WHITE PAPER

TEXAS INSTRUMENTS

Programming Details of Codec Engine for DaVinci™ Technology

By ***Steve Preissig,*** *Technical Training Organization, Texas Instruments*

Executive Summary

This white paper reviews the basics of the Remote Procedure Call (RPC), shows how RPC is adapted into the CE framework and examines challenges of heterogeneous processor architectures and how they may be addressed through the CE framework.

TMS320DM644x devices based on DaVinci technology are attractive options for many embedded systems due to their processing capability, high level of integration and aggressive cost. By joining a general-purpose ARM® core with the TMS320C64x+™ digital signal processor (DSP) core into a single system-on-chip (SoC) , the DaVinci platform of processors combines the time-to-market advantages of a familiar operating system such as Linux with the raw processing power of the industry's highest performance DSP core.

Combining two processor cores into a single design does, however, introduce new challenges to the software designer. How should the system be partitioned for optimal loading levels between the processors? How will scheduling be performed between the independent processors to ensure dependent activities are executed in order and with the lowest latency? And how can inter-processor communications be optimized so that the computational benefits of a heterogeneous design are not lost to data-transfer overhead?

In order to address these and other challenges, Texas Instruments has developed the Codec Engine (CE) software framework for use with DaVinci processors. This framework is built around the tried-and-true method of the Remote Procedure Call (RPC).

Remote Procedure Call Basics

The first widespread usage of the Remote Procedure Call was in the UNIX operating system in the 1980s, though its usage dates back at least to the 1970s. Whereas a Local Procedure Call (also known as a function call or subroutine) is a command issued on a processor (or computer) to be executed on that same processor, a Remote Procedure Call is a command issued on a processor to be executed on a different processor.

In the terminology of the RPC, the command-issuing processor is known as the Client and the command-executing processor is known as the Server. (See Figure 1 on the following page.) The Client sends the command and its parameters to the Server

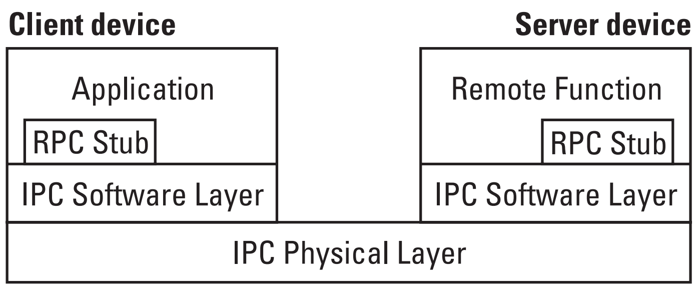

Figure 1. A simplified view of the RPC software layers.

over a physical communications medium, possibly employing some kind of communications protocol or stack. Once the Server completes execution of the command, it sends a message back over the communications medium to the Client, providing any return values of the procedure. We will refer to the communications medium used as the Inter-Processor Communications (IPC) Layer. In the case of computers, this is typically an IP network. For embedded processors there are many options, such as PCI, serial or parallel data port, or shared memory. The TMS320DM644x processors based on DaVinci technology all use shared memory for the IPC layer, employing a communications protocol named DSPLink.

As shown in Figure 1, the mechanics of an RPC are implemented through stub functions. There is a client stub function and a server stub function for every remote procedure. (Note: in the CE framework documentation, server stubs are also referred to as skeletons.) The client stub function is called by the application just as a local procedure call would be. However, instead of performing the requested function, the client stub packages the command and any needed parameters into a message and accesses the IPC layer to send the message to the server. The IPC layer on the server device then receives the message and passes it to the server stub. The server stub unpacks the command and parameters from the message and makes what is from its perspective a standard local procedure call. As such, the server stub function acts as a remote proxy for the client application, making the requested function call on its behalf. When the remote function completes, the server stub packages any return values into a return message which it sends via the IPC back to the client stub, which then returns the values to the application.

As shown in Figure 2 on the following page, there is an amount of time after the client has issued an RPC request to the server (point a) until it receives the RPC complete message from the server (point d). If the client application halts after an RPC request until the RPC complete message is returned, this is considered synchronous operation. If the client application instead continues executing other functions while the RPC is serviced, this is considered asynchronous operation. Asynchronous calls reduce latency since they allow the client to continue operating in parallel with the server. The disadvantage to asynchronous

