

A dual-processor architecture is a great way to add digital audio to an existing host application, but care must be taken.

# Creating a Dual-Processor Architecture for Digital Audio

By Paul Cohrs, William Powell and Eric Williams

**Y**ou have a great host application running on a high-end processor that's just dying to include digital audio. Texas Instruments' Internet Audio team has some nice stand-alone DSP audio applications available. It seems as if it should be fairly simple to bring the two applications together. Actually, it is—if you conduct a thorough analysis up front, take into account certain design constraints before you begin, and pay attention to details during the integration.

Anytime you consider implementing a dual-processor design, you need to be sure that this type of architecture makes sense. Certainly, if you already have one or both of the desired applications running on their own, combining the functionality by integrating the two processors is a logical solution. This type of architecture also provides for a logical division of functionality, including letting the DSP focus on the key algorithms. The DSP of course is fully capable of supporting a stand-alone application, but if you already have an existing host application, using the DSP as a digital audio “engine,” or coprocessor, will enable

you to add this functionality with minimal impact to the existing design while taking advantage of the proven algorithms on the DSP. This logical separation also allows for modular development, improved maintenance, and easier enhancement capabilities.

To demonstrate this concept, let's examine how Indesign used the DSP-based MP3 encoder application from the Internet Audio group to add MP3 encoding to a host application.

Figure 1 shows block diagrams of the existing applications. The audio

application, running on the TMS320C5416 DSP, is designed to be a self-contained program, including user interface (display and keypad), data input (codec), data output, and media storage (Compact Flash), along with the core encoding engine. PCM data is received in real time from the I2S codec interface, processed through the encoder, and stored in the Compact Flash. The host application contains a user interface, a hard disk drive, and a CD-ROM drive.

The new dual-processor architecture is shown in Figure 2. It retains the user interface (UI), the media interfaces, and the existing features from the host application. We added a UI for the encoding feature, an encoding API, and a DSP driver to the host application. The DSP retains the core MP3 encoding engine and changes the input and output data sources. The host application now sends the input (PCM) data to the encoder, and the encoder sends the output (MP3) data to the host, all via the Host Port Interface

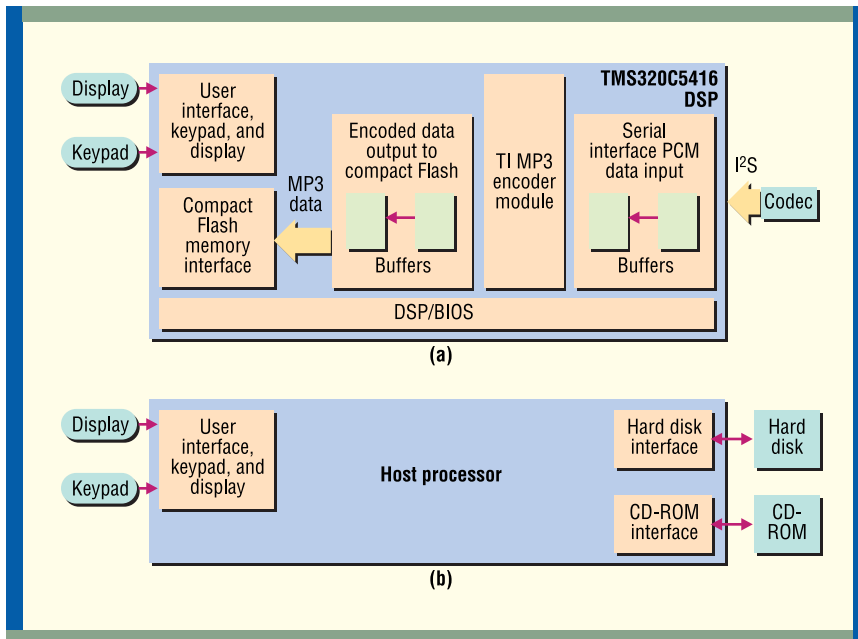


Figure 1. It's fairly simple to take stand-alone applications, such as Internet audio (a) and a host (b) and merge them into a common application providing additional functionality without having to start all over.

(HPI) bus, an integrated mechanism of TI DSPs to communicate with an external host. Finally, we removed the DSP's UI and media storage.

When developing a dual-processor design, especially when merging existing stand-alone applications, it's important to understand fully how the two processors will communicate. Some important aspects we considered were communication bandwidth, minimum and maximum latencies within each processor, initialization, and firmware program flow control. In particular, to support the new data and program flow requirements, we considered both the data bandwidth to the host and the impacts of latency in the system.

