New software tools are taking aim at multiprocessor DSP systems, particularly for the C6000 DSP platform.

Speeding the Development of Multi-DSP Applications

By Fiona Culloch

Although many of the latest top-of-the-line processors are extremely fast—and the 1.1-GHz TMS320C6000 from Texas Instruments is extremely fast—some customers still have performance requirements that mandate a multiprocessor solution That's why many multiprocessor C6000 systems are already in the field.

New software tools like TI's Code Composer Studio (CCS) are helping to speed software development, but for the most part they're targeted at the DSP device. Although CCS supports loading and debugging multiprocessor target systems via JTAG, its focus is understandably the DSP chips, not the additional board-specific hardware involved in interprocessor communications.

Complementary software tools are needed to truly unlock the promise of faster software development for multiprocessor DSP systems. Indeed, software tools for multiprocessor C6000 development are now emerging, such as Diamond from 3L, Virtuoso from Eonic Systems, Pegasus from Jovian Systems, and Accelera from Spectrum Signal Processing (which is specific to the company's boards).

Diamond and Virtuoso ease the software burden of coordinating many processors with features like Eonic's Virtual Single Processor model and 3L's "virtual channels." These features free you from the protocol complexities of forwarding messages among a network of processors and the interface details of your particular communications hardware. This class of products also provides ready-made software that lets a range of off-the-shelf DSP development boards communicate with user-written GUI code on a host PC.

Pegasus and Accelera are higher-level tools: graphical development environments for multi-DSP applications. Instead of writing C code to implement DSP algorithms and interprocessor communications, you can simply choose from a palette of predefined functional blocks (or add your own) and visually place the blocks onto processors.

Both kinds of tool are useful, since you often want to work at both levels. In fact, Pegasus can use either Diamond or Virtuoso as its underlying communications framework. Accelera uses Spectrum's own quicComm software as its communications layer, as it targets proprietary communications hardware.

COMMON PROBLEMS WITH MULTIPROCESSORS

Ever since computers became inexpensive enough to use more than one on the same problem, the industry has gained a lot of experience about the significant software problems introduced by multiprocessing.

Partitioning of the problem and its data. Getting multiple processors to work together effectively on the same task involves splitting the job into chunks that can be efficiently processed independently or with minimum communications. Except in special cases, partitioning is as much art as science. Creative engi-

neering is needed here, and software is of little help.

Processor communications and synchronization. Although some systems can consist of multiple, completely independent activities, in most cases the processors must exchange data via shared memory or point-to-point links.

With shared memory, each processor has access to a shared bank of memory, often through a common bus. The processors signal the availability of data by interrupts. Point-to-point links provide a dedicated hardware path between each pair of processors that need to communicate. Some DSP hardware directly supports that model, such as TMS320C4x communications ports (comports) or C6000 multichannel buffered serial ports (McBSP).

Simulation. It's hard to develop the software for a multiprocessor application on custom hardware before the hardware is available and substantially debugged. The usual single-processor solution—developing and testing software on a host platform, then porting it to the target system—isn't feasible if the software must assume the presence of the target hardware's interprocessor communications features.

Debugging. The only practical way to debug some common failure modes of multiprocessor systems, especially deadlocks caused by a particular pattern or ordering of communications, is to log the interprocessor communications (rather than single-stepping code, which is next to useless for some problems). As mentioned, that can require sophisticated software support in a system built from point-to-point links.

The key element for solving communications, simulation, and debugging problems is *abstraction* of interprocessor communications and synchronization. Unfortunately, the opposite happens in most DSP projects. Instead of keeping the details of the communications hardware out of the application code, developers tend to tie them as closely together as possible, in a misguided quest for "efficiency." Consider some concrete examples.

