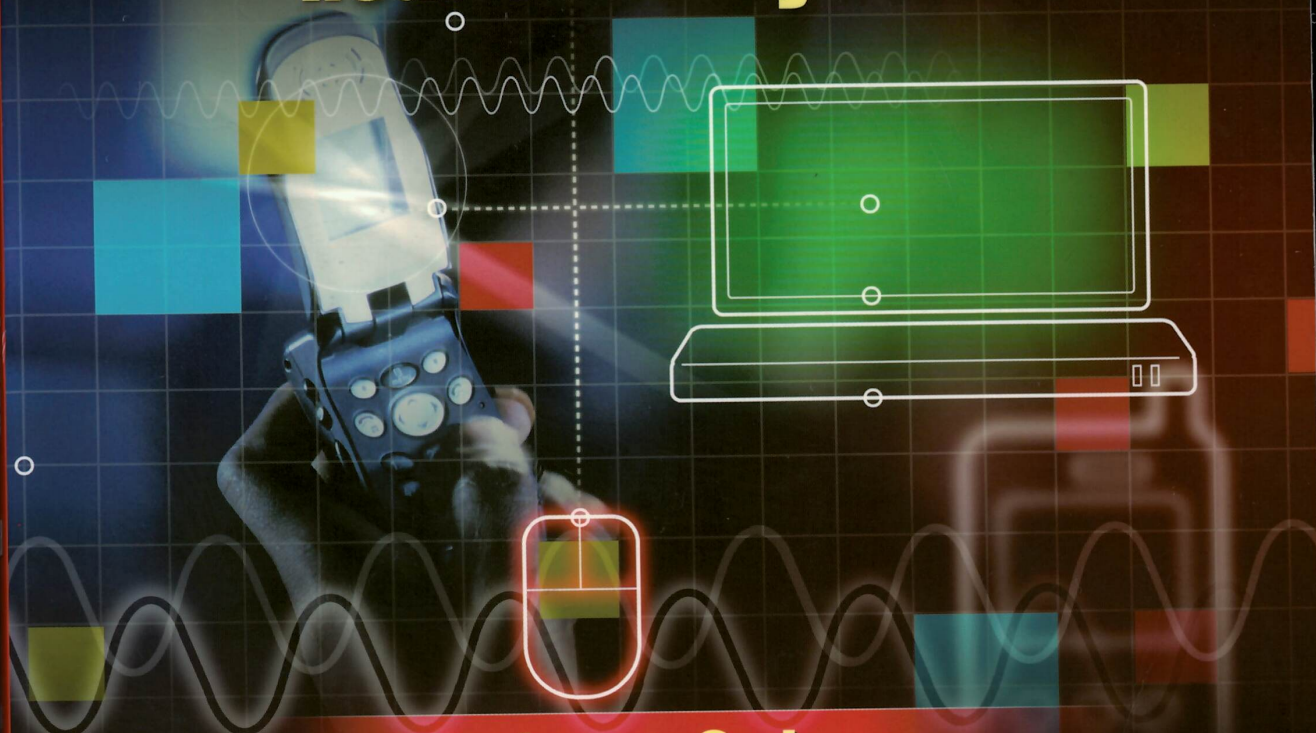


EMBEDDED TECHNOLOGY™
S E R I E S

DSP Software Development Techniques for Embedded and Real-Time Systems



Robert Oshana



**CD-ROM Contains
Code Composer Studio™ Development
Tools and Getting Started with DSP Guide**



Newnes

**DSP Software Development Techniques
for Embedded and Real-Time Systems**

DSP Software Development Techniques for Embedded and Real-Time Systems

by Robert Oshana



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

Newnes is an imprint of Elsevier



Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Copyright © 2006, Elsevier Inc. All rights reserved.

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

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



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

Library of Congress Cataloging-in-Publication Data

Application submitted.

British Library Cataloguing-in-Publication Data

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

ISBN-13: 978-0-7506-7759-2

ISBN-10: 0-7506-7759-7

For information on all Newnes publications,
visit our website at www.books.elsevier.com.

06 07 08 09 10 10 9 8 7 6 5 4 3 2

Printed in the United States of America.

Working together to grow
libraries in developing countries

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

ELSEVIER

BOOK AID
International

Sabre Foundation

*Dedicated to
Susan, Sam, and Noah*

Table of Contents

Acknowledgments	iv
Introduction: Why Use a DSP?	xi
What's on the CD-ROM?	xvii
Chapter	
1 Introduction to Digital Signal Processing	1
2 Overview of Embedded and Real-Time Systems	19
3 Overview of Embedded Systems Development Life Cycle Using DSP	35
4 Overview of Digital Signal Processing Algorithms	59
5 DSP Architectures	123
6 Optimizing DSP Software	159
7 Power Optimization Techniques Using DSP	229
8 Real-Time Operating Systems for DSP	260
9 Testing and Debugging DSP Systems	321
10 Managing the DSP Software Development Effort	351
11 Embedded DSP Software Design Using Multicore System on a Chip (SoC) Architectures	389
12 The Future of DSP Software Technology	411
Appendixes	
A Software Performance Engineering of an Embedded DSP System Application	419
B More Tips and Tricks for DSP Optimization	433
C Cache Optimization in DSP and Embedded Systems	479
D Specifying Behavior of Embedded DSP Systems	507
E Analysis Techniques for Real-time DSP Systems	525
F DSP Algorithmic Development—Rules and Guidelines	539
About the Author	569
Index	571

Overview of Embedded Systems Development Life Cycle Using DSP

Embedded Systems

As mentioned earlier, an embedded system is a specialized computer system that is integrated as part of a larger system. Many embedded systems are implemented using digital signal processors. The DSP will interface with the other embedded components to perform a specific function. The specific embedded application will determine the specific DSP to be used. For example, if the embedded application is one that performs video processing, the system designer may choose a DSP that is customized to perform media processing, including video and audio processing. An example of an application specific DSP for this function is shown in Figure 3.1. This device contains dual channel video ports that are software configurable for input or output, as well as video filtering and automatic horizontal scaling and support of various digital TV formats such as HDTV, multichannel audio serial ports, multiple stereo lines, and an Ethernet peripheral to connect to IP packet networks. It is obvious that the choice of a DSP "system" depends on the embedded application.

In this chapter we will discuss the basic steps to develop an embedded application using DSP.

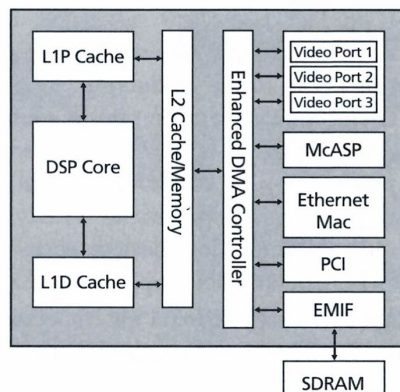


Figure 3.1 Example of a DSP-based "system" for embedded video applications

The Embedded System Life Cycle Using DSP

In this section we will overview the general embedded system life cycle using DSP. There are many steps involved in developing an embedded system—some are similar to other system development activities and some are unique. We will step through the basic process of embedded system development, focusing on DSP applications.

