Application Report
SLAA166 – January 2003

# TMS320C54x DSP Reference Framework & Device Driver for the TLV320AIC20 HPA Data Converter

*Randy Wu* *Semiconductor Sales & Marketing / Digital Applications*

**ABSTRACT**

The TLV320AIC20 is a high performance analog (HPA) data converter geared for voice-band digital applications (typically 8-kHz sampling rates). The TLV320AIC20EVM is an evaluation module that connects to a TI DSP starter kit (DSK) via a standard AIC motherboard (Part no. AICDEVPLATEVM). The EVM contains two AIC20 dual-codec devices connected in a cascaded configuration, allowing the user to configure four separate but simultaneous I/O channels using a host processor, such as the TMS320VC5416™ DSP low-power device.

This application note, along with the associated source code (which provides the standard *data pass-through* system allowing DSP algorithms to be inserted for digital signal processing), explains how to use the provided reference platform to evaluate the data converter, create AIC20-based device drivers, insert sample algorithms to apply digital signal processing to the data streams, and how to use the provided eXpressDSP™ software framework as the foundation for developing actual AIC20-based applications. A complete Code Composer Studio™ project with all the source code (completely written in *C* for readability, portability, maintainability, and ease of use) is available for download along with this application note. Project collateral discussed in this application note can be downloaded from the following URL: http://www.ti.com/lit/zip/SLAA166.

**Contents**

## Figures



## Tables

# 1 Introduction

The TLV320AIC20 dual-channel codec device, like all data converters used in a digital signal processing system, needs a host processor to control the device during run-time of the system. For example, a TMS320 TM DSP Platform can be used to filter the voice of a microphone of a headset, and/or apply some noise cancellation algorithm to the microphone, left ear, and right ear channels of the headset. A DSP is the ideal processor to set up the AIC20 to sample the different channels of analog-to-digital converters at some conventional sample rate (e.g. 8 kHz) and process those samples through the filter and noise reduction algorithms on the DSP. The DSP then routes the processed samples back to the digital-to-analog converters, producing the desired voice-band outputs. Other popular applications include (but not limited to) digital hearing aids, interactive toys with voice recognition and/or speech synthesis capabilities, modems, and cell/speaker-phones.

Writing software device drivers (i.e. the physical layer of code which allows communications with hardware devices) can be a challenge. It involves knowing the details of the device, as well as how the host processor needs to interact with the device (and vice versa) to get the desired results. This application note (and provided source code) is meant to give potential users of the AIC20 device a reference platform for evaluation and actual development. The source code is written completely in *C* to provide the ultimate in portability, readability, maintainability, and reusability. The device driver itself is packaged in a modular style so that only minor changes need to be implemented to use the driver for different hardware configurations (e.g. connecting the AIC20 device to a different peripheral of the host processor) without affecting the overall interface of the device driver.

A simple reference framework is provided as an example to demonstrate how the device driver is used in a typical digital signal processing system. We have chosen to use the popular TMS320VC5416TM DSP starter kit as the development platform. This allows the system developer to get started in a matter of minutes simply by obtaining a C5416DSK, AIC20EVM, AIC motherboard, and the source code provided with this application note.

By not starting from scratch every time a new project is commenced, DSP developers get to market quicker than starting with a blank piece of paper. Texas Instruments is fully committed to providing our DSP developers with as much off-the-shelf content as possible so that less time is spent for each system design. This strategy is implemented as TI's eXpressDSPTM software and development tool set.

# 2 Reference Platform Setup and Program Execution

This application note (and accompanying source code) allows potential AIC20 data converter users to get something up and running quickly, and it allows for evaluation of the device and the DSP algorithms which can be applied to the data stream of the device. The remainder of this application note discusses the reference framework and device driver in detail. For those who just want to get the DSK and EVM combination running with a host PC (as shown below) and not worry about the implementation details, follow the steps in this section and read the remainder of this application note as time allows to fully understand how to use the reference framework (RF) and learn how the device driver was implemented. This section describes how to configure the target hardware and set up the C5416DSK-AIC20EVM sample project code to run in real-time along with a host PC running code composer studio.

Before starting this setup procedure (starting on the following page), the following hardware components must be obtained:

- DSP Platform: Complete TMS320VC5416™ DSP starter kit (DSK)
- HPA Platform: AIC motherboard & TLV320AIC20 evaluation module
- Voice-band input device(s) (of preferred choice): microphone, handset, headset, signal tone generator, etc.†

  † If using a high-quality stereo device to simulate voice input (such as a CD player, MP3 player, Walkman, PC Soundcard, etc.) the output of the AIC20EVM will result in decreased sound quality due to the voice-band 8-kHz samplingrate.

- Voice-band output device(s) (of preferred choice): mini speakers (8 Ω), handset(150 Ω), headset(150 Ω), oscilloscope (to view output signals), etc.

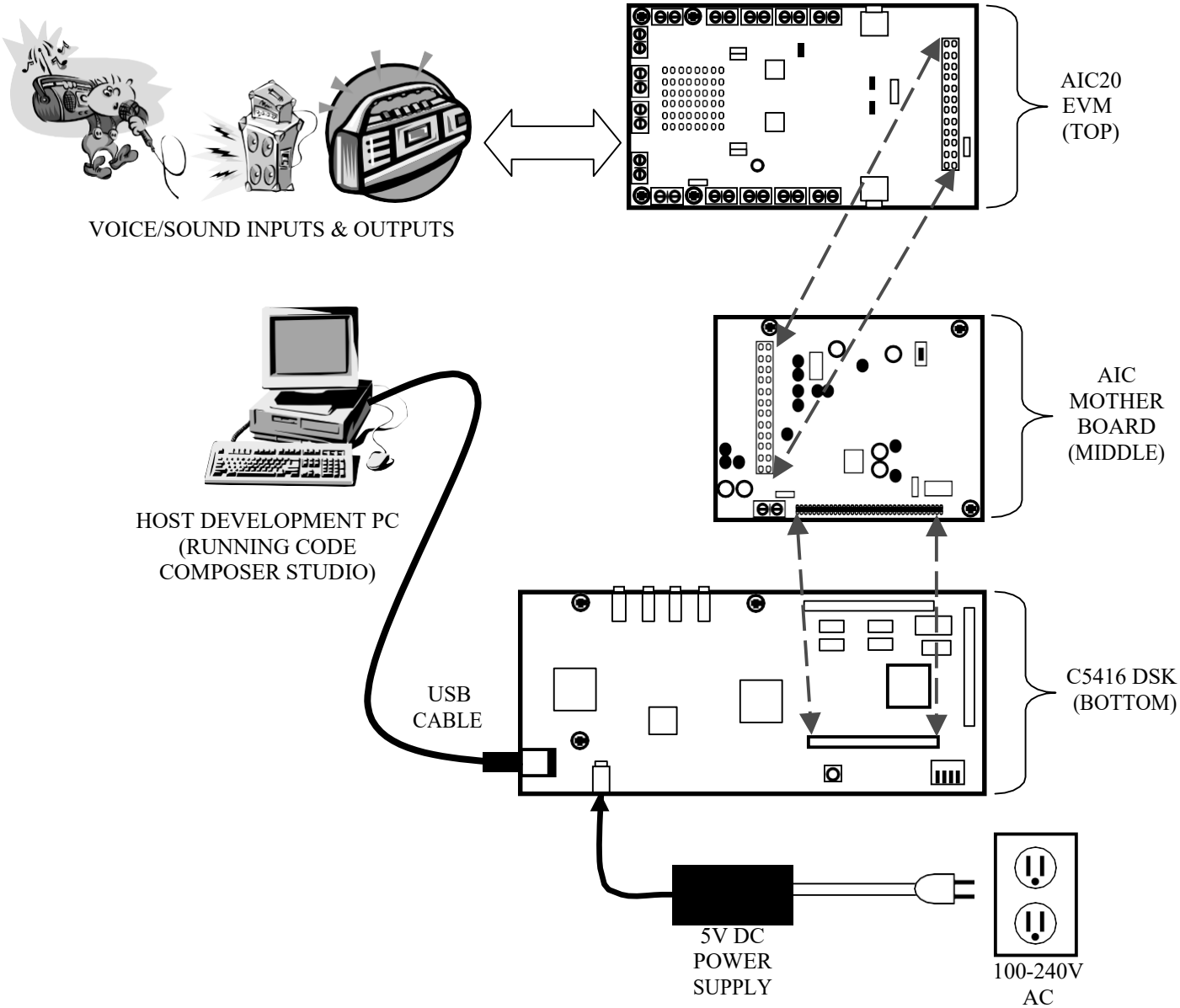


**Figure 1. DSK5416-AIC20EVM Host-Target Development Platform**

1. Download the corresponding application note source code from the same web site this application note was downloaded and extract the files onto the PC that is used as the host development platform.

2. Verify that the jumper settings on both AIC boards match the information in the following tables (use the factory default jumper settings for the DSK).

**Table 1. Configuration for AIC20EVM**

| JUMPER | POSITION | DESCRIPTION |
|---|---|---|
| W1 | Installed | Connects 3.3-V analog drive power ground to AGND (vs no connection) |
| W2 | 2 – 3 | Connects the first device's FSD to the second device's FS (vs connecting the first device's FSD to constant high[1] or low[0]) |
| W3 | Not installed | (1 – 2 connects first channel's FSD to high[1]; 2 – 3 connects FSD to low[0]) |
| W4 | 1 – 2 | Connects the first device's M/S high[1] to make it *the master* of the cascade |
| W5 | Installed | Connects analog and digital grounds together |

**Table 2. Configuration for AIC Motherboard**

| JUMPER | POSITION | DESCRIPTION |
|---|---|---|
| W1 | 1 – 2 | Codec EVM system power-up through DSK board (vs external power supply) |
| W2 | 1 – 2 | MCLK source: Use DSP's CLKOUT (vs onboard 100-MHz oscillator) |

3. Connect the AIC motherboard and AIC20EVM to the C5416DSK (using the included standoffs and screws) as shown below. The correct combination of the three boards results in a multilayered PCB interconnection with the top board being the AIC20EVM, the middle board being the AIC motherboard, and the bottom being the C5416DSK board. The C5416DSK board should be almost completely covered by the AIC20EVM and AIC motherboard when connected (from top view).