As an example of the real-time processing required, with an audio signal sampled at 44.1 kHz and 16-bit stereo, the data input for 2x processing is 2,822,400 bits per second. With MP3 encoding at 128 kbps, the

output will average 256,000 bps. Those rates aren't high for an 8- or 16-bit parallel bus, but the system response time can complicate the design.

A significant part of the data transfer time can be attributed to latency in the host application. A typical host application uses an RTOS and will have some periods during which tasks with a higher priority delay the host's servicing the DSP's data

task latency swamped the actual data transfer time.

To give a sense of how latency affects the data transfer rate, assume that the average host latency for a DSP request is 2 ms and that the DSP requests 1,024 16-bit words (16,384 bits) each time it needs data. To support a transfer rate of 2,822,400 bps, the DSP must make approximately 172 requests for data each second. Transferring the data on a 25-MHz 16-bit bus would take 7.056 ms, and the latency would be  $172 \times 2 \text{ ms} = 344 \text{ ms}$ .

If you're developing a similar dual-processor application, once you've designed the interface, you should measure the latencies to determine the overall system performance. If an improvement is required, you can employ one or more strategies to reduce the latency: raise the communication task priority, change the buffer transfer sizes, or reduce the high-priority task latencies.

Although adding MP3 encoding to the host application actually removes the real-time encoding constraint, it's desirable to optimize the interface to allow data to be transferred as fast as the encoder can produce output. Our goal was to supply data rates greater than twice real time. To do that, the interface must be able to support the transfer of audio data, a command/response

## When developing a dual-processor design, it's vital to understand how the two processors will communicate.

requests. The duration of those tasks determines the host latency when responding to the DSP. Both the maximum and average latency durations must be considered. In our host application, the higher-priority

structure for control of the encoding, and the downloading of DSP images.

The choices for interfacing to the DSP include using a serial link or the HPI bus. We chose to configure the HPI bus for nonmultiplexed

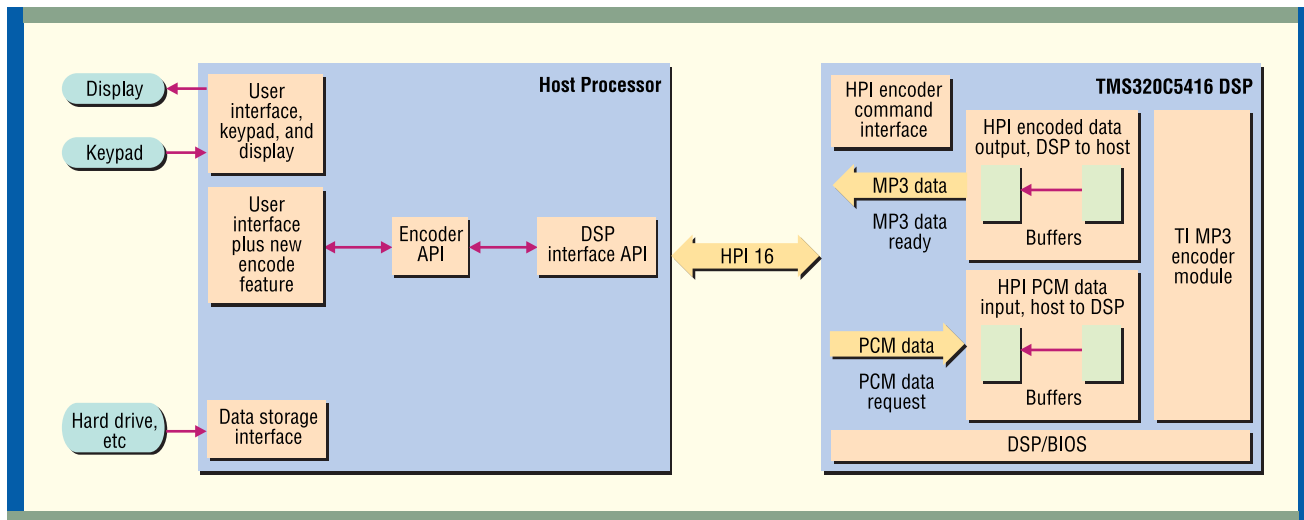


Figure 2. The dual-processor architecture takes advantage of the existing TI Host Port Interface (HPI) for connectivity. Adding some driver software on both sides to handle the HPI data transfers provides quick integration of the applications.

address and data. When using the bus, the host processor can read from and write directly to the DSP's dual-port static memory, requiring no action by the DSP. We mapped the DSP dual-port memory into the host address space and connected the host's address and data bus directly to the DSP's HPI bus. We didn't consider interfacing to the serial link because the host processor lacked a suitable interface.