Step 1—Examine the Overall Needs of the System

Choosing a design solution is a difficult process. Often the choice comes down to emotion or attachment to a particular vendor or processor, inertia based on prior projects and comfort level. The embedded designer must take a positive logical approach to comparing solutions based on well defined selection criteria. For DSP, specific selection criteria must be discussed. Many signal processing applications will require a mix of several system components as shown in Figure 3.2.

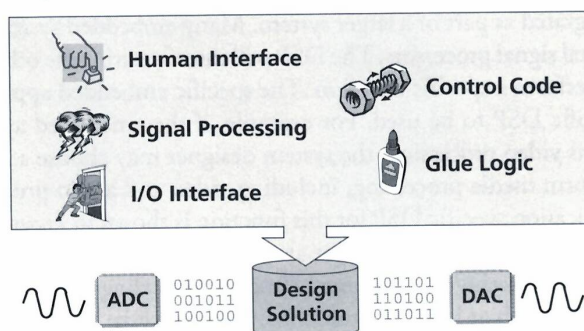


Figure 3.2 Most signal processing applications will require a mix of various system components (courtesy of Texas Instruments)

What is a DSP solution?

A typical DSP product design uses the digital signal processor itself, analog/mixed signal functions, memory, and software, all designed with a deep understanding of overall system function. In the product, the analog signals of the real world, signals representing anything from temperature to sound and images, are translated into digital bits—zeros and ones—by an analog/mixed signal device. Then the digital bits or signals are processed by the DSP. Digital signal processing is much faster and more precise than traditional analog processing. This type of processing speed is needed for today's advanced communications devices where information requires instantaneous processing, and in many portable applications that are connected to the Internet.

There are many selection criteria for embedded DSP systems. Some of these are shown in Figure 3.3. These are the major selection criteria defined by Berkeley Design Technology Incorporated (bdti.com). Other selection criteria may be “ease of use,”

Overview of Embedded Systems Development Life Cycle Using DSP 37

which is closely linked to “time-to-market” and also “features.” Some of the basic rules to consider in this phase are:

- For a fixed cost, maximize performance.
- For a fixed performance, minimize cost.

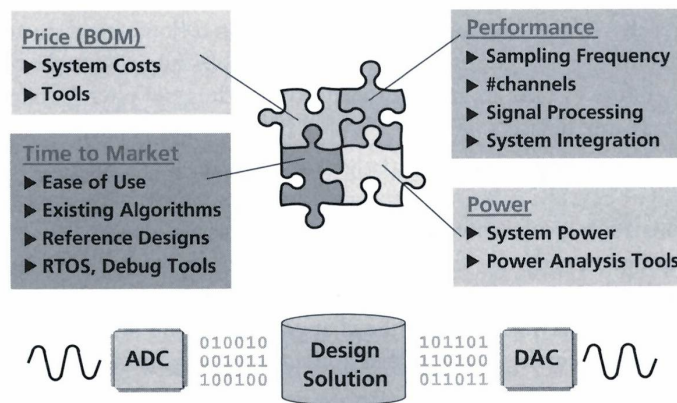


Figure 3.3 The design solution will be influenced by these major criteria and others (courtesy of Texas Instruments)

Step 2—Select the Hardware Components Required for the System

In many systems, a general-purpose processor (GPP), field-programmable gate array (FPGA), microcontroller (mC) or DSP is not used as a single-point solution. This is because designers often combine solutions, maximizing the strengths of each device (Figure 3.4).

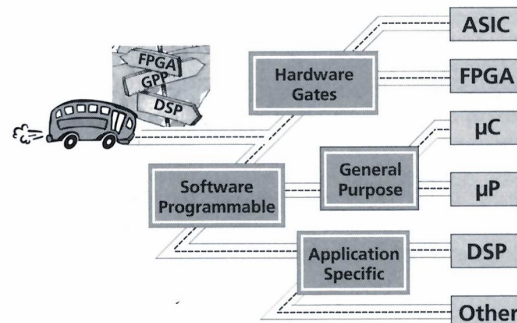


Figure 3.4 Many applications, multiple solutions (courtesy of Texas Instruments)

One of the first decisions that designers often make when choosing a processor is whether they would like a software-programmable processor in which functional

blocks are developed in software using C or assembly, or a hardware processor in which functional blocks are laid out logically in gates. Both FPGAs and application specific integrated circuits (ASICs) may integrate a processor core (very common in ASICs).

Hardware Gates

Hardware gates are logical blocks laid out in a flow, therefore any degree of parallelization of instructions is theoretically possible. Logical blocks have very low latency, therefore FPGAs are more efficient for building peripherals than “bit-banging” using a software device.

If a designer chooses to design in hardware, he or she may design using either an FPGA or ASIC. FPGAs are termed “field programmable” because their logical architecture is stored in a nonvolatile memory and booted into the device. Thus, FPGAs may be reprogrammed in the field simply by modifying the nonvolatile memory (usually FLASH or EEPROM). ASICs are not field-programmable. They are programmed at the factory using a mask which cannot be changed. ASICs are often less expensive and/or lower power. They often have sizable nonrecurring engineering (NRE) costs.

Software-Programmable

In this model, instructions are executed from memory in a serial fashion (that is, one per cycle). Software-programmable solutions have limited parallelization of instructions; however, some devices can execute multiple instructions in parallel in a single cycle. Because instructions are executed from memory in the CPU, device functions can be changed without having to reset the device. Also, because instructions are executed from memory, many different functions or routines may be integrated into a program without the need to lay out each individual routine in gates. This may make a software-programmable device more cost efficient for implementing very complex programs with a large number of subroutines.

If a designer chooses to design in software, there are many types of processors available to choose from. There are a number of general-purpose processors, but in addition, there are processors that have been optimized for specific applications. Examples of such application specific processors are graphics processors, network processors and digital signal processors (DSPs). Application specific processors usually offer higher performance for a target application, but are less flexible than general-purpose processors.

General-Purpose Processors

Within the category of general-purpose processors are microcontrollers (μC) and microprocessors (μP) (Figure 3.5).

Microcontrollers usually have control-oriented peripherals. They are usually lower cost and lower performance than microprocessors. Microprocessors usually have communications-oriented peripherals. They are usually higher cost and higher performance than microcontrollers.

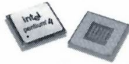
			
XSCALE, ARM	PENTIUM	Hitachi SHx	MOT PowerPC/Coldfire
Strengths	► Familiar Design Environment (tools, s/w, emulation) ► Robust communication peripherals ► Ability to use higher-end O/Ss (control code) ► Great for compiling generic (non-tuned) C code		
Signal Processing	► Fair to good		
Home Run Apps	► PC, PDA		

Figure 3.5 General-purpose processor solutions (courtesy of Texas Instruments)

Note that some GPPs have integrated MAC units. It is not a “strength” of GPPs to have this capability because all DSPs have MACs—but, it is worth noting because a student might mention it. Regarding performance of the GPP’s MAC, it is different for each one.