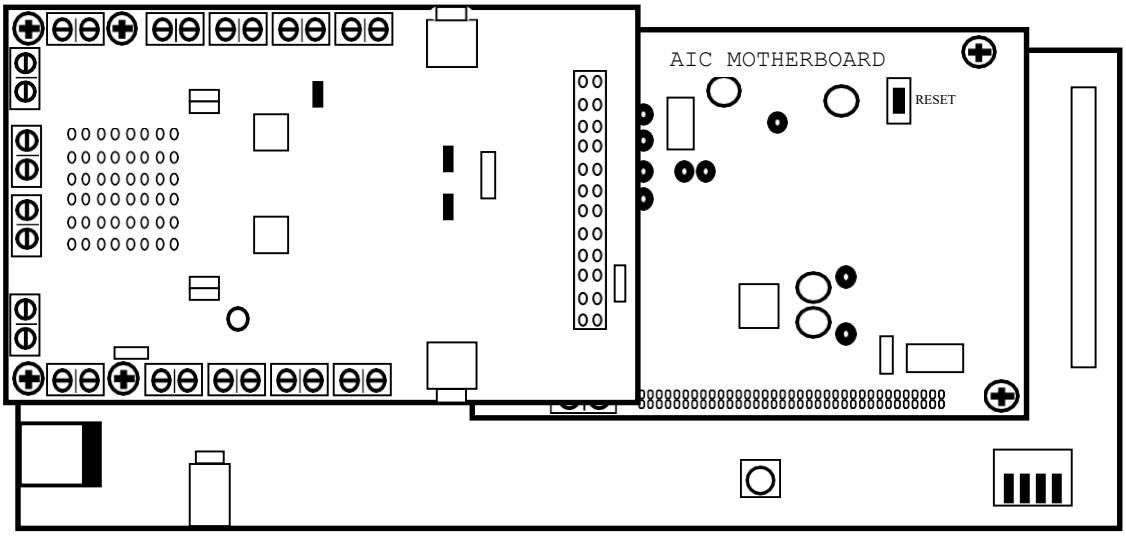


**Figure 2. DSK5416-AIC20EVM Combination (Top View)**

**NOTE**: The reference software acts as a simple pass-through of all data samples for each codec channel. At system start-up, every I/O channel is active using 8-kHz sampling rate and 16-bit data samples. To power down specific codec channels at run-time, the device driver source code needs to be modified and rebuilt. In the main driver file *dsk5416_aic20evm.c* (found in the \drivers subdirectory), locate the `DSK5416_AIC20EVM_setup()` function and modify the following code:

```
/* The following 4 lines will power down each codec channel */
//DSK5416_AIC20EVM_chanConfigParams[MST_CHAN1].reg[CR3A] |= PWDN;  // master ch1 power down
//DSK5416_AIC20EVM_chanConfigParams[MST_CHAN2].reg[CR3A] |= PWDN;  // master ch2 power down
//DSK5416_AIC20EVM_chanConfigParams[SLV_CHAN1].reg[CR3A] |= PWDN;  // slave  ch1 power down
//DSK5416_AIC20EVM_chanConfigParams[SLV_CHAN2].reg[CR3A] |= PWDN;  // slave  ch2 power down
```

To actually power down any specific codec channel, simply uncomment the line of code which corresponds to the channel to be shut down and then rebuild the *dsk5416_aic20evm_l54* (near calls/returns) and *dsk5416_aic20evm_l54f* (far) library project files found in the \drivers subdirectory (be sure to *Rebuild All* to ensure all files are built in their corresponding near or far memory models). Then, rebuild the sample application project (dsk5416_aic20evm.pjt).

4. Connect the desired voice-based devices (inputs and outputs) to the AIC20EVM connectors (input/output TB's and the input Jack) as shown below (TB = terminal block for a balanced, differential 2-wire connection; HNS = handset; HDS = headset).

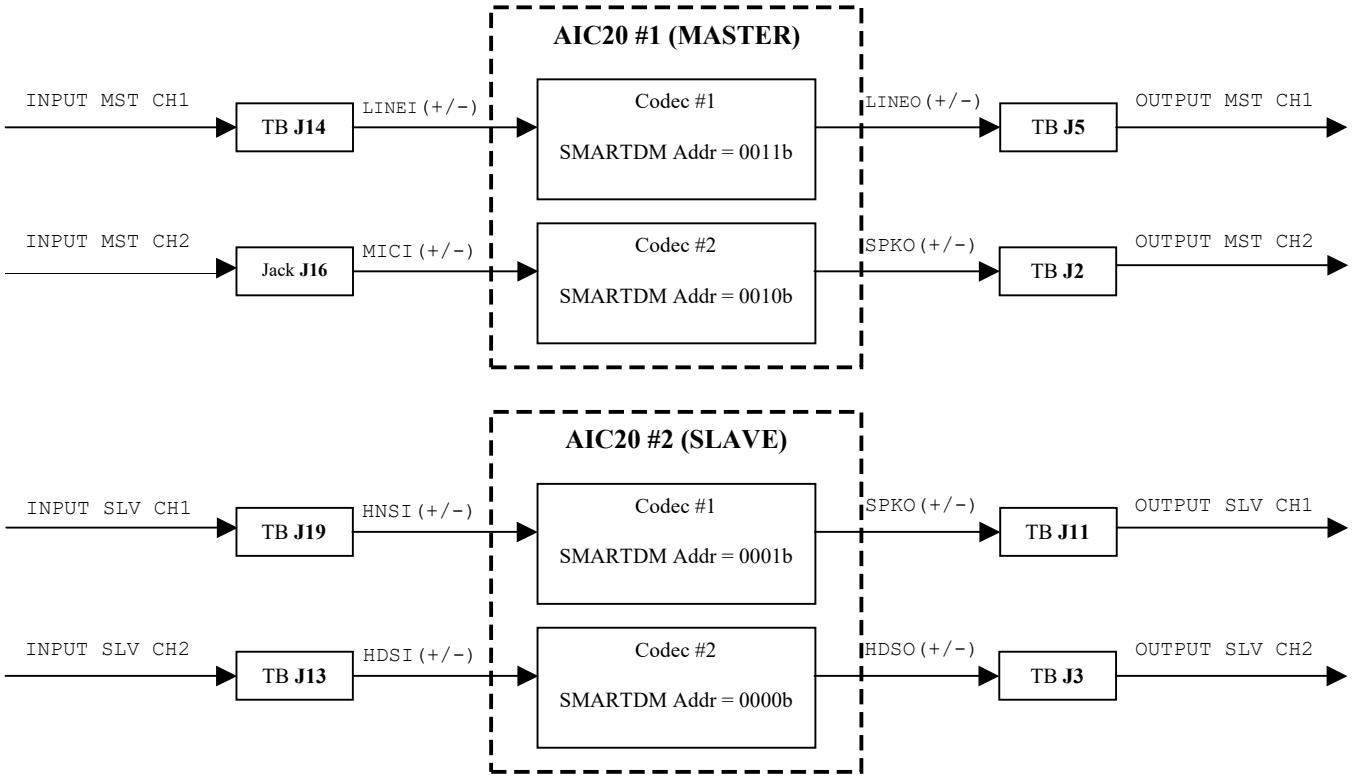


**Figure 3. AIC20EVM I/O Default Channel Selections and Connections**

**NOTE**: The polarity of the wire (+ or -) connections to the TB does not matter. The unique, self-assigned SMARTDM addresses are discussed in a later section, but they are basically used to identify which codec's control register data is being sent back to the host processor whenever the host wants to read specific control register contents of a codec.

5. Set up the C5416DSK host-target platform and invoke Code Composer Studio (C5416 DSK CCS) as per the *Quick Start Guide* that comes with the C5416DSK package. Make sure both the DSK and CCS can be started without any communications errors. If using a spectrum digital XDS-based emulator is preferred, then invoke C5000 CCS instead.

6. Load one of the following CCS workspace files (File ➔ Workspace ➔ Load Workspace):

   - *audioapp_dsk5416usb.wks* (if using the provided C5416DSK USB cable directly)
   - *audioapp_xds510pp.wks* (if using a spectrum digital XDS510-based PP emulator)
   - *audioapp_xds510usb.wks* (if using a spectrum digital XDS510-based USB emulator)

**NOTE**: For the workspace file to load properly for the XDS emulator configuration, make sure that there is a CPU named *CPU_1* when running the C5000 code composer studio setup program. If the workspace file fails to load completely, proceed to the next step.

7. Load the *audioapp.out* (File ➔ Load Program) executable (located in the *Debug* subdirectory). Start the sound source(s) on any or all of the inputs, then Debug ➔ Run the program. You should now hear the sound input(s) at the corresponding channel sound output(s).

   **WARNING:**
   **When running CCS and the C5416DSK under normal emulation mode, the JTAG channel, at times, becomes busy and causes interference to the voice channels. If this random noise is not desired or causes the analysis to be impossible, choosing the *Run Free* option (found under the CCS Toolbar *Debug* column) instructs CCS to not communicate with the target emulator while the DSP is running. The *BUSY* LED on the DSK should no longer flash during run-time (for the USB emulation configuration). Please note that none of the real-time analysis screens are able to update during this period but the target continues to run freely.**

8. To insert DSP algorithms, locate the `processBuffer()` function in the *audioapp.c* source file. Every (pointer to the) sample from every channel passes through this function. This is where the DSP processing routine(s) can be inserted to apply signal processing to any of the AIC20 channels. The default sampling rate is set for 8 kHz on all channels. The resulting processed voice-band outputs can then be analyzed in detail.

9. To learn how to leverage the existing reference framework for such things as evaluating DSP algorithms, building actual system code, writing a system-specific AIC20 device driver, learning how the DSK5416_AIC20EVM device driver was designed and implemented, or understanding how eXpressDSP™ components can be leveraged to reduce time-to-market, read the remaining sections of this application note.

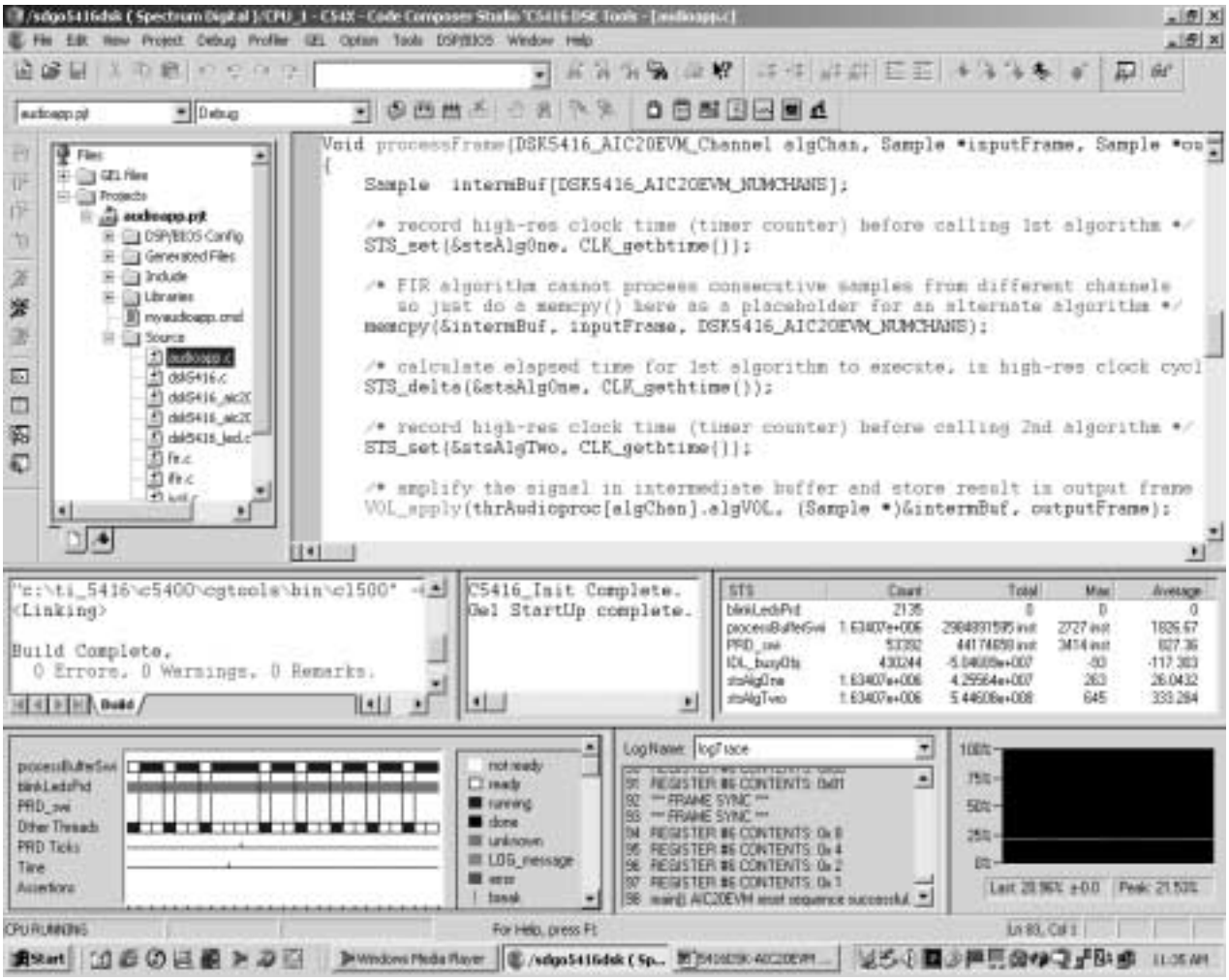


**Figure 4. Code Composer Studio Sample Workspace and Project**

**CAUTION:**

**Halting the processor and then restarting the processor could result in data words being written into control register timing slots inadvertently. Whenever the target is stopped, it is always a good idea to reset the CPU and reload the program before running the target again.**