The SMT3xx range of modular C6000 DSP hardware from Sundance Multiprocessor Technology is based on point-to-point Sundance Digital Links (SDLs) implemented by an on-board FPGA. There are various implementations of the technology: 20-Mb/s versions backward-compatible with TI's TMS320C4x TIM-40 module comport standard, as well as faster ones that take advantage of more recent technologies. From a software point of view, to send data over a link, code must correctly initialize both the FPGA and the C6000 External Memory Interface (EMIF) CE1 control register, assign a CPU interrupt line to the transfer (avoiding any in use by concurrent I/O operations), initialize a C6000 DMA channel to transfer the data between memory and the FPGA data port (the addresses of the FPGA control registers, and the bits within them, are different for different modules in the range), and field the interrupt generated by the DMA channel on completion of the transfer.

Although the code isn't particularly complex, it's usually slow to write. Not only do you have to completely absorb all the low-level details of the hardware before writing any code, but you must also account for "kinks" in the hardware that may not have been fully documented.

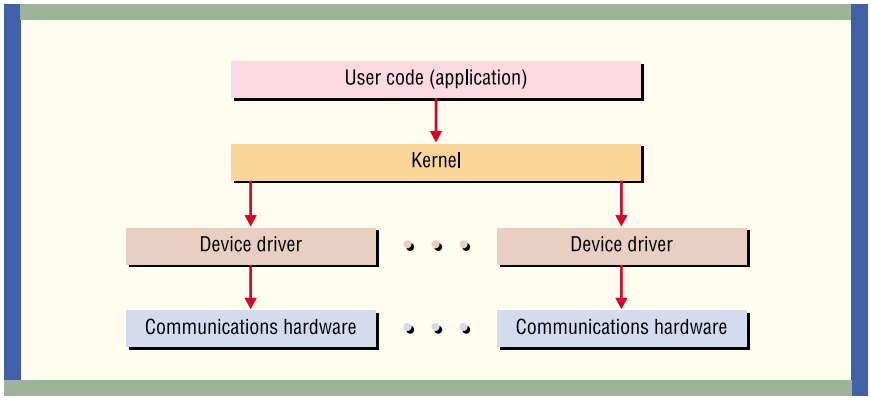


Figure 1. Device driver software provides plug-in knowledge of the specific hardware and maps between a common, device-neutral API and actual interprocessor communications hardware, which may differ for each DSP.

That's a simple case compared with Spectrum Signal Processing's Daytona and Barcelona (dual and quad) C6000 boards, which use shared PCI SRAM blocks for communications. The equivalent code to send a message from one processor to another must correctly initialize the dedicated communications ASIC (Hurricane), which implements a point-to-point link and has 64 individual control registers, as well as shared-memory buffer data to ensure that the buffer addresses are valid in the address space of each processor. It must chop the data to be transferred into chunks that fit into the SRAM bank visible to the Hurricane ASIC and, for each chunk, create a DMA channel control program in memory to drive Hurricane and start it. last, it must

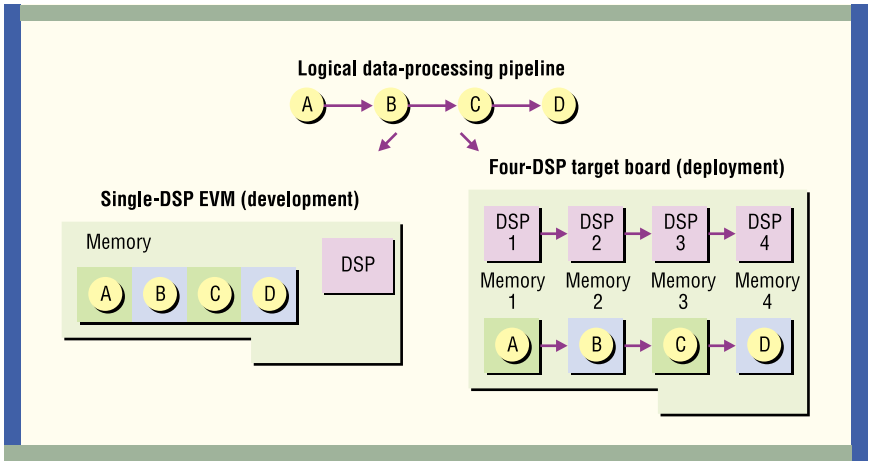


Figure 2. A four-stage pipeline can be developed on a single-processor board, then deployed without source changes on a four-DSP board for a 4x speedup.

field interrupts from the ASIC when channel program operation is complete, either moving on to the next chunk (which may involve copying further data into the Hurricane SRAM) or signaling the user code that the transfer is completed.