Microcontrollers

A microcontroller is a highly integrated chip that contains many or all of the components comprising a controller. This includes a CPU, RAM and ROM, I/O ports, and timers. Many general-purpose computer are designed the same way. But a microcontroller is usually designed for very specific tasks in embedded systems. As the name implies, the specific task is to control a particular system, hence the name microcontroller. Because of this customized task, the device’s parts can be simplified, which makes these devices very cost effective solutions for these types of applications.

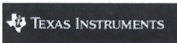
			
PIC12	68HC11/16	MCS51	MSP430
Strengths	► Good control peripherals ► May have ability to use mid-range O/Ss ► Very low cost ► Integrated FLASH ► Can be very low power		
Signal Processing	► Poor to fair		
Home Run Apps	► Embedded control, small home appliances		

Figure 3.6 Microcontroller solutions (courtesy of Texas Instruments)

Some microcontrollers can actually do a multiply and accumulate (MAC) in a single cycle. But that does not necessarily make it a DSP. True DSPs can allow two 16x16 MACs in a single cycle including bringing the data in over the buses, and so on. It is this that truly makes the part a DSP. So, devices with hardware MACs might get a "fair" rating. Others get a "poor" rating. In general, microcontrollers can do DSP but they will generally do it slower.

FPGA Solutions

An FPGA is an array of logic gates that are hardware-programmed to perform a user-specified task. FPGAs are arrays of programmable logic cells interconnected by a matrix of wires and programmable switches. Each cell in an FPGA performs a simple logic function. These logic functions are defined by an engineer's program. FPGAs contain large numbers of these cells (1000–100,000) available to use as building blocks in DSP applications. The advantage of using FPGAs is that the engineer can create special purpose functional units that can perform limited tasks very efficiently. FPGAs can be reconfigured dynamically as well (usually 100–1,000 times per second depending on the device). This makes it possible to optimize FPGAs for complex tasks at speeds higher than what can be achieved using a general-purpose processor. The ability to manipulate logic at the gate level means it is possible to construct custom DSP-centric processors that efficiently implement the desired DSP function. This is possible by simultaneously performing all of the algorithm's subfunctions. This is where the FPGA can achieve performance gains over a programmable DSP processor.

The DSP designer must understand the trade-offs when using an FPGA (Figure 3.7). If the application can be done in a single programmable DSP, that is usually the best way to go since talent for programming DSPs is usually easier to find than FPGA designers. Also, software design tools are common, cheap and sophisticated, which improves development time and cost. Most of the common DSP algorithms are also available in well packaged software components. It's harder to find these same algorithms implemented and available for FPGA designs.

An FPGA is worth considering, however, if the desired performance cannot be achieved using one or two DSPs, or when there may be significant power concerns (although a DSP is also a power efficient device—benchmarking needs to be performed) or when there may be significant programmatic issues when developing and integrating a complex software system.

Typical applications for FPGAs include radar/sensor arrays, physical system and noise modeling, and any really high I/O and high-bandwidth application.

	
FLEX 10K/ Stratix Spartan - 3/ Virtex - II	
Strengths	▶ Fastest possible computation ▶ Excellent design support tools ▶ Some kind of PLD usually required in design ▶ Ability to synthesize almost any peripheral ▶ Easy to develop with ▶ Flexible features – field reprogrammable
Signal Processing	▶ Excellent for hi-speed & parallel signal processing
Home Run Apps	▶ Glue logic, radar/sensor arrays

Figure 3.7 FPGA solutions for DSP (courtesy of Texas Instruments)

Digital Signal Processors

A DSP is a specialized microprocessor used to perform calculations efficiently on digitized signals that are converted from the analog domain. One of the big advantages of DSP is the programmability of the processor, which allows important system parameters to be changed easily to accommodate the application. DSPs are optimized for digital signal manipulations.

DSPs provide ultra-fast instruction sequences such as shift and add, and multiply and add. These instruction sequences are common in many math-intensive signal processing applications. DSPs are used in devices where this type of signal processing is important, such as sound cards, modems, cell phones, high-capacity hard disks and digital TVs (Figure 3.8).

	
BlackFin/SharC C2000/C5000/C6000 DSP56xxx/StarCore	
Strengths	▶ Architecture optimized for computing DSP algorithms ▶ Excellent MIP/mW/\$\$ tradeoff ▶ Efficient compilers – can program entire app in C ▶ Some have a real-time O/S (for task scheduling) ▶ Can be very low power
Signal Processing	▶ Good to excellent
Home Run Apps	▶ Cell phones, telecom infrastructure, digital cameras ▶ DSL/cable/modems, audio/video, multimedia

Figure 3.8 DSP processor solutions (courtesy of Texas Instruments)

A General Signal Processing Solution

The solution shown in Figure 3.9 allows each device to perform the tasks it's best at, achieving a more efficient system in terms of cost/power/performance. For example, in Figure 3.9, the system designer may put the system control software (state machines and other communication software) on the general-purpose processor or microcontroller, the high performance, single dedicated fixed functions on the FPGA and the high I/O signal processing functions on the DSP.

When planning the embedded product development cycle, there are multiple opportunities to reduce cost and/or increase functionality using combinations of GPP/uC, FPGA, and DSP. This becomes more of an issue in higher-end DSP applications. These are applications which are computationally intensive and performance critical. These applications require more processing power and channel density than can be provided by GPPs alone. For these high-end applications, there are software/hardware alternatives that the system designer must consider. Each alternative provides different degrees of performance benefits and must be also be weighed against other important system parameters including cost, power consumption and time-to-market.

The system designer may decide to use an FPGA in a DSP system for the following reasons:

- A decision to extend the life of a generic, lower-cost microprocessor or DSP by offloading computationally intensive work to a FPGA.
- A decision to reduce or eliminate the need for a higher-cost, higher performance DSP processor.
- To increase computational throughput. If the throughput of an existing system must increase to handle higher resolutions or larger signal bandwidths, an FPGA may be an option. If the required performance increases are computational in nature, an FPGA may be an option.
- For prototyping new signal processing algorithms; since the computational core of many DSP algorithms can be defined using a small amount of C code, the system designer can quickly prototype new algorithmic approaches on FPGAs before committing to hardware or other production solution like an ASIC.
- For implementing "glue" logic; various processor peripherals and other random or "glue" logic are often consolidated into a single FPGA. This can lead to reduced system size, complexity and cost.

By combining the capabilities of FPGAs and DSP processors, the system designer can increase the scope of the system design solution. Combinations of fixed hardware and programmable processors are a good model for enabling flexibility, programmability, and computational acceleration of hardware for the system.