The control of the interface can be done through polling by the host or

the DSP. These interrupts were used to indicate that a new message or command was available.

The stand-alone DSP application loaded the DSP application image from Compact Flash. To reduce the system cost, it was desirable to store the DSP image on the host's hard drive and download it to the DSP when initializing the latter. The DSP's built-in boot ROM has an option for loading code, via the HPI, after a reset, which is just what we needed. With the interrupt control

stop encoding commands, but also a level of handshaking to support the transfer of PCM data from the host and encoded data from the DSP. In addition, it had to provide a means to specify configuration parameters for the encoding algorithms. Lastly, the command set had to be capable of supplying images to the DSP. We decided on the command set listed in the table.

We made changes to the host application software at three levels: application, interface, and driver. At the application level, we had to develop a user interface for the new encoding feature that allows the system to use that resource. The new UI uses the existing user input and display mechanisms of the system and, because it's obviously desirable, matches the existing user interface style.

To support the application programming, we developed an API to the encoding engine to insulate the application from the details of interfacing to the DSP. The encoder API supports high-level actions required for encoding, including configuration (selection of sample rate, com-

through interrupts. To provide a quick response to the DSP's data needs, we chose interrupts. This approach required the use of an output bit on the DSP to set an interrupt on the host. We also used an external hardware interrupt input on the DSP connected to a dedicated output bit of the host to provide an interrupt to

of the interface, we were able to download a complete DSP image in 300 to 500 ms.

Once the interface architecture was settled, we needed a command set that would allow control of the encoding as well as bidirectional data transfer. The command set had to provide not only basic start and

pression rate, and other encoding parameters), start encoding, pause encoding, and stop encoding. The API performs a task similar to that done by TI's TMS320 DSP Algorithm Standard, providing a standardized set of rules and guidelines for consistent coding.

To support the encoder API (and possibly other APIs, if additional features are added that use the DSP), we also developed a DSP driver. The driver handles low-level interfacing to the DSP. It includes functions to load and initialize the DSP code image, handle the low-level command interface with the DSP, read from and write to the DSP's dual-port memory, and trigger the host-to-DSP interrupt. It also includes the DSP-to-host interrupt service routine.

Changes to the host system software include mapping the DSP dual-port memory into the host memory map, assigning a Chip Select to the DSP, connecting the host's address and data bus to the DSP's HPI bus, allocating an external interrupt for the DSP-to-host interrupt signal and designating an output bit for the host-to-DSP interrupt.

We implemented the DSP driver as a class. The driver handles the command and response control interface to the DSP and also provides a mechanism for loading the initial DSP image. Although in our design only the encoder image is loaded, this mechanism can be used to load other images providing different functions as well.

One aspect of the DSP driver that required special attention is that the dual-port memory appears as a non-contiguous block of memory to the host while appearing as one contiguous block of memory within the DSP. In addition, the host's memory address space is byte-based, whereas the DSP's memory is word-based. Since the downloaded image fills

most of the DSP memory, the driver provides address translation and byte-to-word translation to map the downloaded instructions to the proper area within the DSP. The interrupt service routine for handling DSP-to-host interrupts is also contained within this driver. The

monitor the DSP communications. The monitoring task waits for the DSP interrupt semaphore, an application command, or a time-out. It also manages the responses to DSP's requests to get new PCM data or to store encoded MP3 data and to application requests to control the

## To support the encoder API, we also developed a DSP driver, which handles low-level interfacing to the DSP.

interrupt routine sets a semaphore that the encoding monitoring task can use.