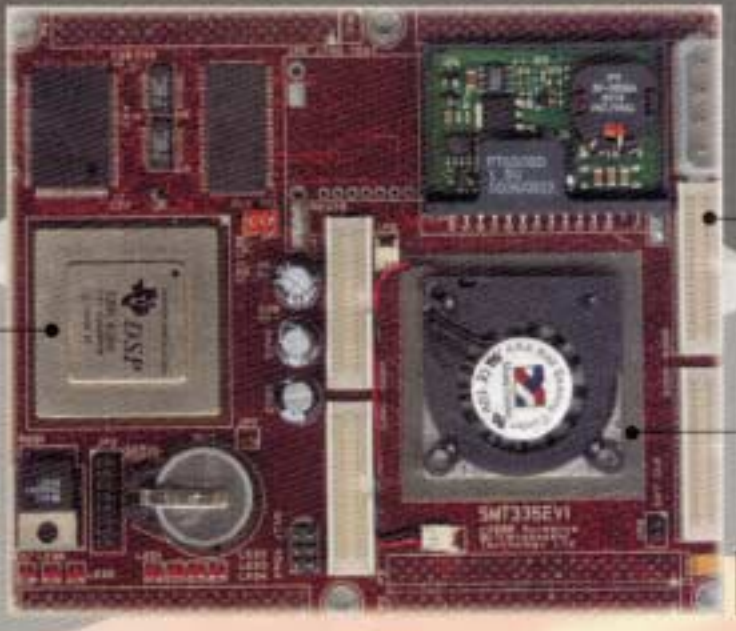
Although the managing code is often scattered throughout a project's application-level programs, it's obvious that the same abstract operation is performed in both cases: Send this much data from here in local memory to a receiver on another processor.

Abstraction decouples the application software—the DSP algorithms—from the communications hardware and contrasts with the point solutions used on many multiprocessor projects, where you try to tackle interprocessor communications by directly manipulating the underlying hardware in the application code. Although that approach

Hand-In-Hand

Power of DSP + Flexibility of FPGA

SMT335E



Features: Compact module, four SDB interfaces for fast IO or for inter processor communication(200MB/S/SDB), six built-in comports, FLASH for embedded applications. Available support for PCI, CPCI, VME and VXI. Can cascade multiple modules.

SUNDANCE DIGITAL SIGNAL PROCESSING INC.
Tel: (775) 827-3103 USA
SUNDANCE MULTIPROCESSOR TECHNOLOGY LTD.
Tel: +44 (0)1494 793167 UK
Email: sales@sundance.com http://www.sundance.com

SUNDANCE

works after a fashion, in most projects time inevitably limits the scope of the sophisticated code required to assist development. Support for tasks like debugging I/O from a node to a host system and the hardware independence required for simulation are omitted in the rush to get something working. Ironically, lacking simulation facilities, the project is likely to take much longer than if you used a tool that supports hardware-independent communications. Such tools offer many benefits.

Complete, off-the-shelf solutions. Solutions for communications, simulation, and debugging problems are feasible if the project is able to take advantage of commercial off-the-shelf (COTS) multiprocessor hardware in conjunction with software tools that abstract interprocessor communications. For example, Diamond software runs out of the box with multiprocessor C6000 boards from Spectrum Signal Processing and Sundance Multiprocessor Technology; it includes drivers for the different interprocessor communications hardware used by the boards, removing a requirement for scarce driver development skills from the project's critical path.

Simple communications API. High-level tools for multiprocessor software development solve communications problems by providing a clean, simple API for application code.