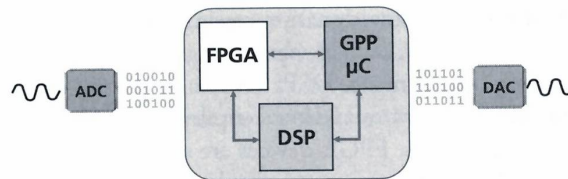


Figure 3.9 General signal processing solution (courtesy of Texas Instruments)

DSP Acceleration Decisions

In DSP system design, there are several things to consider when determining whether a functional component should be implemented in hardware or software:

Signal processing algorithm parallelism – Modern processor architectures have various forms of instruction level parallelism (ILP). One example is the 64x DSP which has a very long instruction word (VLIW) architecture (more about this in Chapter 5). The 64x DSP exploits ILP by grouping multiple instructions (adds, multiplies, loads and stores) for execution in a single processor cycle. For DSP algorithms that map well to this type of instruction parallelism, significant performance gains can be realized. But not all signal processing algorithms exploit such forms of parallelism. Filtering algorithms such as finite impulse response (FIR) algorithms are recursive and are sub-optimal when mapped to programmable DSPs. Data recursion prevents effective parallelism and ILP. As an alternative, the system designer can build dedicated hardware engines in an FPGA.

Computational complexity – Depending on the computational complexity of the algorithms, these may run more efficiently on a FPGA instead of a DSP. It may make sense, for certain algorithmic functions, to implement in a FPGA and free up programmable DSP cycles for other algorithms. Some FPGAs have multiple clock domains built into the fabric, which can be used to separate different signal processing hardware blocks into separate clock speeds based on their computational requirements. FPGAs can also provide flexibility by exploiting data and algorithm parallelism using multiple instantiations of hardware engines in the device.

Data locality – The ability to access memory in a particular order and granularity is important. Data access takes time (clock cycles) due to architectural latency, bus contention, data alignment, direct memory access (DMA) transfer rates, and even the type of memory being used in the system. For example, static RAM (SRAM) which is very fast but much more expensive than dynamic RAM (DRAM), is often used as cache memory due to its speed. Synchronous DRAM (SDRAM), on the other hand, is directly dependent on the clock speed of the entire system (that's why they call it synchronous). It basically works at the same speed as the system bus. The overall performance of the system is driven in part by which type of memory is being used. The physical interfaces between the data unit and the arithmetic unit are the primary drivers of the data locality issue.

Data parallelism – Many signal processing algorithms operate on data that is highly capable of parallelism, such as many common filtering algorithms. Some of the more advanced high-performance DSPs have single instruction multiple data (SIMD) capability in the architectures and/or compilers that implement various forms of vector processing operations. FPGA devices are also good at this type of parallelism. Large amounts of RAM are used to support high bandwidth requirements. Depending on the DSP processor being used, an FPGA can be used to provide this SIMD processing capability for certain algorithms that have these characteristics.

A DSP-based embedded system could incorporate one, two or all three of these devices depending on various factors:

- | | |
|--------------------------------------|----------------------------------|
| ▶ # signal processing tasks/channels | ▶ Amount of control code |
| ▶ Sampling rate | ▶ Development environment |
| ▶ Memory/peripherals needed | ▶ Operating system (O/S or RTOS) |
| ▶ Power requirements | ▶ Debug capabilities |
| ▶ Availability of desired algorithms | ▶ Form factor, system cost |

The trend in embedded DSP development is moving more towards programmable solutions as shown in Figure 3.10. There will always be a trade-off depending on the application but the trend is moving towards software and programmable solutions.

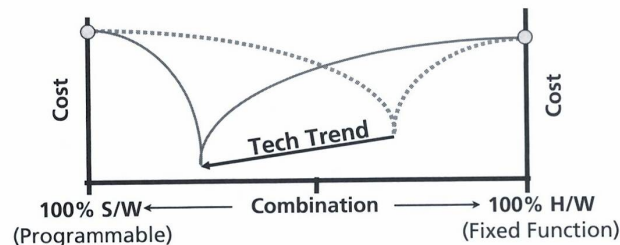


Figure 3.10 Hardware/software mix in an embedded system; the trend is towards more software (courtesy of Texas Instruments)

“Cost” can mean different things to different people. Sometimes, the solution is to go with the lowest “device cost.” However, if the development team then spends large amounts of time re-doing work, the project may be delayed; the “time-to-market” window may extend, which, in the long run, costs more than the savings of the low-cost device.

The first point to make is that a 100% software or hardware solution is usually the most expensive option. A combination of the two is the best. In the past, more functions were done in hardware and less in software. Hardware was faster, cheaper (ASICs) and good C compilers for embedded processors just weren’t available. However, today, with better compilers, faster and lower-cost processors available, the trend is toward more of a software-programmable solution. A software-only solution is not (and most likely never will be) the best overall cost. Some hardware will still be required. For example,

let's say you have ten functions to perform and two of them require extreme speed. Do you purchase a very fast processor (which costs 3–4x the speed you need for the other eight functions) or do you spend 1x on a lower-speed processor and purchase an ASIC or FPGA to do only those two critical functions? It's probably best to choose the combination.

- Cost can be defined by as a combination of the following:

▶ Device Cost	▶ Power Dissipation
▶ NRE	▶ Time to Market
▶ Manufacturing Cost	▶ Weight
▶ Opportunity Cost	▶ Size

A combination of software and hardware always gives the lowest cost system design.

Step 3—Understand DSP Basics and Architecture

One compelling reason to choose a DSP processor for an embedded system application is performance. Three important questions to understand when deciding on a DSP are:

- What makes a DSP a DSP?
- How fast can it go?
- How can I achieve maximum performance without writing in assembly?

In this section we will begin to answer these questions. We know that a DSP is really just an application specific microprocessor. They are designed to do a certain thing, signal processing, very efficiently. We mentioned the types of signal processing algorithms that are used in DSP. They are shown again in Figure 3.11 for reference.

Algorithm	Equation
Finite Impulse Response Filter	$y(n) = \sum_{k=0}^M a_k x(n-k)$
Infinite Impulse Response Filter	$y(n) = \sum_{k=0}^M a_k x(n-k) + \sum_{k=1}^N b_k y(n-k)$
Convolution	$y(n) = \sum_{k=0}^N x(k)h(n-k)$
Discrete Fourier Transform	$X(k) = \sum_{n=0}^{N-1} x(n) \exp[-j(2\pi/N)nk]$
Discrete Cosine Transform	$F(u) = \sum_{x=0}^{N-1} c(u) f(x) \cos \left[\frac{\pi}{2N} u(2x+1) \right]$

Figure 3.11 Typical DSP algorithms (courtesy of Texas Instruments)

Notice the common structure of each of the algorithms:

- They all accumulate a number of computations.
- They all sum over a number of elements.
- They all perform a series of multiplies and adds.

These algorithms all share some common characteristics; they perform multiplies and adds over and over again. This is generally referred to as the sum of products (SOP).

DSP designers have developed hardware architectures that allow the efficient execution of algorithms to take advantage of this algorithmic specialty in signal processing. For example, some of the specific architectural features of DSPs accommodate the algorithmic structure described in Figure 3.11.