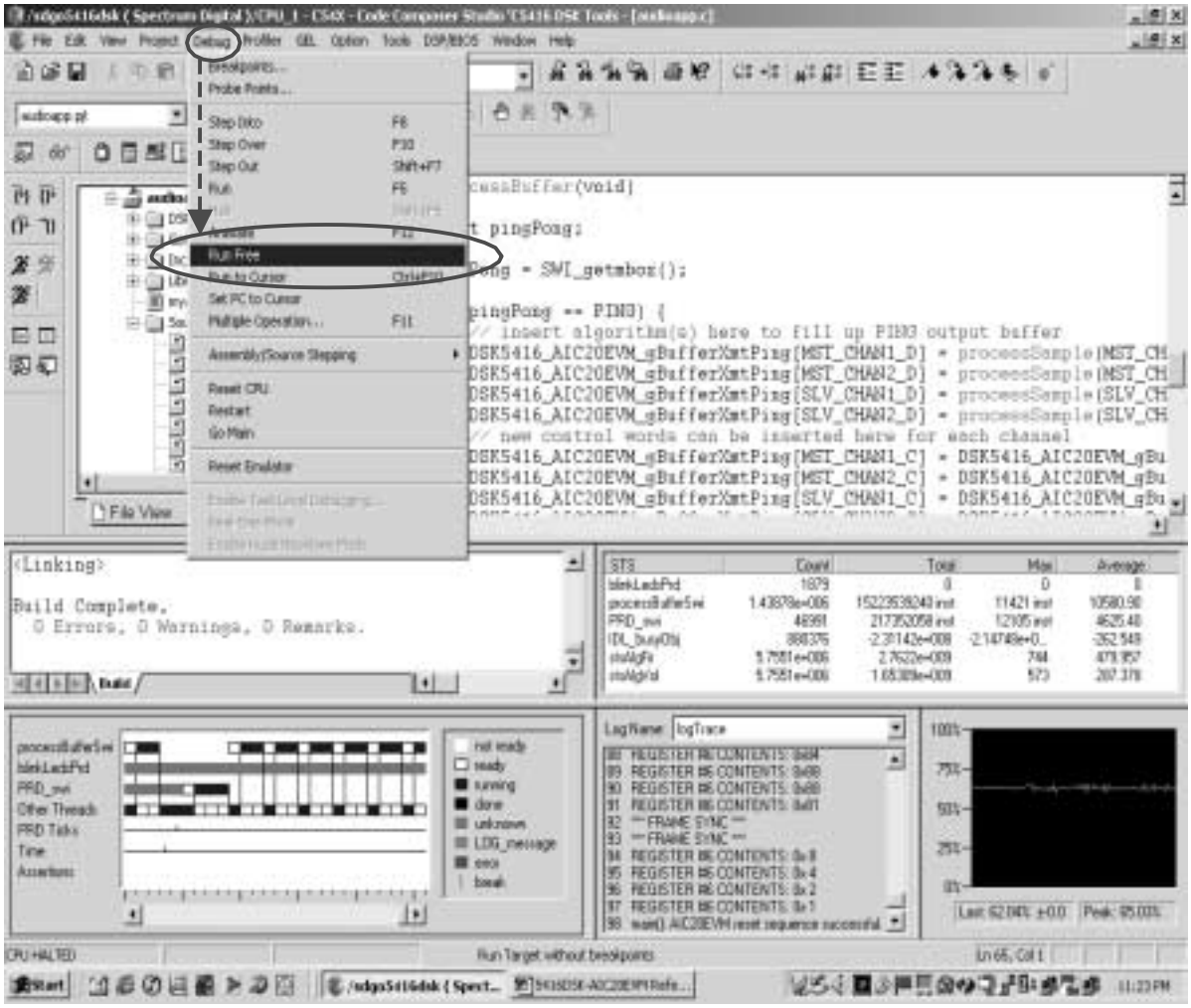


**Figure 5. Code Composer Studio *Run Free* Command**

# 3 What Is eXpressDSP™?

TI's real-time eXpressDSP™ software and development tool strategy includes three tightly knit ingredients that empower developers to tap the full potential of TMS320™ DSPs:

1. Code Composer Studio – the world's most powerful DSP integrated development environment
2. Target content software
   a. DSP/BIOS: Scalable, real-time software foundation
   b. TMS320 DSP algorithm standard (XDAIS): coding guidelines for interoperability and reuse
   c. Reference frameworks: design-ready starterware code common to many applications
3. Third party network: a growing base of TI DSP-based products that can be easily integrated into systems

Each element is designed to simplify DSP programming and move development from a custom crafted approach to a new paradigm of interoperable software from multiple vendors supported by a worldwide infrastructure. All of these components have been used in the development of the DSP reference framework and AIC20 device driver described by (and provided with) this application note.

**NOTE**: For more detailed information on all components of eXpressDSP™, please refer to TI's one-stop shop for DSP development on the Internet: www.dspvillage.com.

# 4 TMS320 DSP Algorithm Standard (XDAIS)

The TMS320™ DSP algorithm standard, also known as *XDAIS* (pronounced *DAY-yiss*), is a DSP. A single standard set of coding conventions and application programming interfaces (APIs) for algorithm creators to *wrap* the algorithm for system-ready use in any application. In the past, algorithm creators had to re-engineer an algorithm to integrate it into each different system. Now, the algorithms are written once by the creator and reused widely by the system integrators. The standard includes algorithm programming rules, which when followed by the algorithm creators, enable interoperability of compliant algorithms in the same system. Algorithm standardization increases the quantity and quality of algorithms available for faster use by OEMs. TI's third party network provides off-the-shelf compliant algorithms for ease of integration and reduced time-to-market.

All of TI's generic eXpressDSP™ reference frameworks, as well as the specific framework used in this application note, allows the developer to seamlessly integrate any algorithm which is XDAIS-compliant without having to re-engineer the algorithm module nor modify the system code to instantiate and execute the algorithms. To provide an example and to create entry points into the framework, two fully XDAIS-compliant algorithms developed by TI, the FIR_TI and VOL_TI algorithms are used in the sample framework and applied to the data stream of the AIC20 cascade. These two algorithms are easily replaced with the specific algorithms to be evaluated with the AIC20EVM.

# 5 TMS320VC5416™ DSP Starter Kit

The TMS320VC5416™ DSP starter kit (DSK) is a low-cost development platform designed to speed the development of power-efficient applications based on TI's TMS320C54x™ DSPs. The kit, which provides new performance-enhancing features such as USB communications and true plug-n-play functionality, gives both experienced and novice designers an easy way to get started immediately with their innovative product designs.

**NOTE**: For more detailed information on all components of eXpressDSP™, please refer to TI's one-stop shop for DSP development on the Internet: www.dspvillage.com.

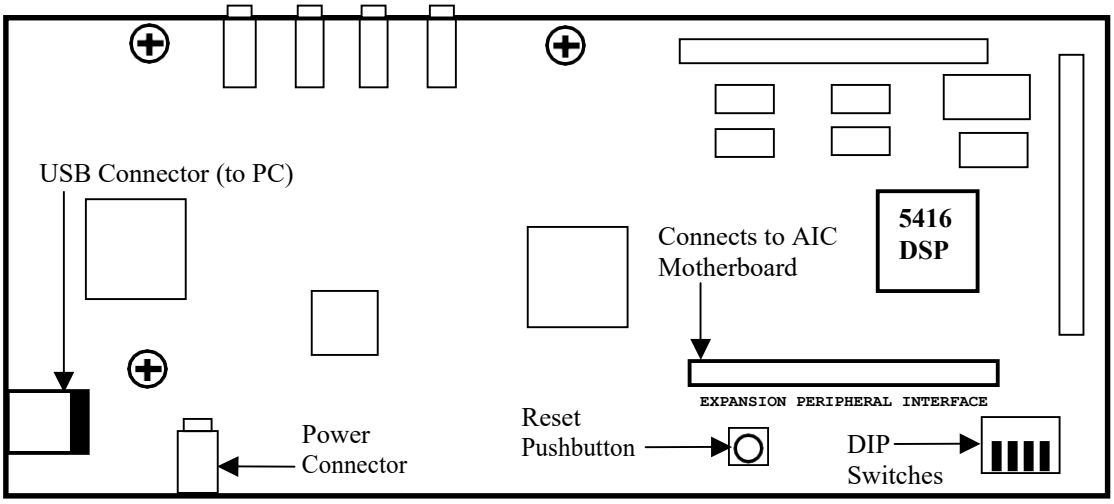


**Figure 6. TMS320VC5416™ DSP Starter Kit (DSK) Board**

## *6* TLV320AIC20EVM and the *DSP-Codec Development Platform*

The TLV320AIC20 is a true low-cost low-power highly integrated high-performance dual voice codec designed with new technological advances. It features two 16-bit analog-to-digital (A/D) channels and two 16-bit digital-to-analog (D/A) channels, which can be connected to a handset, headset, speaker, microphone, or a subscriber line via a programmable analog crosspoint. The maximum sampling rate is 26 KSPS (with on-chip IIR/FIR filter) and 104 KSPS (with IIR/FIR bypassed).

An AIC20 EVM is available to quickly evaluate the codec device. This board plugs into a generic AIC motherboard (also referred to as the *DSP-Codec Development Platform*) that plugs directly to the expansion peripheral interface (EPI) connector of the C5416 DSK. By combining these three boards, a reference platform can be used to quickly evaluate the AIC20 device as well as XDAIS-compliant algorithms used to process the data streams. The AIC expansion board allows any AIC-based EVM to be plugged directly into any DSK expansion peripheral Interface connector.