Software development for custom hardware using COTS boards. When you use tools that decouple application code from the communications hardware, you can develop software with COTS hardware, even if the final target uses custom multiprocessor boards. Good multiprocessor development tools let you port the working code to the target system without application source changes, thus addressing the simulation problem, provided that multiprocessor COTS hardware is available.

Multiprocessor software development on a single-processor platform. Being able to develop multiprocessor software on a single-processor platform means that not only is the application code independent of the communications hardware that connects the processors, it also can be independent of the number of processors and their connectivity. In particular, you can take a multiprocessor application consisting of separate programs for several processors and run it unchanged on a single processor, as long as there is adequate memory. The software automatically relocates each program to a separate position in the single processor's memory. An RTOS kernel timeshares the processor between the programs and transforms what were previously interprocessor communications into intertask communications calls to the kernel.

More interestingly, it's equally possible to go the other way: Develop a system of independent programs (processes) on one processor—for example, a simple evaluation module (EVM) from a DSP silicon vendor—and then later distribute the processes to separate DSPs when real target hardware appears, again with no application software changes (Figure 1). That feature widens the range of options for solving the simulation problem. The same flexibility that supports switching between single-processor and multiprocessor configurations without recoding also supports boosting the performance of properly designed applications by simply adding more processors, again without code changes.

Development under Windows. A corollary of independence from the underlying communications hardware is that, with software processes in a host environment like Windows

NT, it's possible to simulate multiple processors during development, expanding your armory of techniques for developing working multiprocessor software in parallel with custom hardware development (simulation problem). The trade-off is the increased availability of software tools on the host system versus difficulties introduced by simulating the target system on a CPU with a different architecture (assembly functions can't be directly tested, for example).

Framework for multiprocessor development. As you can see, you must watch out for a number of pitfalls in multiprocessor systems before achieving application speedup. Most high-level multiprocessor development tools help by providing a ready-made working framework for multiprocessor application design. The abstract model of several such tools, including Diamond, is based on C. A. R. Hoare's *Communicating Sequential Processes* (Prentice-Hall, Englewood Cliffs, N.J., 1985). Software based on communicating sequential processes (CSP) has been widely employed in the industry for at least ten years. Using such a road-tested model in place of ad hoc twiddling of hardware control registers eliminates many design and implementation errors and helps with communications and debugging.

The abstraction that produces the benefits is almost entirely conceptual—you have no implementation overhead above that of the traditional hardware-specific approach, other than selecting the required hardware-specific implementation of each communications function with a pointer rather than directly. The extra memory load required is orders of magnitude less than the amount of time required for the communications itself.

AN IMPLEMENTATION

How is the framework concept presented to you, the application developer? We'll focus on frameworks based on writing C code, using 3L's Diamond as an example, but most of the underlying concepts also apply to other high-level multiprocessor development frameworks. Note that a graphical tool like Pegasus essentially acts as a "wizard" to generate

C code from a visual block diagram, but it still allows you to work with the generated code at that level.

The key to most of the framework benefits is the hardware-independent API for interprocessor communications. Some multiprocessing aspects of Diamond and Virtuoso were influenced by Occam, the native language of the Inmos transputer, which was itself an implementation of Hoare's CSP concept. The CSP notation expresses concurrency by mathematical operators denoting synchronized message passing. Like other languages based on the CSP model, Occam has a special syntax for transmitting messages (! to send, ? to receive).

The key to most of the framework benefits is the hardware-independent API for interprocessor communications.

However, for a CSP-based concurrency framework to operate successfully with standard C tools like CCS, that syntax must give way to function calls. In Diamond's case, the API is basically two functions that operate on abstract, point-to-point channels, represented by the CHAN data type:

```
void chan_in_message(int length, void
        *buffer, CHAN *channel);
void chan_out_message(int length, void
        *buffer, CHAN *channel);
```

Those primitive operations handle both message-based communications and (implicitly) synchronization: After calling chan_out_message, the sending thread is blocked until the message is received. Similarly, a thread that calls chan_in_message is suspended until the incoming data arrives on the specified channel. Virtuoso generally uses channels directly only for system processes; its application-level tasks use mailbox and FIFO objects that the system constructed from channels. But with those frameworks, calling a simple function encapsulates all the hardware-specific work required on a particular board—for example, on the Sundance SMT3xx modules, setting up and driving the FPGA, fielding the interrupts it generates, and controlling a C6000 DMA channel (and its interrupts) to move the data between memory and the FPGA that handles interprocessor I/O.

A code-based framework presents its hardware-independent communications services as API functions in the usual way, with a header file and corresponding run-time library. C code using the API is compiled and linked by CCS's tools, as usual. Thus, just as the CCS optimizing C compiler frees you from explicitly coding DSP instructions, a code-based framework frees you from coding board-level interprocessor communications at the hardware register level.

Frameworks differ significantly in the area of system configuration. Generally, a framework has a configuration file that governs the mapping of software tasks onto the physical topology of the processor network. Code-based systems like Diamond or Virtuoso use a text file; graphical systems like Pegasus generate the file automatically from the system block diagram on screen.

The underlying configuration technology also varies. Virtuoso uses the standard linker to bind tasks, producing one executable file for each node so that tasks on the same node share a single namespace for external variables and functions. Diamond, on the other hand, can bind multiple separately linked images for each node into a single bootable file for the whole network. Any node can run multiple programs, each with its own namespace, like a process in Unix or Windows NT. An RTOS kernel component with full preemptive scheduling of dynamically created threads is loaded onto each target processor. Basic debugging takes place via JTAG with CCS, as usual. In fact, you can view a communications framework like Diamond as complementary to CCS, extending it with mechanisms for simplified handling of multiprocessor applications that have significant interprocessor communications requirements.

As well as sending and receiving messages, the other primitive required by systems based on the CSP model is waiting until a message arrives over one of several alternative channels, where the source of the message is not known in advance (similar to the select operation in some Unix variants):

```
i = alt_wait(n, &chan_1, &chan_2, ...,
        &chan_n);
```

The communications channels can be implemented with any hardware that allows for some form of communications and synchronization. For example, 3L previously implemented the same API on bit-serial transputer links, TMS320C4x comports, C6000s using PCI-bus shared memory and interrupts for signaling, and C6000s using VMEbus shared memory and interrupts. Applications can run unchanged on all of them.

What do you do to take advantage

of the leverage provided by that framework? With supported COTS multiprocessor hardware—nothing. The implementation of the framework's abstract communications primitives in terms of the supported hardware is built into the system. For other custom hardware, you create a link device driver for the framework.

DRIVERS FOR CUSTOM HARDWARE

To make use of a high-level framework for multiprocessor application development on custom hardware, you must write a device driver that maps from the framework's abstract communications API to the available hardware (Figure 2). Three approaches are possible: porting kits, fee-based service, and IP licensing.

With the first approach, you write the driver based on a porting kit provided by the framework vendor that documents the interfaces between the driver and the rest of the system. The porting kit must contain sufficient software components so that you can link your custom drivers into the system.

With the second, you go to the framework vendor. In most cases, the vendor has a great deal of experience writing drivers that interface their software to the COTS boards that it supports and is willing to write custom drivers for a fee.

IP licensing is an intermediate approach to speed up driver development based on a porting kit. You license working source code for existing implementations from the framework vendor and use that code as a base for development.

In all cases, as soon as a working driver exists for the custom hardware, multiprocessor code written with the framework—either on a supported COTS board, on a single-processor EVM, or on Windows—should run unchanged on the new hardware. ◆

Fiona Culloch (fc@3L.com) is technical support director at 3L Limited in Edinburgh, Scotland, where she has been involved with real-time kernel development and software to simplify integration of DSPs with host GUI applications. From 1985 to 1987, she was software development manager for the compilers group at Lattice Logic Limited.