As an example, consider the FIR diagram in Figure 3.12 as an example DSP algorithm which clearly shows the multiply/accumulate and shows the need for doing MACs very fast, along with reading at least two data values. As shown in Figure 3.12, the filter algorithm can be implemented using a few lines of C source code. The signal flow diagram shows this algorithm in a more visual context. Signal flow diagrams are used to show overall logic flow, signal dependencies, and code structure. They make a nice addition to code documentation.

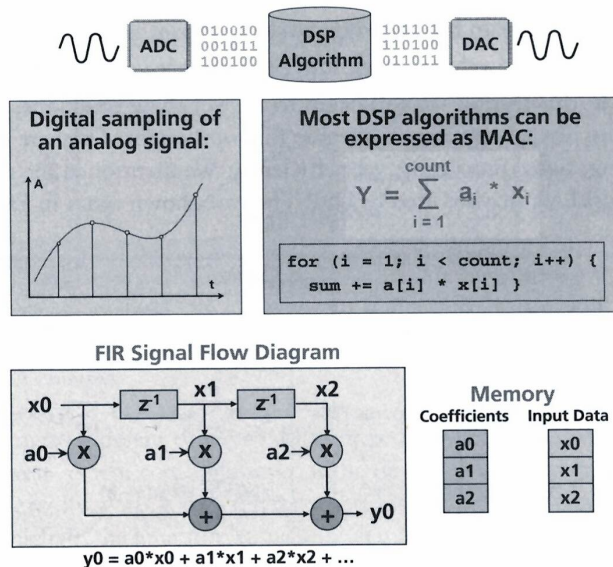


Figure 3.12 DSP filtering using a FIR filter (courtesy of Texas Instruments)

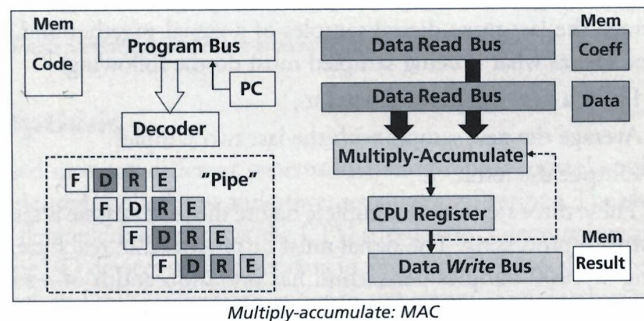
To execute at top speed, a DSP needs to:

- read *at least* two values from memory (minimum),
- multiply coeff * data,
- accumulate (+) answer (an * xn) to running total...,
- ...and do all of the above in a single cycle (or less).

DSP architectures support the requirements above (Figure 3.13):

- High-speed memory architectures support multiple accesses/cycle.
- Multiple read buses allow two (or more) data reads/cycle from memory.
- The processor pipeline overlays CPU operations allowing one-cycle execution.

All of these things work together to result in the highest possible performance when executing DSP algorithms. A deeper discussion of DSP architectures is given in Chapter 5).



- ◆ Hi-speed memory architecture supports multiple accesses/cycle
- ◆ Multiple read buses allow two (or more) data reads/cycle from memory
- ◆ Pipeline overlays CPU operations allowing one-cycle execution

Figure 3.13 Architectural block diagram of a DSP (courtesy of Texas Instruments)

Other DSP architectural features are summarized in Figure 3.14.

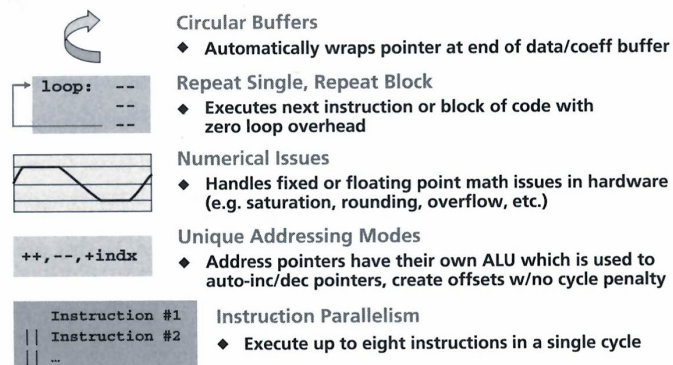


Figure 3.14 DSP CPU architectural highlights (courtesy of Texas Instruments)

Models of DSP Processing

There are two types of DSP processing models—*single sample model* and *block processing model*. In a single sample model of signal processing (Figure 3.15a), the output must result before next input sample. The goal is minimum latency (in-to-out time). These systems tend to be interrupt intensive; interrupts drive the processing for the next sample. Example DSP applications include motor control and noise cancellation.

In the block processing model (Figure 3.15b), the system will output a buffer of results before the next input buffer fills. DSP systems like this use the DMA to transfer samples to the buffer. There is increased latency in this approach as the buffers are filled before processing. However, these systems tend to be computationally efficient. The main types of DSP applications that use block processing include cellular telephony, video, and telecom infrastructure.

An example of stream processing is averaging data sample. A DSP system that must average the last three digital samples of a signal together and output a signal at the same rate as what is being sampled must do the following:

- Input a new sample and store it.
- Average the new sample with the last two samples.
- Output the result.

These three steps must complete before the next sample is taken. This is an example of stream processing. The signal must be processed in real time. A system that is sampling at 1000 samples per second has one thousandth of a second to complete the operation in order to maintain real-time performance.

Block processing, on the other hand, accumulates a large number of samples at a time and processes those samples while the next buffer of samples is being collected. Algorithms such as the fast Fourier transform (FFT) operate in this mode.

Block processing (processing a block of data in a tight inner loop) can have a number of advantages in DSP systems:

- If the DSP has an instruction cache, this cache will optimize instructions to run faster the second (or subsequent) time through the loop.
- If the data accesses adhere to a locality of reference (which is quite common in DSP systems) the performance will improve. Processing the data in stages means the data in any given stage will be accessed from fewer areas, and therefore less likely to thrash the data caches in the device.
- Block processing can often be done in simple loops. These loops have stages where only one kind of processing is taking place. In this manner there will be less thrashing from registers to memory and back. In many cases, most if not all of the intermediate results can be kept in registers or in level one cache.

- By arranging data access to be sequential, even data from the slowest level of memory (DRAM) will be much faster because the various types of DRAM assume sequential access.

DSP designers will use one of these two methods in their system. Typically, control algorithms will use single-sample processing because they cannot delay the output very long such as in the case of block processing. In audio/video systems, block processing is typically used—because there can be some delay tolerated from input to output.