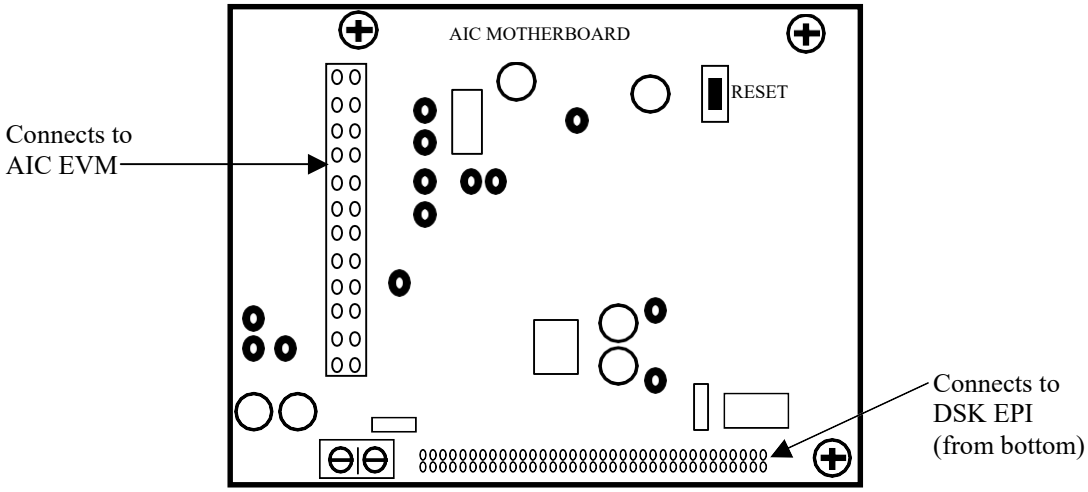


**Figure 7. AIC Motherboard (DSP-Codec Development Platform)**

The AIC20EVM board contains two AIC20 devices connected in a cascaded configuration. One device serves as the *master* while the other device is the *slave*. Each device contains two data channels, resulting in a total of four independent data channels supported on the EVM. When the EVM is connected to the DSK, the devices communicate to the C5416DSP's multichannel buffered serial port (McBSP) via time division multiplexed (TDM) stream. The SMART time division multiplexed serial port (SMARTDM) of the AIC20 uses the four wires DOUT, DIN, SCLK, and FS to transfer data into and out of the AIC20 device. The SMARTDM allows for a serial connection of up to 16 AIC20 devices to a single host serial port. The SMARTDM feature automatically adjusts the number of time slots per frame sync (FS) to match the number of codecs in the serial interface so that no time slot is wasted. Each time slot contains a 16-bit word representing sample or control information. When the *master* AIC20 device is reset, each codec in the cascade assigns itself a unique 4-bit SMARTDM address which is used to identify the time slot used for sending control register information from the codec back to the host processor.

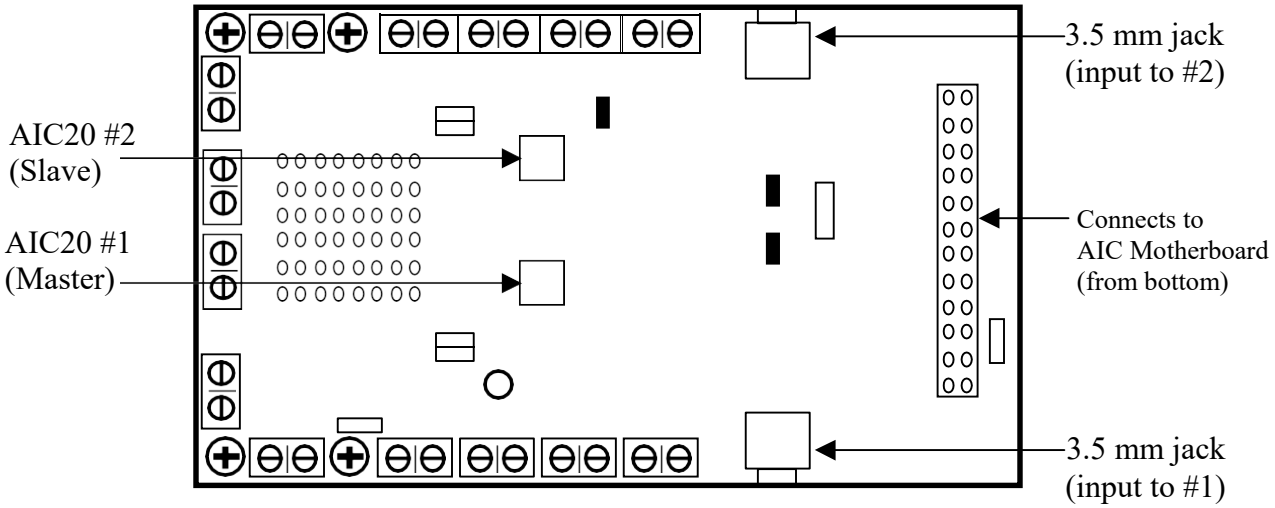


**Figure 8. AIC20 Evaluation Module (EVM)**

Figure 9 shows how the 2 AIC20 devices on the AIC20EVM connect to the DSP in the C5416DSK. The AIC20 closest to the DSP's McBSP is the *Master device* that provides the FS signal to the DSP. The FS acts as a signal to the DSP so that it knows when to write and read data to/from the correct TDM slot within the FS period. On the falling edge of the FS signal should be the read or write from/to the first *Master* channel's slot. Figure 9 shows the slot within the FS period that corresponds to the channels in the cascade. This configuration allows a single McBSP to talk to any number of cascaded AIC20 devices, up to a maximum of eight devices (each AIC20 device supports two codec channels, resulting in a maximum of ($8 \times 2$) = 16 time division multiplexed (TDM) channels in a single serial data stream).

**NOTE**: Up to four AIC20EVMs can be stacked on top of the AIC motherboard to achieve the 16-TDM cascaded channel configuration, but requires minor modifications to the existing device driver as the provided driver is configured for a single AIC20EVM (two devices / four channels).

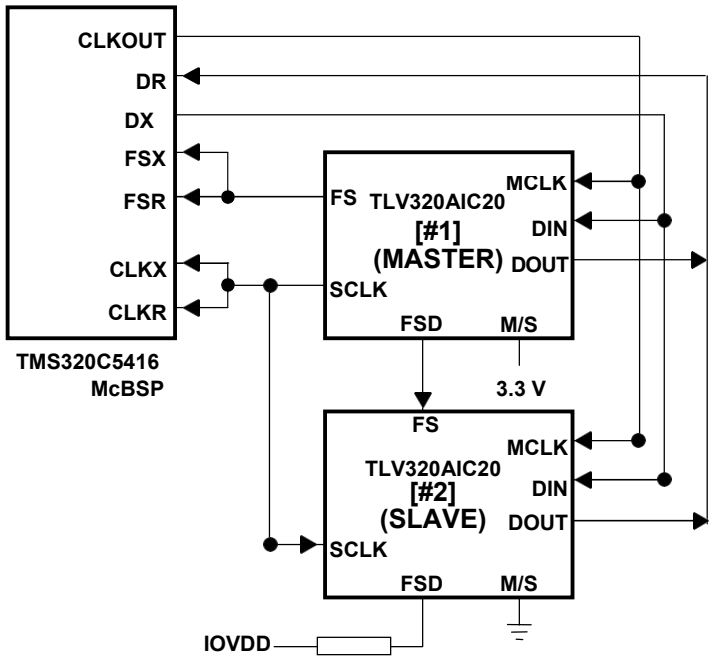


**Figure 9. 2-Device (4-Channel) Cascade Connection to Host Processor Serial Port**

# 7 Software Reference Framework

DSP/BIOS is TI's real time operating system foundation. It provides multiple thread scheduling, memory management services, hardware abstraction/configuration, real-time analysis capabilities, interprocess communication, and structured device driver I/O.

Applications which use DSP/BIOS take advantage of all the common DSP real-time kernel services with easy-to-use API's and hand-optimized program modules for increased portability, maintainability, reusability, and reduced time-to-market.

Accelerating the software development process for designers of DSP-based applications, TI produces and supports a series of DSP software reference frameworks (RF). The design-ready RFs are getting-started solutions for designers in the early stages of application development, featuring easy-to-use source code that is common to many applications. With TI's RFs, many of the initial low-level design decisions are eliminated, allowing developers more time to focus on the code that truly differentiates products. Designers can choose the specific RF that best meets their system needs and then populate the RF with XDAIS-compliant algorithms, creating specific applications for a wide range of end equipments.

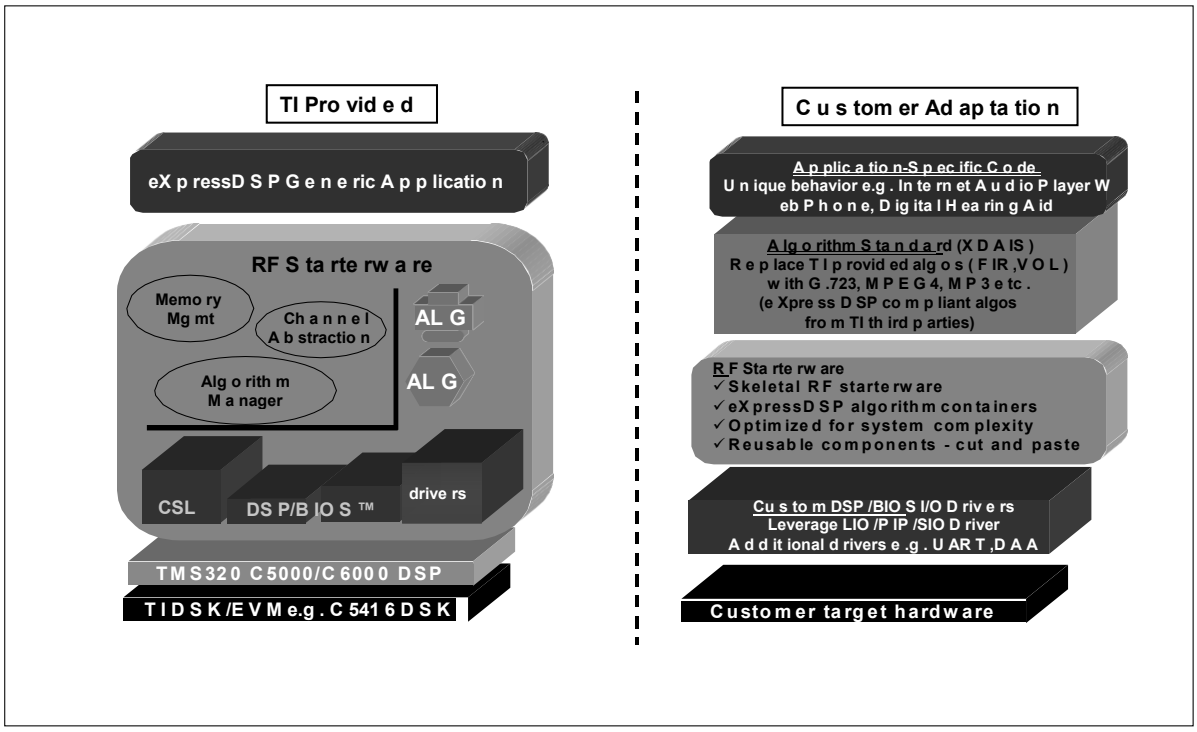


**Figure 10. eXpressDSP™ Reference Framework Architecture**

The reference framework used in this application note is built on the same DSP/BIOS foundation and allows the developer to easily insert various XDAIS algorithms to evaluate digital signal processing on the data channels as well. There is also a simulated host control capability where a control thread is periodically scheduled to run on the target and checks a shared *device I/O area* memory space. If the host sets certain flags, the target can perform the appropriate function during run-time. GEL sliders are provided as an example to change certain parameters during run-time.

## 7.1 Data Channel Processing Threads

In this application note example, a single thread of execution is used to read a single sample from each of the AIC20 ADC's at a time, and then each sample is subsequently sent back out to the same channel's DAC. Using this convenient entry point, any number of DSP algorithms can be inserted to apply sample-by-sample or frame-based DSP processing on every channel. The AIC20 device driver exposes a pair of receive and transmit *ping-pong* buffers to pass data between itself and the framework (this is discussed in greater detail in the device driver section of this application note).