We also implemented the encoder API as a class. The API provides the interface for the application to control the encoding operation and also contains the communication-monitoring task for encoding. The initialization of the API allows the application to set the input and output file streams, configure the encoding parameters, and create a task to

encoding process. When the DSP-to-host interrupt semaphore is set, the monitoring task reads the DSP command using DSP driver functions, acknowledges receipt of the command to the DSP, and then takes the appropriate action. When an application command is received, the monitoring task translates it into DSP driver function calls for execution. To minimize the latency for interactions with the DSP and to maximize the data throughput, we

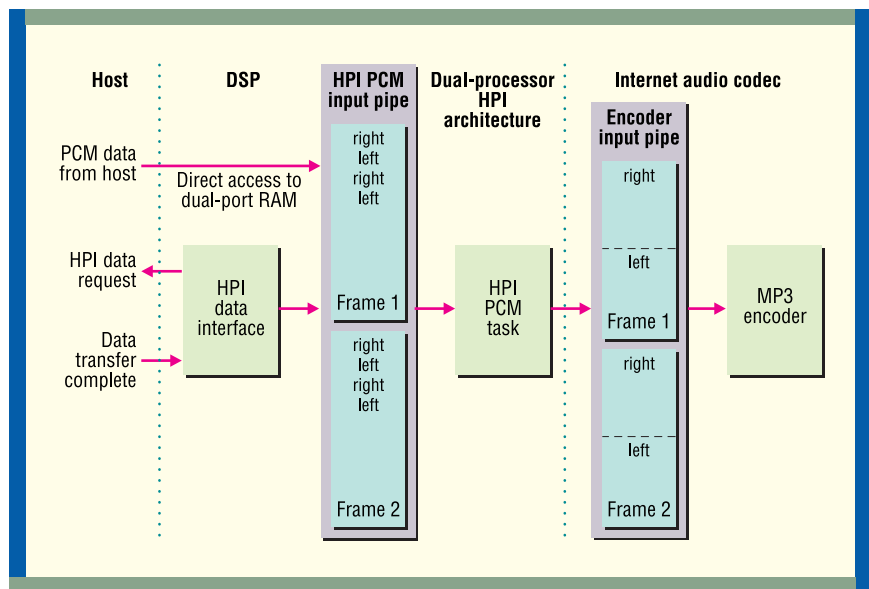


Figure 3. The encoding path takes PCM data directly from the host processor, via the HPI, through new input pipes, and on to the existing encoding engine.

# Dual-Processor Architecture

set the priority of the encoder monitoring task high.

The encoder API and the DSP driver work together with the application code to control and monitor the DSP encoding resource.

We used DSP/BIOS data pipes and software interrupts to manage the program flow control. The encoder initialization code provides data from the first buffer of to the encoder module. After this data is used, the data pipe function initiates a software interrupt to get additional data. The output of the encoder then fills the output data pipes, and the output pipe function initiates a software interrupt to transfer the data to the host. The speed of the encoder function controls the flow

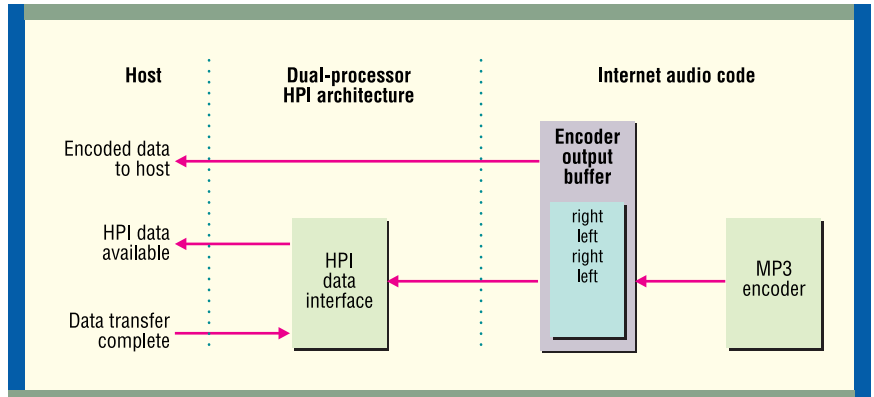


Figure 4. The encoded data is taken from the encoding engine, through the new HPI code, to the host processor, via the HPI.

of the PCM data from the host and the encoded data back to the host.

A major area of change we made

to the DSP firmware was switching the data input and output from the codec and Compact Flash to the

## DGL2K02 PCI BOARD

**Dual TMS320C6202 and AHA4210 Viterbi Decoder**

DGL2K02 PCI BOARD allows to develop and test present and future telecommunications standards simply by software algorithms implementation.