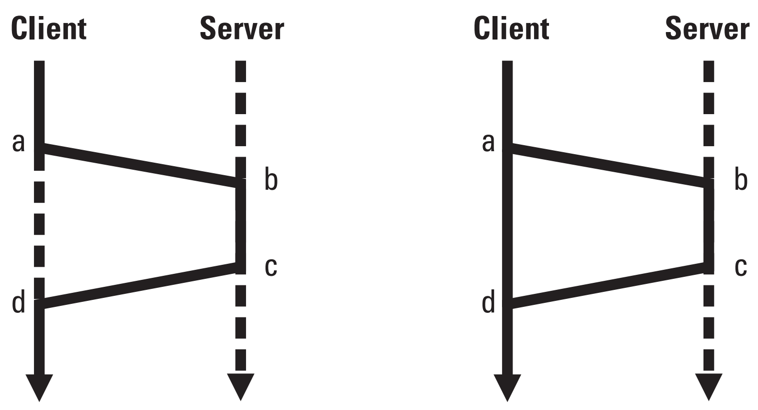

Figure 2: Synchronous (left) and asynchronous RPC flow. In both cases, a Client application thread executes until making an RPC at point a, at which point it invokes the RPC stub which communicates to the server over the IPC layer to the RPC stub on the server device. The server RPC stub begins execution at point b and completes at point c at which time it sends a message back over the IPC to the Client, which is received at point d.

calls is that they are more complex for the application programmer, who now has to manage system timing. The CE framework implements only synchronous function calls. Thus, the details of system timing are handled implicitly by the framework.

The simplest implementation of a synchronous call would be the client stub to wait in a spin loop, continuously polling for the server's RPC complete message, but this is obviously inefficient. Instead, CE RPCs are implemented with blocking execution threads. On Linux, the standard client O/S in the CE framework, the client stub blocks the POSIX thread, using a synchronization object called a semaphore. When the RPC complete message arrives, this semaphore is used to unblock the thread and continue execution. While the application thread is blocked, other threads on the client (ARM®) device can execute, so the only processing inefficiency on the client is that of context switching overhead, which is usually small in comparison to the overall time of the RPC call.

Codec Engine Framework Overview

The CE framework for DaVinci technology extends the basic RPC concept (see Figure 1) with a VISA software layer in conjunction with the Engine functional layer (see Figure 3 on page 4). The Engine functional layer manages the instantiation (i.e., creation) of algorithm objects. (By "object," we are referring to the common programming concept used in C++, Java and other object-oriented programming languages.) The VISA layer is an interface into the Engine functional layer that defines the procedure to create, delete or use a given algorithm object. Any algorithm may be created, deleted and used via the VISA interface layer as long as it conforms to the eXpressDSP™ software for Digital Media (xDM) interface standard. These two interfaces are closely related since the CE acts basically as a

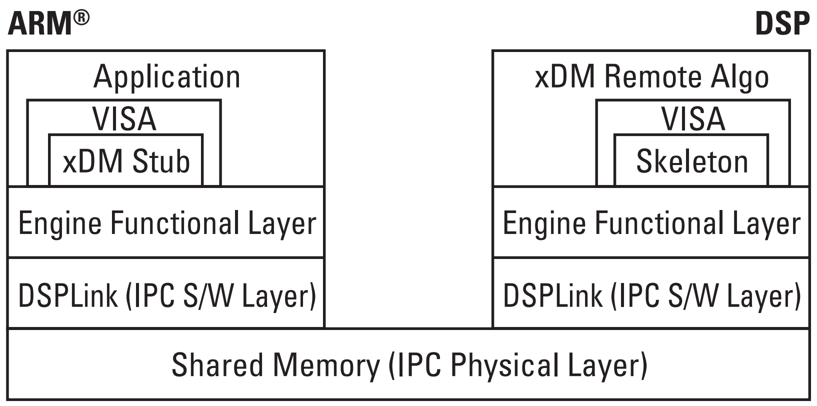

Figure 3. A simplified view of the CE framework.

conduit to transfer the VISA calls made by the application to remote xDM algorithm function calls. For this reason, the VISA interface is said to reflect the xDM interface.

Because the Engine functional layers residing on the client and server are identical as well as O/S and device independent, the VISA functions which utilize the Engine functional layer can be made remote-callable via the underlying RPC framework discussed in the previous section. The result is that the dynamic object creation capability of the Engine functional layer can be utilized either locally or remotely. As a result, algorithms are not only given support for remote *calling*, but now have support for remote *instantiation* as well.

The VISA and Engine functional layers provide other benefits as well. Firstly, the VISA layer simplifies the application programmer's interface to local and remote algorithms by providing four simple, consistent functions that can be used to access any xDM-compliant algorithm. Secondly, the Engine functional layer manages the status of algorithms as remote or local configurable during the application's build stage. This makes the calling of a function stub versus a local procedure transparent to the application programmer.