The *audioapp.c* source file contains the main framework code. The `processBuffer()` function is automatically called every time a sample from all AIC20 channels have been read at the end of each FS period (i.e. Rcv *ping* or *pong* buffer has just been filled up by the device driver). The `processBuffer()` can be configured for either sample-by-sample processing or frame-based processing. It gets samples from the currently filled Rx buffer and place them in the corresponding Tx *ping-pong* buffer. This is where the processing is applied to each sample (or the entire frame of data samples) as the Rx data is being transferred to the Tx buffer by the CPU.

### 7.1.1 Sample-by-Sample Processing

```
void processBuffer(void)
{
   short pingPong;

   pingPong = SWI_getmbox();

   if (pingPong == PING) {
   // insert algorithm(s) here to fill up PING output buffer
      DSK5416_AIC20EVM_gBufferXmtPing[MST_CHAN1_D] =
         processSample(MST_CHAN1, &DSK5416_AIC20EVM_gBufferRcvPing[MST_CHAN1_D]); // MST Ch1
      DSK5416_AIC20EVM_gBufferXmtPing[MST_CHAN2_D] =
         processSample(MST_CHAN2, &DSK5416_AIC20EVM_gBufferRcvPing[MST_CHAN2_D]); // MST Ch2
      DSK5416_AIC20EVM_gBufferXmtPing[SLV_CHAN1_D] =
         processSample(SLV_CHAN1, &DSK5416_AIC20EVM_gBufferRcvPing[SLV_CHAN1_D]); // SLV Ch1
      DSK5416_AIC20EVM_gBufferXmtPing[SLV_CHAN2_D] =
         processSample(SLV_CHAN2, &DSK5416_AIC20EVM_gBufferRcvPing[SLV_CHAN2_D]); // SLV Ch2
   }
   else { // pingPong == PONG
      // insert algorithm(s) here to fill up PONG output buffer
      DSK5416_AIC20EVM_gBufferXmtPong[MST_CHAN1_D] =
         processSample(MST_CHAN1, &DSK5416_AIC20EVM_gBufferRcvPong[MST_CHAN1_D]); // MST Ch1
      DSK5416_AIC20EVM_gBufferXmtPong[MST_CHAN2_D] =
         processSample(MST_CHAN2, &DSK5416_AIC20EVM_gBufferRcvPong[MST_CHAN2_D]); // MST Ch2
      DSK5416_AIC20EVM_gBufferXmtPong[SLV_CHAN1_D] =
         processSample(SLV_CHAN1, &DSK5416_AIC20EVM_gBufferRcvPong[SLV_CHAN1_D]); // SLV Ch1
      DSK5416_AIC20EVM_gBufferXmtPong[SLV_CHAN2_D] =
         processSample(SLV_CHAN2, &DSK5416_AIC20EVM_gBufferRcvPong[SLV_CHAN2_D]); // SLV Ch2
   }
}
```

In this case, the `processSample()` function is called on every new sample that is read into the receive data buffer that has just been filled by the device driver. This is where one or more algorithms can be applied to the sample before it is written to the output buffer. The sample framework comes with simple, fully XDAIS-compliant finite impulse response (FIR) and volume/gain control (VOL) algorithms (developed by TI) that are applied in sequence to every sample received from each data channel.

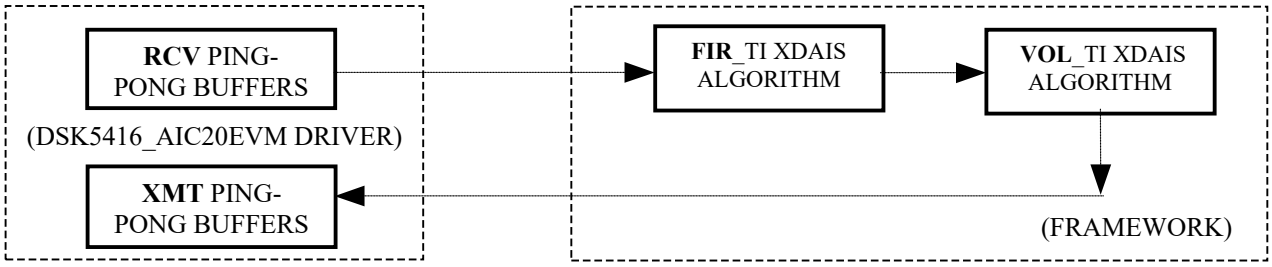


**Figure 11. Framework Channels: Data Flow**

These entry points serve as placeholders for the *real* algorithms that could be inserted and evaluated with the AIC20EVM data streams. Most XDAIS algorithms come packaged with `<ALGORITHM>_apply()` functions which in most cases can be inserted at the `FIR_apply()` and `VOL_apply()` entry points. XDAIS algorithms allow for ease of integration, especially when swapping out one XDAIS algorithm module for another.

```
Uint16 processSample(DSK5416_AIC20EVM_Channel chan, Uint16 *inputSample)
{
   Sample   tmp1, tmp2;

   STS_set(&stsAlgFir, CLK_gethtime());
   FIR_apply(thrAudioproc[chan].algFIR, (Sample *)inputSample, &tmp1);
   STS_delta(&stsAlgFir, CLK_gethtime()); // measure FIR algorithm execution time

   STS_set(&stsAlgVol, CLK_gethtime());
   VOL_apply(thrAudioproc[chan].algVOL, &tmp1, &tmp2);
   STS_delta(&stsAlgFir, CLK_gethtime()); // measure VOL algorithm execution time

   return ((Uint16)tmp2);
}
```

### *7.1.2 Frame-Based Processing*

To reduce the overhead of calling the same processing function(s) on every sample on every channel and/or to simply evaluate the same algorithm processing on all cascaded channels, a `processFrame()` function is also supplied as another option for the framework:

```
void processBuffer(void)
{
   short pingPong;

   pingPong = SWI_getmbox();

   if (pingPong == PING) { // Fill up Xmt Ping output buffer
      processFrame(MST_CHAN1, (Sample *)&DSK5416_AIC20EVM_gBufferRcvPing[MST_CHAN1_D],
                              (Sample *)&DSK5416_AIC20EVM_gBufferXmtPing[MST_CHAN1_D]);
   }
   else { // pingPong == PONG
      // Fill up Xmt Pong output buffer
      processFrame(MST_CHAN1, (Sample *)&DSK5416_AIC20EVM_gBufferRcvPong[MST_CHAN1_D],
                              (Sample *)&DSK5416_AIC20EVM_gBufferXmtPong[MST_CHAN1_D]);
   }
}
```

In this case, `processFrame()` can run any algorithm that supports frame-based or block-oriented processing. In other words, the current samples from every channel are treated as a single frame of multiple samples and can be processed by a single function call, to apply the same processing function to each channel from the current FS period. In this application note example, the DSP CPU loading is normally reduced by up to 23% when switching from sample-by-sample processing to frame-based processing.

**NOTE**: If choosing the `processFrame()` option, be sure to only use an algorithm that can process each sample in a frame independently from the other samples in the frame. The FIR_TI algorithm will only work using the `processSample()` option since each frame of data contains samples from different channels, since a typical filter-type algorithm needs to operate on consecutive samples from the same sound stream.

```
Void processFrame(DSK5416_AIC20EVM_Channel algChan,
                  Sample *inputFrame, Sample *outputFrame)
{
   Sample            intermBuf[DSK5416_AIC20EVM_NUMCHANS];

   /* record high-res clock time (timer counter) before calling 1st algorithm */
   STS_set(&stsAlgOne, CLK_gethtime());

   /* FIR algorithm cannot process consecutive samples from different channels
      so just do a memcpy() here as a placeholder for an alternate algorithm */
   memcpy(&intermBuf, inputFrame, DSK5416_AIC20EVM_NUMCHANS*sizeof(Sample));

   /* calculate elapsed time for 1st algorithm to execute, in high-res clock cycles */
   STS_delta(&stsAlgOne, CLK_gethtime());

   /* record high-res clock time (timer counter) before calling 2nd algorithm */
   STS_set(&stsAlgTwo, CLK_gethtime());

   /* amplify the signal in interm. buffer and store result in output frame buffer */
   VOL_apply(thrAudioproc[algChan].algVOL, (Sample *)&intermBuf, outputFrame);

   /* calculate elapsed time for 2nd algorithm to execute, in high-res clock cycles */
   STS_delta(&stsAlgTwo, CLK_gethtime());
}
```

On return from the `processFrame()` function call, the processed samples filled up the corresponding transmit *ping* or *pong* output buffer and be sent out to the device driver. For best results, XDAIS-compliant algorithms should be used for ease of integration and interoperability, especially when integrating algorithms from multiple sources/vendors.

## 7.2 Data Channel State Objects

A global array of channel structures (named `thrAudioproc[]` of data structure type `ThrAudioproc`) is declared during compile time and is initialized during run-time. Once initialized, the framework code can access each of the data channel's state information at any time. Currently, these channel structures store the unique algorithm instance objects that are used for the processing of each channel. The structure definition is found in the *thrAudioproc.h* header file, and can be modified to include any additional channel state information as required by the application developer.

```
/*
 *  Here we define a structure that contains all the "private"
 *  thread information: algorithm handles, input pipe(s), output
 *  pipe(s), intermediate buffer(s), if any, and all the other
 *  information that encapsulates thread state for each channel.
 */
typedef struct ThrAudioproc {
    /* algorithm(s) */
    FIR_Handle  algFIR;     /* an instance of the FIR algorithm */
    VOL_Handle  algVOL;     /* an instance of the VOL algorithm */

    /* everything else that is private for a thread comes here */

} ThrAudioproc;
```

```
ThrAudioproc thrAudioproc[ DSK5416_AIC20EVM_NUMCHANS ] = {
    { /* data channel #1 (Master Channel 1) */
        /* algorithm handle(s) (to be initialized in runtime) */
        NULL,                   /* algFIR */
        NULL,                   /* algVOL */

        /* everything else private for the thread */

    }, /* end data channel #1 */

    { /* data channel #2 (Master Channel 2) */
        /* algorithm handle(s) (to be initialized in runtime) */
        NULL,                   /* algFIR */
        NULL,                   /* algVOL */

        /* everything else private for the thread */

    }, /* end data channel #2 */

    { /* data channel #3 (Slave Channel 1) */
        /* algorithm handle(s) (to be initialized in runtime) */
        NULL,                   /* algFIR */
        NULL,                   /* algVOL */

        /* everything else private for the thread */

    }, /* end data channel #3 */

    { /* data channel #4 (Slave Channel 2) */
        /* algorithm handle(s) (to be initialized in runtime) */
        NULL,                   /* algFIR */
        NULL,                   /* algVOL */

        /* everything else private for the thread */

    }, /* end data channel #4 */
};
```

The above code shows the global `thrAudioproc[]` array defined in source code and its ability to take on default settings within each channel structure. Each array element (channel structure) corresponds to one of the data channels on the AIC20EVM. Fields such as the algorithm handles are set during run-time since the XDAIS algorithms in this example are created and initialized during system start-up, and additional structure fields can be added as needed.