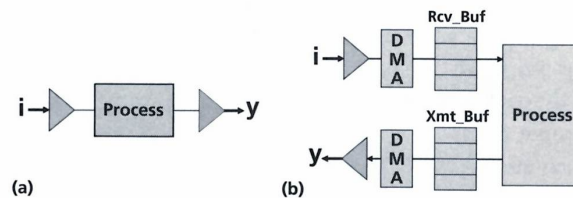


Figure 3.15 Single sample (a) and block processing (b) models of DSP

Input/Output Options

DSPs are used in many different systems including motor control applications, performance-oriented applications and power sensitive applications. The choice of a DSP processor is dependent on not just the CPU speed or architecture but also the mix of peripherals or I/O devices used to get data in and out of the system. After all, much of the bottleneck in DSP applications is not in the compute engine but in getting data in and out of the system. Therefore, the correct choice of peripherals is important in selecting the device for the application. Example I/O devices for DSP include:

GPIO – A flexible parallel interface that allows a variety of custom connections.

UART – Universal asynchronous receiver-transmitter. This is a component that converts parallel data to serial data for transmission and also converts received serial data to parallel data for digital processing.

CAN – Controller area network. The CAN protocol is an international standard used in many automotive applications.

SPI – Serial peripheral interface. A three-wire serial interface developed by Motorola.

USB – Universal serial bus. This is a standard port that enables the designer to connect external devices (digital cameras, scanners, music players, etc) to computers. The USB standard supports data transfer rates of 12 Mbps (million bits per second).

McBSP – Multichannel buffered serial port. These provide direct full-duplex serial interfaces between the DSP and other devices in a system.

HPI – Host port interface. This is used to download data from a host processor into the DSP.

A summary of I/O mechanisms for DSP application class is shown in Figure 3.16.

Motor	•12-bit ADC •PWM DAC •McBSP •UART	•CAN 2.0B •GPIO •EMIF	•SPI •SCI •I ² C
Power	•USB •McBSP •HPI	•EMIF •GPIO •10-bit ADC	•MMC/SD serial ports •UART •I ² C
Perf	•PCI •McBSP •HPI •Utopia SP	•EMIF •GPIO •I ² C	•Video ports •Audio ports •McASP •Ethernet 10/100 MAC

Figure 3.16 Input/output options (courtesy of Texas Instruments)

Calculating DSP Performance

Before choosing a DSP processor for a specific application, the system designer must evaluate three key system parameters as shown below:

- ▶ **Maximum CPU Performance**
"What is the maximum number of times the CPU can execute your algorithm? (max # channels)"
- ▶ **Maximum I/O Performance**
"Can the I/O keep up with this maximum #channels?"
- ▶ **Available Hi-Speed Memory**
"Is there enough hi-speed internal memory?"

With this knowledge, the system designer can scale the numbers to meet the application's needs and then determine:

- CPU load (% of maximum CPU).
- At this CPU load, what other functions can be performed?

The DSP system designer can use this process for any CPU they are evaluating. The goal is to find the "weakest link" in terms of performance so that you know what the system constraints are. The CPU might be able to process numbers at sufficient rates, but if the CPU cannot be fed with data fast enough, then having a fast CPU doesn't really matter. The goal is to determine the maximum number of channels that can be processed given a specific algorithm and then work that number down based on other constraints (maximum input/output speed and available memory).

As an example, consider the process shown in Figure 3.17. The goal is to determine the maximum number of channels that this specific DSP processor can handle given a specific algorithm. To do this, we must first determine the benchmark of the chosen algorithm (in this case, a 200-tap FIR filter). The relevant documentation for an algorithm like this (from a library of DSP functions) gives us the benchmark with two variables: nx (size of buffer) and nh (# coeffs)—these are used for the first part of the computation. This FIR routine takes about 106K cycles per frame. Now, consider the sampling frequency. A key question to answer at this point is “How many times is a frame FULL per second?” To answer this, divide the sampling frequency (which specifies how often a new data item is sampled) by the size of the buffer. Performing this calculation determines that we fill about 47 frames per second. Next, is the most important calculation—how many MIPS does this algorithm require of a processor? We need to find out how many cycles this algorithm will require per second. Now we multiply frames/second * cycles/frame and perform the calculation using these data to get a throughput rate of about 5 MIPS. Assuming this is the only computation being performed on the processor, the channel density (how many channels of simultaneous processing can be performed by a processor) is a maximum of $300/5 = 60$ channels. This completes the CPU calculation. This result can not be used in the I/O calculation.



How many channels can the DSP handle given this algorithm?

C P U	FIR benchmark:	$(nx/2)(nh+7) = 128 * 207 =$	26496 cyc/frm
	#times frm full/s:	$(\text{samp freq} / \text{frm size}) = 48000/256 =$	187.5 frm/s
	MIP calc:	$(\text{frm/s})(\text{cyc/frm}) = 187.5 * 26496 =$	4.97M cyc/s
	Conclusion:	FIR takes ~5MIPS on a C5502	
	Max #channels:	60 @300MHz	

Max # channels: does not include overhead for interrupts, control code, RTOS, etc.

Are the I/O and memory capable of handling this many channels?

I / O	Required I/O rate:	$48\text{Ksamp/s} * \#Ch = 48000 * 16 * 60 =$	46.08 Mbps
	DSP SP rate:	serial port is full duplex	50.00 Mbps ✓
	DMA Rate:	$(2 \times 16\text{-bit xfrs/cycle}) * 300\text{MHz} =$	9600 Mbps ✓
	Req'd Data Mem	$(60 * 200) + (60 * 4 * 256) + (60 * 2 * 199) =$	97K x 16-bit
	Avail int'l mem:		32K x 16-bit X

Required memory assumes: 60 different filters, 199 element delay buffer, double buffering rcv/xmt

Figure 3.17 Example – performance calculation (courtesy of Texas Instruments)

The next question to answer is “Can the I/O interface feed the CPU fast enough to handle 60 channels?” Step one is to calculate the “bit rate” required of the serial port. To do this, the required sampling rate (48 KHz) is multiplied by the maximum channel density (60). This is then multiplied by 16 (assuming the word size is 16—which

it is given the chosen algorithm). This calculation yields a requirement of 46 Mbps for 60 channels operating at 48 KHz. In this example what can the 5502 DSP serial port support? The specification says that the maximum bit rate is 50 Mbps (half the CPU clock rate up to 50 Mbps). This tells us that the processor can handle the rates we need for this chosen application. Can the DMA move these samples from the McBSP to memory fast enough? Again, the specification tells us that this should not be a problem.

The next step considers the issue of required data memory. This calculation is somewhat confusing and needs some additional explanation.

Assume that all 60 channels of this application are using different filters—that is, 60 different sets of coefficients and 60 double-buffers (this can be implemented using a ping-pong buffer on both the receive and transmit sides. This is a total of 4 buffers per channel hence the *4 + the delay buffers for each channel (only the receive side has delay buffers...) so the algorithm becomes:

$$\begin{aligned} &\text{Number of channels} * 2 * \text{delay buffer size} \\ &= 60 * 2 * 199 \end{aligned}$$

This is extremely conservative and the system designer could save some memory if this is not the case. But this is a worst-case scenario. Hence, we'll have 60 sets of 200 coefficients, 60 double-buffers (ping and pong on receive and transmit, hence the *4) and we'll also need a delay buffer of number of coefficients—1 which is 199 for each channel. So, the calculation is:

$$(\# \text{Channels} * \# \text{coefficients}) + (\# \text{Channels} * 4 * \text{frame size}) + (\# \text{Channels} * \# \text{delay\_buffers} * \text{delay\_buffer\_size})$$

$$= (60 * 200) + (60 * 4 * 256) + (60 * 2 * 199) = 97320 \text{ bytes of memory}$$

This results in a requirement of 97K of memory. The 5502 DSP only has 32K of on-chip memory, so this is a limitation. Again, you can redo the calculation assuming only one type of filter is used, or look for another processor.