To understand a final detail of the CE framework, we will examine the four VISA functions. These are: create, control, process and delete. Create and delete are used to manage algorithm object instances. Process and control are used to access a given algorithm instance once it has been created. One might expect from our previous discussion that each of these functions or methods for the algorithm would have its own skeleton to manage its RPC call, and that each algorithm would have a unique set of skeletons. This is one possible solution; however, in practice, redundancy between the functionality of different skeletons can be taken advantage of. An example of this is in the create and delete methods.

xDM is an extension of the xDAIS standard for algorithm instantiation and deletion. As a result, every xDM-compliant algorithm is created and deleted with exactly the same function calls (i.e., the same interface). This means that all xDM-compliant algorithms can use the same skeleton function for create and delete. This is exactly what is done in the Codec Engine, and this skeleton is given a special name. It is called the Resource Management Server (RMS).

Maintaining Data Coherency

Now that we have examined the basics of the CE framework for DaVinci technology, we will address some of the possible issues that designers need to be aware of when using this architecture. The first issue that we will examine is data coherency. As we will see, most issues of Data Coherency are addressed in the stub and skeleton functions of the framework and are therefore transparent to the applications programmer.

The interprocessor communications scheme used by the Codec Engine for RPCs from the ARM® to the DSP is called DSP/BIOS™ Link. It is implemented via shared memory and internal interrupts from the ARM to the DSP and vice versa. The process is fairly straightforward. Both processors agree to a pre-determined memory address for messages sent from the ARM to the DSP and another for messages sent from the DSP to the ARM. One processor sends messages to the other by writing the message into the pre-determined address and then sending an interrupt to signal the other processor that a new message is available. Once the processor receiving the message has read it, it marks a flag in shared memory to indicate that the message memory is now available to be rewritten with another message. Using this IPC method allows large buffers of data to be passed very efficiently by pointer. Only a pointer to a given buffer needs to be passed since the buffer resides already in shared memory. However, a shared memory architecture does introduce concerns associated with the ARM's usage of virtual memory addresses as well as both processors' usage of cache.

The ARM processor contains a memory management unit (MMU) which is used to translate physical memory addresses into virtual addresses based on a translation table. This helps the processor combat memory fragmentation issues by allowing virtually contiguous buffers to be formed by mapping together physically non-contiguous segments. This causes two considerations in regard to the DSP. The first is that the DSP core on current DaVinci processors does not have an MMU, so it has no means of mapping physically non-contiguous segments into a contiguous block. The consideration is easily solved as long as the application programmer is aware of the issue. In the case of the Codec Engine framework, there is a driver module that is used to allocate physically contiguous buffers on the ARM, and the application programmer simply needs to be aware that any buffers that will be used by the DSP need to be allocated using this module. The second issue that the MMU creates is that the virtual address used by the ARM is meaningless to the DSP. This problem is even more simply solved than the first, by adding a command to the client stub function to translate all pointers into physical addresses before packaging them into the message to be sent to the DSP.

The second memory issue is cache coherency. Both the ARM and DSP cores use cache to improve the efficiency of using external memory. Both cores manage the coherency of their associated cache for any reads or writes performed by that core, so that from this standpoint, the cache is transparent to the programmer. However, the ARM and DSP cores are both unaware of reads and writes performed by the other core, and therefore neither manages cache coherency when data is passed from one core to the other via shared memory. As with the virtual address issue, this is easily solved so long as the programmer is aware of it. The client stub function is written to perform a cache writeback on any variable or buffer which is passed to the server, and the server stub function is written to perform a cache invalidation on any variables or buffers it receives.

Maintaining Temporal Coherency

Data coherency is of obvious importance because if data becomes corrupted, the remote procedure calls will not work properly. This is half of the challenge a heterogeneous architecture presents. Functions in real-time systems need not only to take the correct action, but need to take the correct action at the correct time. As previously discussed, the CE framework uses only synchronous RPCs (see "Remote Procedure Call Basics"), which automatically solves the problem of synchronizing the ARM and DSP for a given function call. As we will see, however, this does not necessarily yield the proper synchronization between threads that make remote procedure calls.

Probably the most obvious approach to scheduling a Server's fulfillment of RPC requests would be via a first-in-first-out approach. This approach is relatively simple to implement. Messages are sent through shared memory using interrupt-driven handshaking, and each received message is placed on the server's incoming message queue. The server's main loop blocks or spins until one or more messages are available and then dequeue the next message and services it.

In fact, this system would work exactly as we have described it. The issue that arises as a result is a subtle one of possible priority inversion (See Figure 4 on the following page). As an example, consider the common application of decoding an audio stream in parallel with decoding a video stream. In such a system, it is usually important to prioritize the audio portion of the application higher than the video portion. The reason is that if a temporary overloading condition occurs, dropping a frame of video tends to be far less perceptible than dropping or even repeating an audio frame.