## 7.3 Data Channel Algorithm Creation

The `thrAudioprocinit()` function references the `thrAudioproc[]` array and take care of querying the XDAIS algorithms for their memory requirements, dynamically allocate those memories from internal and/or external memory heaps defined by the user, and store the handles to the newly-created algorithm instance objects so that each channel can reference its own set of algorithm instances.

```
Void thrAudioprocInit( Void )
{
    /* declaration of filter, volume parameter structures */
    FIR_Params firParams;
    VOL_Params volParams;
    Int i;

    for (i = 0; i < DSK5416_AIC20EVM_NUMCHANS; i++) {
        /*
         *  Set the parameters structure to the default, i.e.
         *  the one used in i<alg>.c, and modify fields that are different.
         */
        firParams = FIR_PARAMS;             /* default parameters */
        firParams.coeffPtr   =              /* filter coefficients */
            (Short *)filterCoefficients[i];
        firParams.filterLen  =              /* filter size */
            sizeof( filterCoefficients[i] ) / sizeof( Sample );
        firParams.frameLen = 1;             /* frame size */

        /* create algorithm instance for channel #i */
        thrAudioproc[i].algFIR = FIR_create( &FIR_IFIR, &firParams );

        /*
         *  Confirm that the instantiation was successful. If it failed,
         *  most likely the heap is not big enough. To find out the needed
         *  value (rather than to guess), in appThreads.c you can do
         *  ALGRF_setup( EXTERNALHEAP, EXTERNALHEAP ); i.e. force all
         *  allocation in external memory, run the initialization functions,
         *  and examine the reports from UTL_showAlgMem() below.
         */
        UTL_assert( thrAudioproc[i].algFIR != NULL );

        /* and show algorithm memory usage */
        UTL_showAlgMem( thrAudioproc[i].algFIR );

        /* do the same for the VOLume algorithm: create parameters structure */
        volParams = VOL_PARAMS;             /* default parameters */
        volParams.frameSize = DSK5416_AIC20EVM_NUMCHANS; /* frame size */

        /* create instance, confirm creation success, show memory usage */
        thrAudioproc[i].algVOL = VOL_create( &VOL_IVOL, &volParams );
        UTL_assert( thrAudioproc[i].algVOL != NULL );
        UTL_showAlgMem( thrAudioproc[i].algVOL );
    }
}
```

The `FIR_create()` and `VOL_create()` are XDAIS standardized *`<ALGORITHM>_create()` wrapper functions* which are called to automate the process of dynamically creating a XDAIS algorithm instance object pertaining to the specific algorithm which is referenced by the <ALGORITHM> designation. Since all XDAIS-compliant algorithms implement a standard interface for algorithm instance creation, each `<ALGORITHM>_create()` function references the same generic `ALGRF_create()` function that is implemented by the RF ALGRF standard library that can be used to instantiate any XDAIS-compliant algorithm. The creation parameters for the algorithm instances are also set in this function – i.e. all algorithm create-type code is bundled in this single function which is called once during system initialization.

## 7.4 System-Specific Initialization

Finally, the `main()` function contains all of the one-time system initialization code. In a DSP/BIOS application, the `main()` function is called once and must return to give control over to the DSP/BIOS scheduler. The DSK board and device driver (and all DSP peripherals associated with the device) need to be initialized once within `main()`.

Once `main()` has finished execution and returns, the DSP/BIOS scheduler takes control over the system and is ready to service hardware/software interrupts and execute tasks and background functions. Any additional *run once code* should be added to `main()` since it only runs once in a DSP/BIOS system on reset.

```
Void main()
{
   // Initialize the DSK Board
   DSK5416_init();

   // Initialize the AIC20EVM Device Driver as a whole
   if (DSK5416_AIC20EVM_init()) {
      DSK5416_AIC20EVM_setup();
      DSK5416_AIC20EVM_hDevice = DSK5416_AIC20EVM_open();
      LOG_printf(&logTrace, "main(): AIC20EVM reset sequence successful.\n");
   }
   else {
      LOG_printf(&logTrace, "main(): Could not establish presence of AIC Motherboard!!!\n");
      SYS_exit(0);
   }

   // Initialize the XDAIS algorithm modules as a whole
   FIR_init();
   VOL_init();

   // Create the algorithm instances for each channel state structure
   thrAudioprocInit();

   // Return and drop into the DSP/BIOS environment
}
```

The `FIR_init()` and `VOL_init()` are XDAIS standardized *`<ALGORITHM>_init()` master initialization functions* which are called to initialize the XDAIS algorithm modules as a whole during system initialization, before any XDAIS algorithm instances are created in the system. The `thrAudioprocInit()` function, as described in the previous section, is used to instantiate the channel state objects representing each data stream in the system.

## 7.5 Algorithm Benchmarking

This RF includes two DSP/BIOS statistics (STS) objects used to benchmark the FIR_TI and VOL_TI algorithm performance. Each STS object accumulates the following statistical information about an arbitrary 32-bit wide data series:

- **Count.** The number of values in an application-supplied data series
- **Total.** The sum of the individual data values in this series
- **Maximum.** The largest value already encountered in this series

Using the count and total, the CCS statistics view plug-in calculates the average on the host. Additional custom STS objects are added to the system using the DSP/BIOS configuration file (*.CDB) which is part of the CCS project. The following STS run-time API's allow the target application to maintain the various statistics:

- `STS_add()` – updates the count, total, and maximum using the value provided
- `STS_set()` – sets a previous value for reference
- `STS_delta()` -- accumulates the difference between the value currently passed in and the previous value which was set by the most recent call to `STS_set()` or `STS_reset()`

By using custom STS objects and various combinations of STS operations, the following statistics can be computed automatically:

- Count the number of occurrences of an event
- Track the maximum and average values for a variable in the program
- Track the minimum value for a variable in the program
- Time events or monitor incremental differences in a value
- Monitor differences between actual values and desired values

The following code sample uses the STS operations to programmatically accumulate the amount of instruction cycles elapsed by using paired `STS_set()` & `STS_delta()` function calls around each algorithm function call. The `CLK_gethtime()` function is a DSP/BIOS API used to read the current value of the high-resolution timer counter; thus the unit of measurement is the number of instruction cycles.

In this case, the STS *object stsAlgOne* is used to store statistics each time the `FIR_apply()` function is called, and the *stsAlgTwo* is used to benchmark the `VOL_apply()` algorithm execution times.

```
/* record high-res clock time (timer counter) before calling FIR algorithm */
STS_set(&stsAlgOne, CLK_gethtime());

/* apply filter and store result in temp buffer */
FIR_apply(thrAudioproc[chan].algFIR, (Sample *)inputSample, &tmp1);

/* calculate elapsed time for FIR algorithm to execute, in high-res clock cycles */
STS_delta(&stsAlgOne, CLK_gethtime());

/* record high-res clock time (timer counter) before calling 2nd algorithm */
STS_set(&stsAlgTwo, CLK_gethtime());

/* amplify the signal and store result in temp buffer */
VOL_apply(thrAudioproc[chan].algVOL, &tmp1, &tmp2);

/* calculate elapsed time for 2nd algorithm to execute, in high-res clock cycles */
STS_delta(&stsAlgTwo, CLK_gethtime());
```

The statistics are viewed in real-time with the statistics view plug-in by choosing the CCS DSP/BIOS ➔ Statistics View menu item. The reference framework project already comes with the Statistics View window open and configured to show the statistics for both of the included STS objects (stsAlgOne and stsAlgTwo). More STS objects can be inserted to benchmark other portions of system code as needed. To create an STS object, right-click the *STS – Statistics Object Manager* icon and select *Insert STS*. Right-click on the newly created STS object and select *Rename* to give the STS object a meaningful name. This is the name of the STS object used in the corresponding programmatic calls to the STS API's in the system code to gather statistics during run-time of the system without ever halting the target processor. The statistics data is sent from the target to the host only during CPU idle time using a host-target communications technology called real-time data exchange (RTDX™). DSP/BIOS real-time analysis data is always transferred via RTDX which is completely nonintrusive and never breaks the real-time processing functionality of the DSP system.

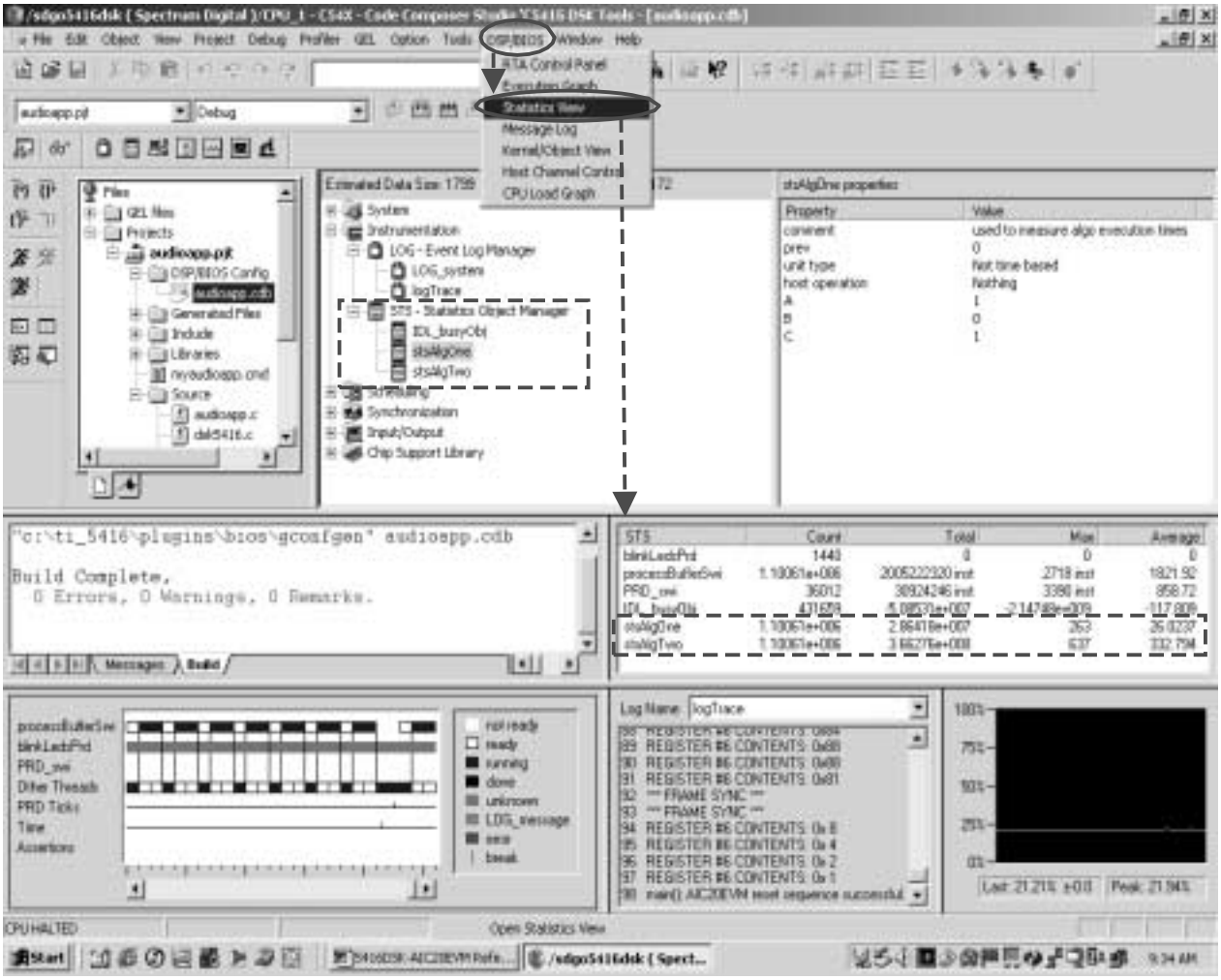


**Figure 12. Configuring and Viewing DSP/BIOS Statistics (STS) Objects**

# 8 *DSK5416_AIC20EVM* Device Driver

## 8.1 Requirements for Writing the Device Driver

In general, writing a device driver requires detailed knowledge of both the host processor and the device itself. Without understanding how the host processor and device interface to each other and the exact timing of communication between the two, writing the device driver can be a very difficult task, especially when one does not have an expensive logic analyzer and other sophisticated instruments. The device driver should also implement a modular and easy-to-understand interface. The baseline driver developed here is the *DSK5416_AIC20EVM* driver.