Overview of Embedded Systems Development Life Cycle Using DSP 53

	DSP	FIR Benchmark	cyc/frm	frm/s	cyc/s	%CPU	Max Ch
C P U	C2812	$(nx/2)(nh+12) + \hat{a}$	27712	187.5	5.20M	3.5	28
	C5502	$(nx/2)(nh+7)$	26496	187.5	4.97M	1.7	60
	C6416	$(nx/4+15)(nh+11)$	16669	187.5	3.13M	0.4	230

$\hat{a} = 36nx/16 =$ additional time to transfer 16 samples to memory

	DSP	#Ch	Req'd IO rate	Avail SP rate	Avail DMA Rate	Req'd Memory	Avail Int Mem
I / O	C2812	28	21.5 Mbps	50Mbps ✓	None	46K	18K ✗
	C5502	60	46.1 Mbps	50Mbps ✓	9.6 Gbps	97K	32K ✗
	C6416	230	176.6 Mbps	100Mbps ✗	46.1 Gbps	373K	512K ✓

- ◆ Bandwidth calculations help determine processor's capability
- ◆ Limiting factors: I/O rate, available memory, CPU performance
- ◆ Use your system needs (such as 8 Ch) to calculate CPU loading (for example, 3%).

CPU load can help guide your system design...

Figure 3.18 Performance calculation analysis (courtesy of Texas Instruments)

Now we extend the calculations to the 2812 and the 6416 processors (Figure 3.18). A couple of things to note:

The 2812 is best used in a single-sample processing mode, so using a block FIR application on a 2812 is not the best fit. But for example purposes it is done this way to benchmark one processor vs. another. Where block processing hurts the 2812 is in relation to getting the samples into on-chip memory. There is no DMA on the 2812 because in single-sample processing, it is not required. The term "beta" in the calculation is the time it takes to move (using CPU cycles) the incoming sampled signals from the A/D to memory. This would be performed by an interrupt service routine and it must be accounted for. Notice that the benchmarks for the 2812 and 5502 are very close.

The 6416 is a high performance machine when doing 16-bit operations—it can do 269 channels given the specific FIR used in this example. Of course, the I/O (on one serial port) can't keep up with this, but it could with 2 serial ports in operation.

Once you've done these calculations, you can "back off" the calculation to the exact number of channels your system requires, determine an initial theoretical CPU load that is expected and then make some decisions about what to do with any additional bandwidth that is left over (Figure 3.19).

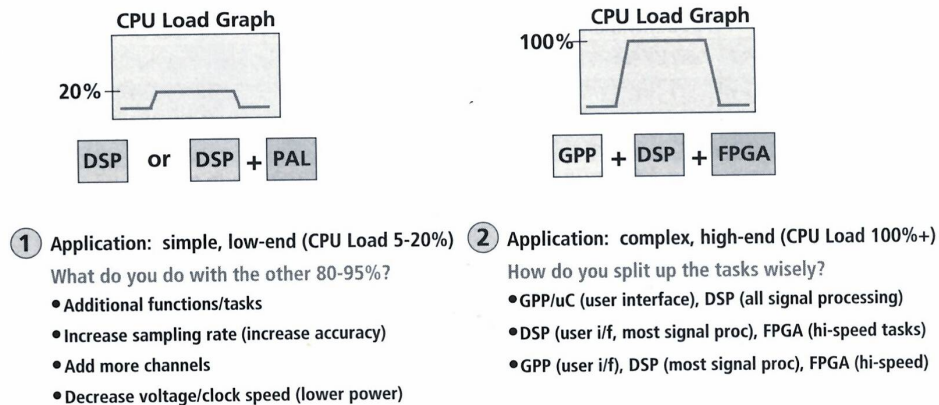


Figure 3.19 Determining what to do based on available CPU bandwidth (courtesy of Texas Instruments)

Two sample cases that help drive discussion on issues related to CPU load are shown in Figure 3.19. In the first case, the entire application only takes 20% of the CPU's load. What do you do with the extra bandwidth? The designer can add more algorithmic processing, increase the channel density, increase the sampling rate to achieve higher resolution or accuracy, or decrease the clock/voltage so that the CPU load goes up and you save lots of power. It is up to the system designer to determine the best strategy here based on the system requirements.

The second example application is the other side of the fence—where the application takes more processing power than the CPU can handle. This leads the designer to consider a combined solution. The architecture of this again depends on the application's needs.

DSP Software

DSP software development is primarily focused on achieving the performance goals of the system. It's more efficient to develop DSP software using a high-level language like C or C++ but it is not uncommon to see some of the high performance, MIPS intensive algorithms written at least partially in assembly language. When generating DSP algorithm code, the designer should use one or more of the following approaches:

- Find existing algorithms (free code).
- Buy or license algorithms from vendors. These algorithms may come bundled with tools or may be classes of libraries for specific applications (Figure 3.20).
- Write the algorithms in house. If using this approach, implement as much of the algorithm as possible in C/C++. This usually results in faster time-to-market and requires a common skill found in the industry. It is much easier to find a C

programmer than a 5502 DSP assembly language programmer. DSP compiler efficiency is fairly good and significant performance can be achieved using a compiler with the right techniques. There are several tuning techniques used to generate optimal code and these will be discussed in later chapters.

- ◆ May be bundled with tools and contains:
 - C-callable highly-optimized assembly routines
 - Documentation on each algorithm
 - Examples: FIR, IIR, FFT, convolution, min/max, log, etc.
- ◆ Other libraries available for specific DSPs:
 - Image libraries
 - Other control-specific free libraries
- ◆ Use third parties
 - Lists application software by platform, algorithm, and third party
 - Includes specs such as data/code size, performance, licensing fees

Figure 3.20 Reuse opportunities—using DSP libraries and third parties

To fine-tune code and get the highest efficiency possible, the system designer needs to know three things:

- The architecture.
- The algorithms.
- The compiler.

Figure 3.21 shows some ways to help the compiler generate efficient code. These techniques will be discussed in more detail in Chapter 6. Compilers are pessimistic by nature, so the more information that can be provided about the system algorithms, where data is in memory, and so on, the better. The C6000 compiler can achieve 100% efficiency vs. hand-coded assembly if the right techniques are used. There are pros and cons to writing DSP algorithms in assembly language as well, so if this must be done, these must be understood from the beginning (Figure 3.22).

- **Pragmas : target-specific instructions/hints to the compiler**

```
#pragma DATA_SECTION (buffer, "buffer_sect");
int buffer[32];
// places buffer in specific location in memory map
```
- **Intrinsics : C function call to access specific ASM instructions**

```
C: y = a*b;      Intrinsic: y = smpy(a,b); //saturated multiply
```
- **Compiler options: affect efficiency of compiler**
 - Optimization levels
 - Target-specific options
 - Amount of debug info
 - Many, many more...

Figure 3.21 Compiler optimization techniques for producing high performance code (courtesy of Texas Instruments)

Pros	• Can result in highest possible performance • Access to native instruction set (including application-specific instructions)
Cons	• Usually difficult learning curve (often increases development time) • Usually not portable
Conclusions	• Write in C when possible (most of the time, assembly is not required) • Don't reinvent the wheel— make full use of libraries, third parties, etc.

Figure 3.22 Pros and cons of writing DSP code in assembly language (courtesy of Texas Instruments)

DSP Frameworks

All DSP systems have some basic needs—basic requirements for processing high performance algorithms. These include:

Input/Output

- Input consists of analog information being converted to digital data.
- Output consists of digital data converted back out to analog format.
- Device drivers to talk to the actual hardware.

Processing

- Algorithms that are applied to the digitized data, for example an algorithm to encrypt secure data streams or to decode an MP3 file for playback.

Control

- Control structures with the ability to make system level decisions, for example to stop or play an MP3 file.

A DSP framework must be developed to connect device drivers and algorithms for correct data flow and processing (Figure 3.23).

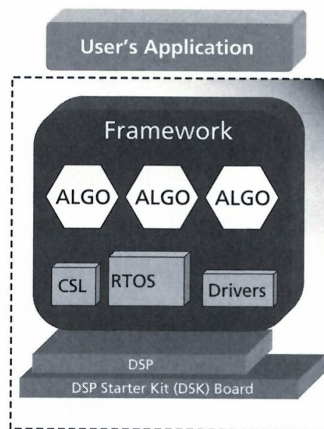


Figure 3.23 A model of a DSP framework for signal processing

A DSP framework can be custom developed for the application, reused from another application, or even purchased or acquired from a vendor. Since many DSP systems have similar processing frameworks as described above, reuse is a viable option. A framework is system software that uses standardized interfaces to algorithms and software. This includes algorithms as well as hardware drivers. The benefits of using a DSP framework include:

- The development does not have to start from scratch.
- The framework can be used as a starting point for many applications.
- The software components within a framework have well defined interfaces and work well together.
- The DSP designer can focus on the application layer which is usually the main differentiator in the product being developed. The framework can be reused.

An example DSP reference framework is shown in Figure 3.24. This DSP framework consists of:

- I/O drivers for input/output.
- Two processing threads with generic algorithms.
- Split/join threads used to simulate/utilize a stereo codec.

This reference framework has two channels by default. The designer can add and remove channels to suit the applications needs.

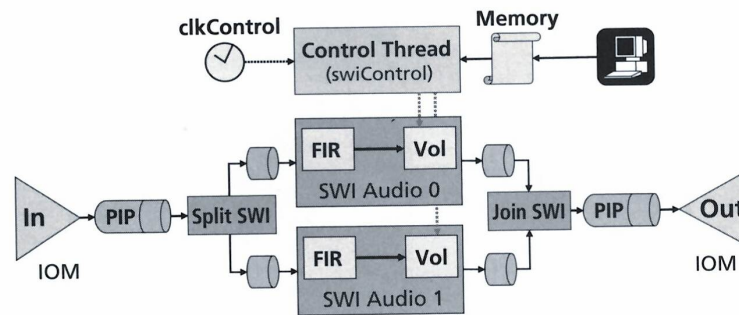


Figure 3.24 An example DSP reference framework (courtesy of Texas Instruments)

An example complete DSP Solution is shown in Figure 3.25. There is the DSP as the central processing element. There are mechanisms to get data into and out of the system (the ADC and DAC components). There is a power control module for system power management, a data transmission block with several possible peripherals including USB, FireWire®, and so on, some clock generation components and a sensor for the RF component. Of course, this is only one example, but many DSP applications follow a similar structure.

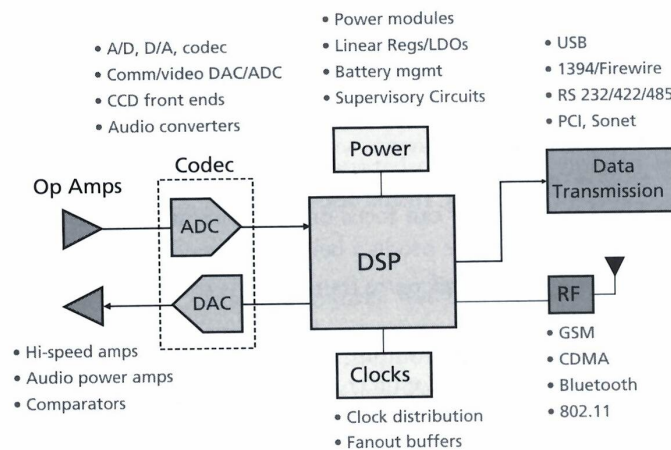


Figure 3.25 An example DSP application with major building blocks (courtesy of Texas Instruments)

DSP Software Development Techniques for Embedded and Real-Time Systems

Robert Oshana

Developing embedded and real-time DSP applications is a complex and often daunting process, influenced by a wide range of parameters and issues. This introduction to DSP software development for embedded and real-time systems guides engineers and software developers through the complicated world of embedded DSP by clearly explaining how to use digital signal processors efficiently. Robert Oshana, a highly regarded expert with decades of experience in the management and development of real-time embedded systems, covers design principles from processor architectures and basic theory to the selection of appropriate languages and algorithms. In this book, you will find practical guidelines, diagrammed techniques, tool descriptions, and code templates for developing and optimizing DSP software and firmware. Integration and testing of DSP systems are covered, in addition to thorough discussions of:

- Multicore SoC architecture DSP software designs
- Power optimization techniques
- Hard real-time vs. soft real-time systems
- Debugging DSP systems
- Cache optimization tips
- Future trends in Embedded DSP
- A case study of Software Performance Engineering
- And more!

**DIAGRAMMED TECHNIQUES, TOOL DESCRIPTIONS AND CODE TEMPLATES HELP YOU
DEVELOP AND OPTIMIZE DSP SOFTWARE!**

The accompanying **CD-ROM** contains a free trial version of **Code Composer Studio™** Development Tools, allowing you to benchmark, write sample algorithms, and put your new knowledge into practice.

Related titles:

Digital Signal Processing: A Practical Guide for Engineers and Scientists

Steven Smith, ISBN: 0-7506-7444-X

Mixed-signal and DSP Design Techniques

Walt Kester (Editor), ISBN: 0-7506-7611-6

Embedded Software: The Works

Colin Walls, ISBN: 0-7506-7954-9



Newnes

An imprint of Elsevier
www.newnespress.com