C6202



DIGITAL SYSTEMS ENGINEERING

Via Andrea Vici - zona Ind.le "La Paciana",  
06034 Foligno (Perugia) - ITALY  
Tel. +39 0742 326344 Fax +39 0742 318317  
E-mail: info@digilab2000.it



### Application Areas

- ✓ Wireless Systems
- ✓ Video Audio Broadcasting
- ✓ Software Defined Radio
- ✓ Real Time Video and Audio processing
- ✓ Embedded Systems prototyping

### Main features

- ✓ 2 TI DSPs (TMS320C6202 @250 MHz )
- ✓ PCI interface with an host PC
- ✓ Hardware Viterbi decoder up to 62,5 Mbit/s
- ✓ 1 Mbit Dual Port SRAM @125 MHz
- ✓ 16 Mbit SBSRAM for each C6202 @125 MHz
- ✓ 128 Mbit SDRAM
- ✓ Expansion connectors compliant with TI EVM
- ✓ Drivers for TI Code Composer Studio™ IDE

[www.digilab2000.it](http://www.digilab2000.it)

**ORDER NOW!**

HPI. As shown in Figure 2, we divided the HPI firmware on the DSP into three major areas: a command interface, a buffer area for PCM data from the host, and a buffer area for the encoded data back to the host. To notify the DSP of a new command being received, we added an external interrupt from the host to the DSP. We also used an output bit on the DSP to create an interrupt to the host.

The buffer area for receiving PCM data from the host replaced the I2S interface from the codec. Since the audio input is being fed from the host controller, we changed the flow of the program execution to allow the DSP to process data as quickly as possible.

We used the DSP/BIOS buffered pipe manager to control the buffer area and a pipe with two frames of 1,024 words each to buffer the data from the host. As an example, for 16-bit stereo and 44.1-kHz sample rate for the input, this system provides up to 23.2 ms of storage.

The host processor loads the buffer after receiving a Data Request command from the DSP. The Data Request command includes the start address and the minimum and maximum number of samples that the host should write. As shown in Figure 3, a new HPI PCM task takes the data received from the host processor and loads up the existing encoder input pipes.

We changed the interface from the encoder to the Compact Flash to direct the data to the host instead of to the Compact Flash driver. We modified the buffer sizes to account for the delay in the host response to the data being available and set the buffer area to 2,400 words. The DSP loads the buffers once the encoded data is available, and the host processor is notified via a Data Ready command, indicating the start address and the size of the data

Command	Source	Description
Connect	Host	Establishes connection between the processors
Data	Host	Alerts the DSP that data has been written to it
Data free	Host	Indicates that the host has read data in response to data ready command
Data ready	DSP	Indicates that the DSP has encoded data to be read by the host
Data request	DSP	DSP request for more data from the host
Record	Host	Starts the encoder and sets encoding parameters
Record BSI	Host	Requests bit stream information from the DSP
Stop	Host	Stops the encoder

to be read. After it has read the data, the host responds with a Data Free command.

The command interface, shown in Figure 4, consists of a 16-word input buffer and a 16-word output buffer that can receive the command from the host and load a command to send to the host. The handler for the interrupt from the host retrieves the received command and processes it accordingly.

With the user keypad and display interface controlled by the host controller, we disabled the DSP keypad and display interface functions. Removing the user interface and media interfaces from the DSP eliminated any real-time constraints on the encoding process. Therefore the encoding process can run faster than real time for those applications that may require more processing than would be available with a real-time input.

We evaluated the new dual-processor system using a 75-second segment of jazz music contained in a wave file recorded at 44.1 kHz, 16-bit signed stereo. The tests with the complete system resulted in an encoding rate of 1.1 to 2 times real time, depending on the sample rate and bit rate.

Several factors should be taken into account when assessing the potential limits on the performance

of this system. They include the processing time of the core algorithms, the overhead time to buffer and transfer data, and the response time of the host controller. To optimize data processing, the DSP must provide sufficient buffering for the input data to continually feed the audio algorithm, and must have sufficient output data buffering to always have room to place the processed data. If these criteria are met, the DSP can process data at the maximum rate allowed by the algorithm. ♦

*Paul Cohrs, William Powell, and Eric Williams are senior software engineers at Indesign, LLC in Indianapolis, Ind. Formerly a member of the technical staff at Bell Laboratories, Cohrs (pwcohrs@ndesign-llc.com) has spent more than 10 years in DSP firmware development, with experience in DSP applications for Internet audio, speech compression, and speech recognition. Powell (wvpowell@ndesign-llc.com) has 20 years' experience developing embedded micro-processor and DSP products. His current work involves embedded voice-over-IP terminals and research on Internet-enabled embedded products. Williams (ewilliams@ndesign-llc.com) has 15 years' experience developing and implementing DSP algorithms. He has developed DSP applications that have incorporated Bluetooth, VoIP, and Internet audio.*