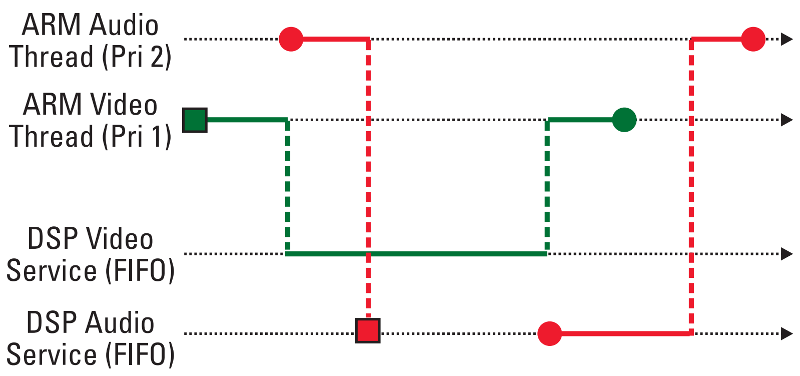

Figure 4. Priority inversion due to FIFO servicing of client requests by a DSP server. As a result of the DSP servicing requests in FIFO order, the higher priority ARM audio thread completes after the lower priority video thread.

Assuming for the moment that the developer is using a simple FIFO prioritization on the DSP Server, the situation of Figure 4 can occur. In this situation, a new video frame arrives and the video thread unblocks and sends an RPC video decode call to the DSP server slightly before the next audio buffer arrives, unblocking the audio thread. (This situation may seem contrived, but since the video and audio are probably sampled at non-divisible rates, it is statistically only a matter of time before this happens). If the server behaves in a non-prioritized, FIFO manner, it will completely decode the video buffer before servicing the audio thread's request to decode the audio buffer. Since these are synchronous RPC calls, the ARM®'s audio thread cannot continue until the server completes its decode operation, which in this case will only occur after the video frame is completely decoded. The result is that the ARM audio thread is delayed by the operation of the lower-priority video thread. The priorities of the two threads, from the ARM application developer's perspective, have been inverted.

The issue is that the priority of the client threads has not been matched on the Server. The issue may seem obvious, but it can be very difficult to track down and debug because of its infrequent occurrence. The Codec Engine framework solves this issue by prioritizing the server's response to RPC requests. The DSP server in the CE framework is built using the DSP/BIOS™ operating system, which supports prioritized, preemptable execution threads (i.e., tasks). In this framework, the skeletons for each algorithm instance are placed in their own prioritized thread when the instance is created. This allows a given RPC call to be pre-empted on the DSP server if a higher priority RPC call follows, resulting in the correct prioritization of threads from the ARM application programmer's perspective (See Figure 5 on the following page). The RMS (see Codec Engine Framework Overview section) is also placed in a thread, typically set to the lowest priority in the system so that create and delete operations will not interfere with the real-time processing of the server.

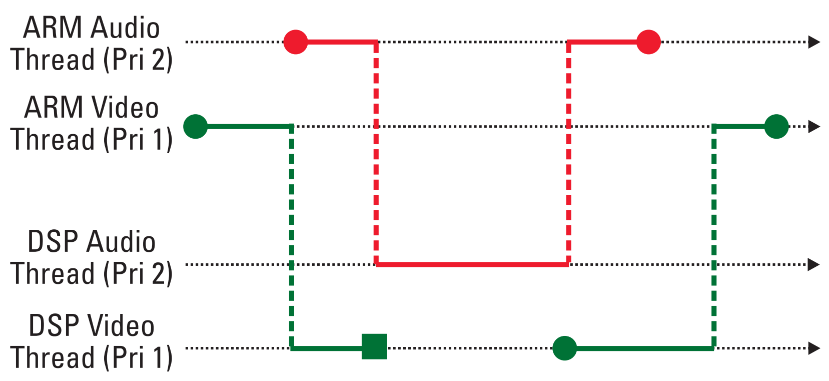

Figure 5. Correct thread operation in a prioritized RPC server. The video RPC on the DSP server is pre-empted when a higher priority audio RPC arrives.

It should be noted that building thread priority into the Server does not, by itself, guarantee that priority inversion will not occur. If the priorities of the threads that the server stubs run within do not match the priorities of the corresponding application threads generating the RPC calls, then priority inversion can occur. It is therefore important for application programmers to be aware of the thread priorities that were used in the server's configuration when it was built.

Conclusion

The Codec Engine framework provides a simple but powerful means of coordinating the ARM® and DSP processors in TI's TMS320DM644x devices. From the standpoint of the application programmer, algorithm functions are called in exactly the same manner whether they are local or remote using the VISA functions of create, process, control and delete. In fact, programmers can move a local algorithm to a remote algorithm (or vice versa) without changing a single line of application code by altering the configuration files used to build the Engine and Server.

The application programmer still needs to be aware of a few simple issues, such as the contiguous buffer and priority inversion issues discussed. A few other issues, such as virtual address translation and cache coherency, need to be taken into account by the framework developer who writes client and server stubs. These issues are typically simple to solve as long as the programmer is aware of them.

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless