USING LINUX TO CONTROL DSP PROCESSES IN MIXED-PROCESSOR SYSTEMS

A DSP process can behave like any other GPP-based Linux process and access the same Linux kernel system calls and GPP-based hardware.

By Gordon McNutt and Todd Fischer

Developing a single-processor platform is bad enough. Adding another to the mix could drive even the best developers over the edge, especially when the mix involves a general-purpose processor (GPP) and a DSP. Take heart. A new way of thinking and working allows DSP processes to make operating system calls that are processed by the GPP. Likewise, the operating system running on the GPP can control the processes on the DSP. Allowing standard process creation mechanisms to be used to load and launch DSP processes and allowing DSP processes to make standard

operating system calls greatly simplifies the design of software for dual-processor systems or DSP plus GPP systems-on-a-chip. (Texas Instruments' Digital Still Camera and OMAP platforms, for instance, contain a digital signal processor, a general-purpose processor, and a variety of I/O peripheral devices and are often used in designs with external flash memory and Internet connectivity.)

DSPs, designed for operating efficiently on huge amounts of data, are architecturally very different from GPPs, which are designed for algorithms with many logic and branching operations. Not surprisingly, DSP operating systems are likewise very different from GPP operating systems. To date, the two types have been regarded as completely independent. Developing software on a platform that contains two very different types of processors—each with its own independent operating system—increases the complexity of the solution.

With traditional approaches, a software process running on the GPP interacts with a process running on the DSP, but the two processes execute independently. The interaction involves loading the DSP code, starting the DSP process, exchanging run-time data, and initiating process shutdown. To start the DSP process, the GPP process uses different mechanisms from those used to create GPP processes. In addition, the interaction between the two processes uses different interprocess communication mechanisms from those used for GPP interprocess communication.

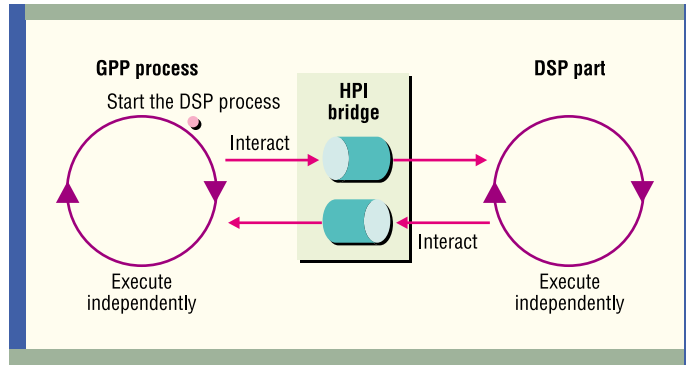


Figure 1. The traditional approach for running processes on a DSP plus GPP system-on-a-chip calls for different operating systems on the two processors. A GPP process starts the DSP process using techniques different from the mechanisms used to create GPP processes; then the DSP and GPP processes execute independently.

As a result, software developers must learn multiple ways to perform similar operations. Furthermore, this approach essentially forces a model in which the two processes have intimate knowledge of each other, or in computer science terms, have a high degree of coupling (Figure 1).

The software engineering complexity is simplified by extending the GPP operating system to support the DSP as well. For an operating system to be extended, it must have a modular design, thorough documentation, and easily available source code. For these reasons the Linux operating system works, and works well.

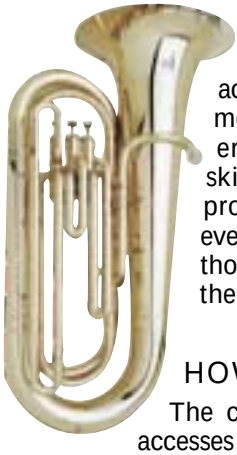
Most people are familiar with the success of Linux on servers and to a lesser degree on desktops. Less well known (but seeming to follow in its predecessor's footsteps) is the rapidly emerging distribution of Linux aimed at embedded systems. More and more, the Linux kernel and associated libraries are being scaled back to fit into embedded devices.

MODIFIED LINUX TO THE RESCUE

A Linux distribution targeted at TI's multicore products, like OMAP and DSC21, and called DSPLinux has been modified to support DSP process control. Obviously, if you're like most software developers, you don't want to monkey with OS kernels, but you need to understand the concepts involved so that you can make the best use of the modified OS as you write your application. The bottom line is that DSP processes can be managed under Linux.

In this approach, a DSP process is a Linux process that spends part of the time running on the DSP and part of the time on the GPP, has a process ID and an entry in the task structure in the Linux kernel like a normal process, can make system calls like a normal process, can send and receive signals like a normal process, and can access all the device drivers and other resource managers of the Linux kernel like a normal process. In other words, to the degree possible, a DSP process behaves like any other Linux process and has access to the same kernel interface as other, GPP-based processes. That way, you can work in a more homogenous environment, simplifying your job.

Being able to use standard Linux system calls for controlling DSP processes means that all the benefits of Linux are available: First, the Linux kernel interface is well defined, well used, well documented, and POSIX-compliant. Second, all the existing Linux features are available to DSP processes with little coding effort, including device drivers, TCP/IP, file system



access, and interprocess communication mechanisms. Third, also available are powerful programming tools and a pool of skilled software engineers. Fourth, Linux provides good code portability to support ever-improving hardware. In addition to all those advantages, the bridge library code on the DSP is very small, about 6 kwords.

HOW IT WORKS

The concept is simple. A GPP Linux process accesses the kernel via system calls—open(), read(), listen(), and so on. System calls are implemented as software interrupts.

Why can't a DSP process also make system calls, whereby the system calls are implemented as hardware interrupts? It can, if you add the necessary pieces to implement the system calls.

To find out what those pieces are, let's take a quick tour through a normal Linux system call. The get process identifier system call, getpid(), serves as a simple example. Suppose you write a program that includes the statement:

```
pid_t pid = getpid();
```

The call is implemented by the standard C library. The Linux implementation packs the system call parameters into CPU registers and then uses the GPP's software interrupt instruction to cause an interrupt; thus the system call is made in the context of the software interrupt. The entry code for all kernel system calls uses the register contents to decode the call and invokes the kernel's sys_getpid() routine. The current process made the system call, but now it's running with kernel privileges as a result of the software interrupt. The sys_getpid() function accesses the current process's kernel task structure to find the process identifier and return it. As the system call exits, the return value is stored in a register and the GPP's return-from-interrupt instruction switches the system back to user mode privileges.

To compile the same statement and execute it properly on a DSP, three items are needed (Figure 2): a library of DSP code to package system calls, invoke a hardware interrupt on the GPP, and then wait for a hardware interrupt from the GPP before unpacking the results (call this library "dsplibc"); an interrupt handler on the GPP to receive and process DSP system call requests and return the results to the DSP; and, as part of the Linux kernel, an entry in the task structure for the DSP process, plus the system calls for adding and removing the DSP process task structure entry.

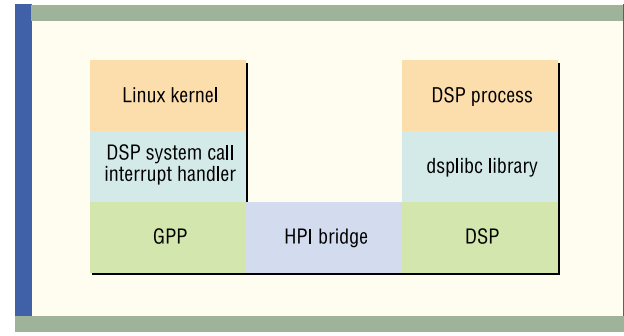


Figure 2. At the hardware level, the two processors communicate via the HPI bridge.

With TI's OMAP and DSC21 devices, the first two items are taken care of by TI's Host Port Interface (HPI) bridge, a shared-memory region that stores parameters to function calls, along with an indication of which function to call. The third item is more interesting. It answers the question how you can cleanly add or remove a DSP process from the kernel. You can do it the same way you add or remove GPP processes from the kernel: Use the existing fork(), exec(), and exit() Linux system calls. With the exception of init (the first process spawned by the kernel on startup), all processes live and die by those system calls. All you need to make this mechanism work for DSP processes is a special form of exec(). Luckily, the Linux kernel has already solved that problem.

You can cleanly add or remove a DSP process from the kernel the same way you add or remove GPP processes: Use the existing fork(), exec() and exit() Linux system calls.

All processes begin life when their parent process calls fork() or one of its variants. A new application starts executing when a process calls exec() or one of its variants and passes in the file name of an executable image. Within the kernel, exec() works by traversing a list of

registered loaders. Each loader recognizes and handles a particular image type (for example, ELF, flat, shell, Java). Then exec() probes each loader in turn, asking it to load the image from the specified file. If the loader succeeds, the application is loaded and starts to run.

Thus our DSP process actually begins life as a GPP Linux process. But when the GPP process calls exec() on a DSP binary image and exec() then probes the special binary loader that recognizes DSP images, the loader loads the image onto the DSP and starts the DSP, and the GPP Linux process is blocked waiting for system call requests from the DSP (Figure 3).

In a way, we've taken the current process and split it into two parts running synchronously. One part runs on the DSP and the other on the GPP; together the two parts make up what we call one single "DSP+GPP process." Only one part of the DSP+GPP process runs at a time; while it's running, the other remains idle. Don't confuse that with only one of the two processors running at a time. Multiple processes can be running on both the DSP and GPP so that both can be fully utilized. That's standard programming for a multitasking operating system, instead of the specialized programming required with previous techniques.

The cycle in Figure 3 continues until the DSP part makes the exit() system call, which is special in that it never returns. When the GPP part receives the exit() system call, it shuts down the DSP part and then exits like a normal Linux process. Similarly, if another process sends a fatal signal to the DSP+GPP process, the GPP process wakes up asynchronously, shuts down the DSP process, and exits.

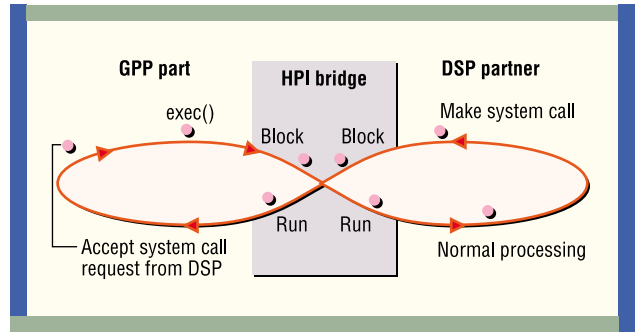


Figure 3. With a DSP+GPP process, to start the DSP part the GPP part calls exec(), ***passing in the file name of a DSP executable image. The process execution can be diagrammed as a figure-eight pattern, which continues until the*** exec() ***system call is made.***

In a way, we've taken a DSP and split it into two parts running synchronously.

When the system call parameters are passed between the partners, the information to be accessed by the system call must be in a memory region accessible to both processors. That can be tricky when the parameter is a pointer to the information. Because the dsplibc library knows which system call is being made, and thus knows about the parameters for each system call, it can take care of making the information available appropriately.

By using standard Linux process creation system calls, using standard Linux extensions to support new executable formats, and creating a kernel library for the DSP, you can harmonize the programming environment for dual-core SoCs or dual-processor systems and easily cross the boundary between DSP and GPP programming. The resulting code has a lower coupling and a lower complexity and thus is easier to maintain and extend. Also, higher-level interprocess communication protocols, like the object-oriented CORBA protocol, and the DSP/BIOS Bridge, can easily be layered on top of the Linux-controlled DSP process mechanism. Now, that should be music to your ears.

PLAYING DIGITAL AUDIO

A network-connected digital audio player is an ideal product to be powered by TI's DSP+GPP products. The DSP decodes the music stream, and the GPP can handle the user interface and the network protocol stack used to retrieve the music.

If you develop a digital audio player using traditional design approaches, a GPP process would include code to handle the intricacies of starting the DSP processes—communicating with the DSP using streaming mechanisms supported by the HPI bridge and supporting other process synchronization mechanisms to handle various errors that may arise. The resulting code would be closely tied to TI's DSP+GPP technology, and you would need expertise in the interaction of the DSP and the GPP.

However, if you use Linux-controlled DSP processes, the code on the GPP side looks very different:

```
/* Setup socket data structure */
struct sockaddr_in saddr;
```

```
saddr.sin_family = AF_INET;
saddr.sin_addr = inet_addr(music_addr.host);
saddr.sin_port=htons(music_addr.port);
/* Create socket and attempt to connect */
fd = socket(PF_INET, SOCK_STREAM, 0);
if (connect(fd, &saddr, sizeof(saddr)) < 0)
                        process_error("can not connect");
```

(To simplify the example, the user interface interaction required to obtain the network address of the music isn't shown but is assumed to be stored in the music_addr structure.) The code shows standard Linux system calls being used to open a socket to the music data. The GPP process handles any problems encountered.

If the socket is opened successfully, the DSP MP3 executable image is started using the fork() and execlp() commands:

```
/* fork a child process */
if ( !fork() ) {
        /* we are the child */
        close(stdin);
        dup(fd);
        execlp(DSP_APP, DSP_APP, 0);
}
/* we are the parent */
 ... etc ...
```

Two instances of the current process are created using fork(); execlp(), a form of exec(), then loads the DSP algorithm and starts its execution. The close(stdin) and dup(fd) commands close the "standard in" file handle, called "stdin," and pass the socket handle to the music data as "standard in" of the DSP process.

When the DSP partner process is created, the code at main() is executed:

```
main()
{
      /* setup DSP for music playback */
      ... etc ...
      read(stdio, buf, sizeof(buf));
      ... etc ...
      exit();
```

The DSP process performs any other setup necessary to start the music playback, including buffering some of the music data and spawning other DSP (or GPP) processes. When music data is needed, the DSP process issues a standard Linux read() to retrieve the music data. When the music stops playing, the DSP partner calls exit() to terminate.

This simplified example shows code for spawning the DSP process, handling the interprocess communication, and shutting down the DSP process—all using standard Linux system calls. One of the surprising results of this approach is that the code can be compiled and tested on a desktop Linux machine, including the portion of the DSP code retrieving the music data.◆

Gordon McNutt (gmcnutt@ridgerun.com) is a senior engineer at RidgeRun Inc. in Boise, Idaho, where he develops kernels and is responsible for porting Linux to embedded devices containing multiple processors. Previously, he spent a year at Hewlett-Packard developing I/O firmware to support USB, IR, and IEEE 1284.4 for LaserJet printers.

Todd Fischer (tfischer@ridgerun.com) is the director of engineering, telephony, at RidgeRun. He is working on Linux-based enabling technologies for embedded multimedia devices, like mobile phones, PDAs, digital audio players, and network-connected security cameras. and has 17 years' experience defining system architectures for embedded devices.