### *8.1.1 Host Processor Considerations and Configuration*

In this case, the host processor is a TMS320VC5416 DSP with three on-chip serial ports, or multichannel buffered serial ports (McBSPs). Each McBSP is bi-directional (i.e. capable of receiving and transmitting data simultaneously using the same port, therefore, a single McBSP is used to communicate with the device, which in this case is an AIC20EVM. When the AIC20EVM is plugged into the C5416DSK, all device lines connect to McBSP #1 of the DSP. Refer to Figure 9 to see exactly which lines are connected between the host and cascade of AIC20 devices. In addition, the McBSP receive mode must be set for 1-bit delay since the AIC20 always responds with its DOUT data delayed by 1 bit for every word.

### *8.1.2 AIC20EVM Device Cautions*

To the host processor, the AIC20EVM is a single device in the system. The AIC20EVM contains two AIC20 devices connected in a cascade configuration. One device acts as the master while the other acts as the slave. The master device is the AIC20 closest to the DSP. The master AIC20 device provides a *Frame Sync* signal to the DSP so that the DSP knows when a complete frame of data has been received from the AIC20EVM. Within this FS period, there are four data and four control timing slots. Each slot corresponds to a specific channel within the overall AIC20 cascade of devices. It is important that the device driver reads and writes data from/to the correct timing slot; otherwise the host processor will be communicating the wrong data to the wrong channel. These eight timing slots per FS period make up a time-division multiplexed data stream – i.e. each channel reads/writes data at a specific time slot within the overall FS period.

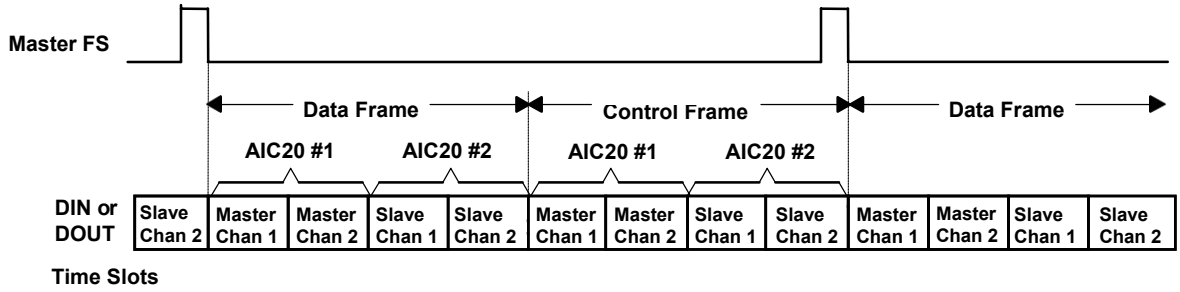


**Figure 13. Time Division Multiplexing: Slot Assignment for Data and Control Words**

## 8.2 Defining the Interface to the Device Driver

The details of how the host and device interact with one another help to determine the specific interface of the device driver. The interface should be easy to understand, easy to use, and present a modular solution to encapsulate and abstract as much detail as possible from the application framework.

### *8.2.1 Framework Interaction with the Driver*

The following diagram shows how a framework interacts with the *DSK5416_AIC20EVM* device driver module. The relevant data structures and functions are shown in the diagram:

- Rx and Tx ping-pong buffers: Frame buffers used to read data from the device driver and store data to be sent out by the device driver
- Driver functions: APIs to initialize, execute, and close the device driver
- Channel configuration array: an array of configuration parameters used to set the attributes for each individual channel of the AIC20 cascade

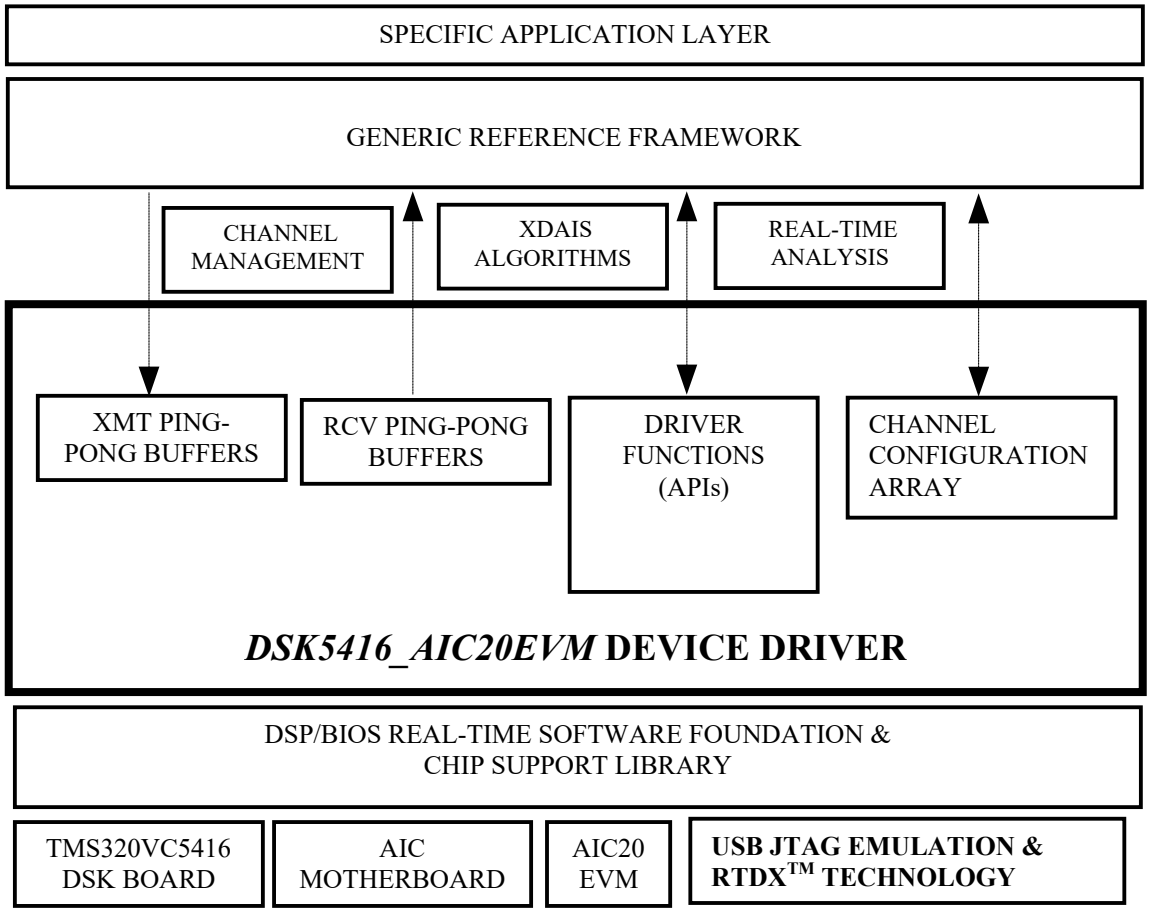


**Figure 14. Reference Platform: Hardware and Software Architecture**

### 8.2.2 Driver Functions

There are a minimum of three functions that need to be invoked by the framework. The function prototypes can be found in the device driver's header file <dsk5416_aic20evm.h>.

- `DSK5416_AIC20EVM_init()` – needs to be called only once during system initialization
- `DSK5416_AIC20EVM_setup()` – needs to be called to set up channel configuration parameters before opening the device
- `DSK5416_AIC20EVM_open()` – needs to be called to physically set up and start the device after the init() and setup() functions have been executed
- `DSK5416_AIC20EVM_close()` – can be called by the framework to power down the entire AIC20 device cascade for system shutdown purposes

### 8.2.3 Relevant Data Structures

#### 8.2.3.1 Ping-Pong Buffers

The *DSK5416_AIC20EVM* device driver defines four global buffers used to pass data between device and framework. The typical ping-pong buffering scheme is implemented, meaning there are two receive buffers and two transmit buffers. When one receive buffer fills up, the driver begins to fill the other receive buffer. Similarly, when the framework wants to output data to the device, it should switch back and forth between transmit buffers each time a buffer becomes full.

| ARRAY INDEX | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | SLAVE CHAN 2 CONTROL REG | MASTER CHAN 1 DATA WORD | MASTER CHAN 2 DATA WORD | SLAVE CHAN 1 DATA WORD | SLAVE CHAN 2 DATA WORD | MASTER CHAN 1 CONTROL REG | MASTER CHAN 2 CONTROL REG | SLAVE CHAN 1 CONTROL REG |
| | SLV_CHAN2_C | MST_CHAN1_D | MST_CHAN2_D | SLV_CHAN1_D | SLV_CHAN2_D | MST_CHAN1_C | MST_CHAN2_C | SLV_CHAN1_C |

**Figure 15. RCV and XMT Ping-Pong Buffer Format**

The following enumerated types, defined in the header file <dsk5416_aic20evm.h>, are used to locate specific channel information within each buffer, rather than trying to remember which time slot corresponds to which channel's data and control information:

```
/* Enumerated types for array locations in the DSK5416_AIC20EVM buffers */
typedef enum DSK5416_AIC20EVM_BufferIndex {
  SLV_CHAN2_C, // "Slave" Channel 2 CTRL slot
  MST_CHAN1_D, // "Master" Channel 1 DATA slot
  MST_CHAN2_D, // "Master" Channel 2 DATA slot
  SLV_CHAN1_D, // "Slave" Channel 1 DATA slot
  SLV_CHAN2_D, // "Slave" Channel 2 DATA slot
  MST_CHAN1_C, // "Master" Channel 1 CTRL slot
  MST_CHAN2_C, // "Master" Channel 2 CTRL slot
  SLV_CHAN1_C // "Slave" Channel 1 CTRL slot
} DSK5416_AIC20EVM_BufferIndex;
```

### 8.2.3.2 Channel Configuration Array

Within the device driver interface, there is an array (of size *DSK5416_AIC20EVM_NUMCHANS*) of channel configuration parameters that is declared globally so that both framework and device driver can access them. Each element of the array is just a structure that contains all of the possible control register settings that pertain to a specific channel. The device driver initializes each channel's structure with a set of common default values during compile time. However, there are certain parameters that cannot be the same for each channel, such as the input (ADC) and output (DAC) settings. Before the device driver sends out these control register settings during device initialization, the device driver code itself needs to be modified manually to incorporate settings other than the default.

In the example source file *dsk5416_aic20evm.c*, locate the following portion of code that is part of the `DSK5416_AIC20EVM_setup()` function:

```
Void DSK5416_AIC20EVM_setup()
{
   // ** TODO: Configure the unique configuration parameters for each channel **
   DSK5416_AIC20EVM_chanConfigParams[MST_CHAN1].reg[CR6A] = LINEI; // master ch1 ADC
   DSK5416_AIC20EVM_chanConfigParams[MST_CHAN2].reg[CR6A] = MICI;  // master ch2 ADC
   DSK5416_AIC20EVM_chanConfigParams[SLV_CHAN1].reg[CR6A] = HNSI;  // slave  ch1 ADC
   DSK5416_AIC20EVM_chanConfigParams[SLV_CHAN2].reg[CR6A] = HDSI;  // slave  ch2 ADC

   DSK5416_AIC20EVM_chanConfigParams[MST_CHAN1].reg[CR6B] = LINEO; // master ch1 DAC
   DSK5416_AIC20EVM_chanConfigParams[MST_CHAN2].reg[CR6B] = SPKO; // master ch2 DAC
   DSK5416_AIC20EVM_chanConfigParams[SLV_CHAN1].reg[CR6B] = SPKO; // slave ch1 DAC
   DSK5416_AIC20EVM_chanConfigParams[SLV_CHAN2].reg[CR6B] = HDSO;  // slave  ch2 DAC
}
```

The above code is setting each channel's ADC and DAC lines for a specific configuration. Here is where any other control register modifications can be added and set for each channel. The above code which is supplied *out of the box* with the associated sample code results in the following voice device I/O configuration on the AIC20EVM board:

**Table 3. DSK5416_AIC20EVM Default I/O Codec Channel Settings**

| AIC20EVM CASCADE CHANNEL | EVM INPUT CONNECTION | CODEC INPUT LINE | EVM OUTPUT CONNECTION | CODEC OUTPUT LINE |
|---|---|---|---|---|
| Master channel #1 | J14 (+/-) | Line input (LINEI) | J5 (+/-) | Line output (LINEO) [600 Ω] |
| Master channel #2 | J16 (3.5 mm input jack) | MIC input (MICI) | J2 (+/-) | Speaker output (SPKO) [8 Ω] |
| Slave channel #1 | J19 (+/-) | Hand set input (HNSI) | J11 (+/-) | Speaker output (SPKO) [8 Ω] |
| Slave channel #2 | J13 (+/-) | Head set input (HDSI) | J3 (+/-) | Head set output (HDSO) [150 Ω] |

**CAUTION:**

**Disconnecting and reconnecting the sound sources from the codec input lines during normal operation could result in unwanted noise spikes input to the channels and cause the DSK5416_AIC20EVM device driver to stop working altogether.**

The device header file <aic20.h> contains the configuration structure definition that is used for each channel. A set of default control registers is set in this header file and can be changed freely by the application developer so that a known default set of control registers is programmed for all AIC20 cascaded channels during the `DSK5416_AIC20EVM_open()` function call. The default sampling rate as specified in the <aic20.h> file that is packaged with the sample application code is 8 kHz for each channel. Making changes to the sampling frequency may involve reconfiguring the DSP CLKOUT which is based on the DSP clock speed.

**WARNING:**

**Do not change the bit fields that determine the different register contents for a Control Register with the same number (e.g. control register #5 has four sub-registers: CRs # 5A, 5B, 5C, 5D). Typically, the 1 or 2 most significant bits of the control word determine which sub-register of the overall control register gets programmed. Refer to the <aic20.h> file comments that identify these bit fields.**

| ARRAY INDEX | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| | MASTER CHAN 1 CONTROL REGISTERS | MASTER CHAN 2 CONTROL REGISTERS | SLAVE CHAN 1 CONTROL REGISTERS | SLAVE CHAN 2 CONTROL REGISTERS |
| reg[0] | CR #01 | CR #01 | CR #01 | CR #01 |
| reg[1] | CR #02 | CR #02 | CR #02 | CR #02 |
| reg[2] | CR #3A | CR #3A | CR #3A | CR #3A |
| reg[3] | CR #3B | CR #3B | CR #3B | CR #3B |
| reg[4] | CR #3C | CR #3C | CR #3C | CR #3C |
| reg[5] | CR #3D | CR #3D | CR #3D | CR #3D |
| reg[6] | CR #4A | CR #4A | CR #4A | CR #4A |
| reg[7] | CR #4B | CR #4B | CR #4B | CR #4B |
| reg[8] | CR #5A | CR #5A | CR #5A | CR #5A |
| reg[9] | CR #5B | CR #5B | CR #5B | CR #5B |
| reg[10] | CR #5C | CR #5C | CR #5C | CR #5C |
| reg[11] | CR #5D | CR #5D | CR #5D | CR #5D |
| reg[12] | CR #6A | CR #6A | CR #6A | CR #6A |
| reg[13] | CR #6B | CR #6B | CR #6B | CR #6B |

**Figure 16. Cascade Channel Configuration (Global Shadow Registers)**

## 8.3 Implementation of the Device Driver

### *8.3.1 Design Decisions and Core Code*

#### 8.3.1.1 DSP Peripherals and Initialization Sequence

Before the DSP can communicate with the AIC20 devices, its McBSP must be configured during the system initialization phase. TI's chip support library (CSL) tools and APIs are used to easily configure the McBSP #1 so that it can properly receive and transmit data from/to the device.

The chip support library can be used in two ways – via a DSP/BIOS configuration file (*.CDB), or by programmatically invoking APIs. The *DSK5416_AIC20EVM* device driver uses the CSL in both ways. To configure the McBSP, the *audioapp.cdb* file is used to store information on how the serial port should be configured at startup. To see the settings that are required for proper receive and transmit operation of the McBSP, double-click on the *audioapp.cdb* file that is part of the CCS project in the Project View window. Expand the *Chip Support Library* category and then expand the *MCBSP Multichannel Buffered Serial Port* and then the *MCBSP Channel Configuration* categories. Right-click on the *mcbspCfg0* icon and select *Properties*.

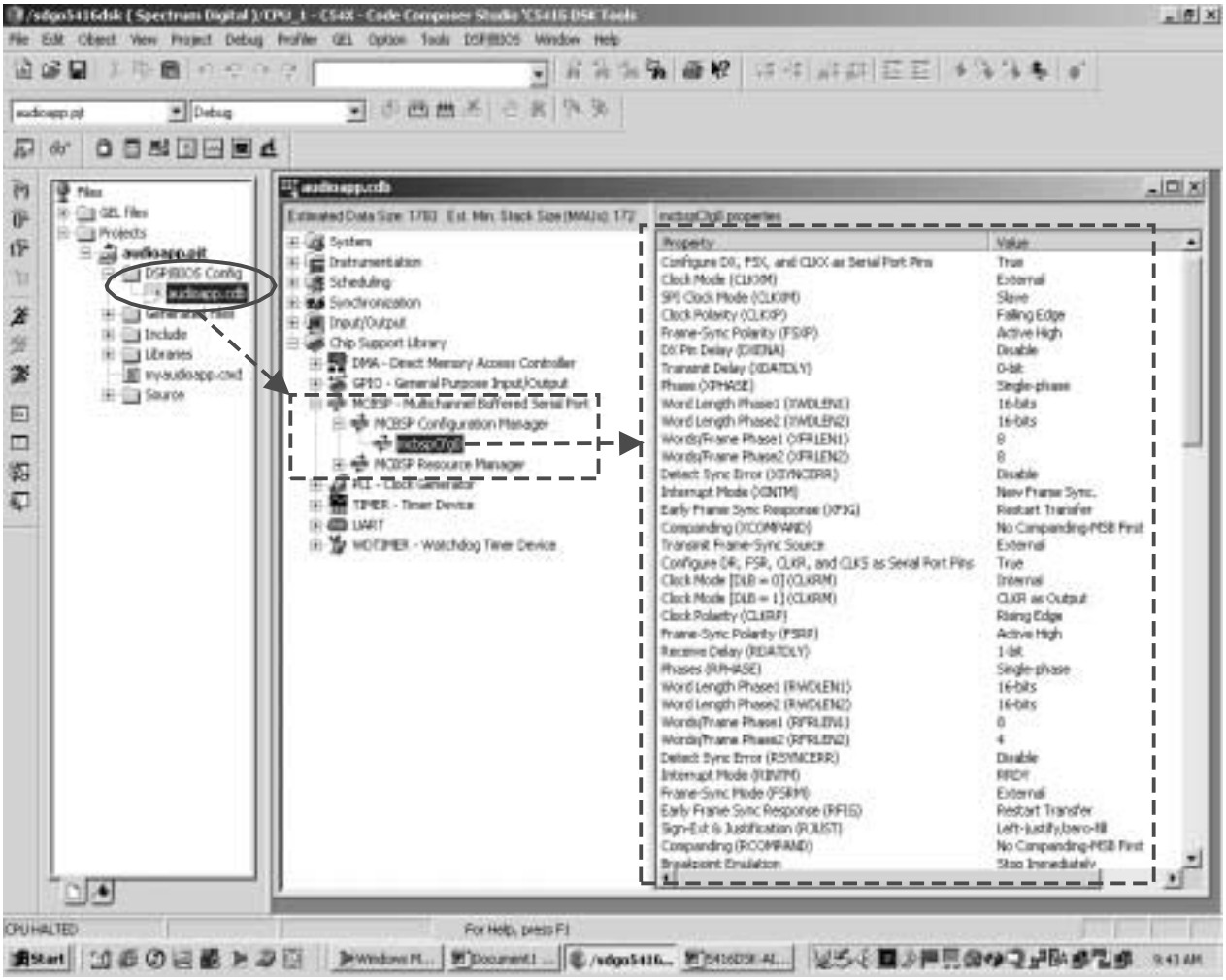


**Figure 17. DSP Peripheral Configuration Using CSL**

The DSP/BIOS system takes care of initializing the McBSP based on these settings. Once the McBSP #1 is configured, only a single call to a CSL API needs to be performed. Once the McBSP has started, a series of control words, based on the contents of the global channel configuration array, is sent out in a single stream all at once. `DSK5416_AIC20EVM_open()` must be called after the one-time call to `DSK5416_AIC20EVM_init()`.

```
DSK5416_AIC20EVM_DeviceHandle DSK5416_AIC20EVM_open()
{
   IRQ_enable(IRQ_EVT_XINT1);

   MCBSP_start(C54XX_MCBSP_hMcbsp, MCBSP_XMIT_START | MCBSP_RCV_START |
                                   MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220);

   // Send out all the Control Register data for all channels
   DSK5416_AIC20EVM_programAllRegs();

   IRQ_enable(IRQ_EVT_RINT1);
}
```

### 8.3.1.2 Configuring DSP Speed, DSP CLKOUT, and AIC20 Sampling Frequency

A master clock (MCLK) signal must be provided to drive each AIC20 device. All of the AIC20's operations and timings are driven off the incoming MCLK signal. In turn, each AIC20 generates a serial clock (SCLK), which is then fed back to the McBSP to drive the read/write bit timings.

The TMS320VC5416 ™ DSP is capable of operating at a maximum speed of 160 MHz. Based on the DSP speed, a CLKOUT can be generated to drive an external device such as the AIC20. In essence, the CLKOUT serves as the MCLK for the AIC20 cascade. The VC5416 allows the CLKOUT to be derived from the DSP speed divided by a factor of 1, 2, 3, or 4.

On the C5416DSK board, a 16-MHz oscillator feeds the CLKIN to the DSP. The DSP PLL multiplier value (PLLMUL) allows the DSP speed to be set as a multiple of the CLKIN, up to 160 MHz. For this reference platform, 144 MHz was chosen for the DSP speed. Why was 160 MHz not chosen – the maximum speed allowable for the TMS320VC5416?

According to the AIC20 data manual, the sampling frequency is set by the following formula:

$F_s$ = *[ MCLK / (16 x M x N x P ) ]*

*{10 MHz <= (MCLK / P) <= 25 MHz}, {1 <= M <= 128}, {1 <= N <= 16}, {1 <= P <= 8}*

By inspection, we see that the MCLK (DSP CLKOUT) value, as well as the restrictions on the values of M, N, P, determine the attainable sampling frequency. In order to achieve exactly 8-kHz sampling rate and get closest to the maximum DSP speed, 144 MHz was chosen because:

*CLKOUT = DSP Speed / PLLDIV = 144 MHz / [1, 2, 3, 4] = 144 MHz / 3 = 48 MHz*

Using a CLKOUT of 48 MHz, it is possible to achieve exactly 8 kHz, 12 kHz, and 24 kHz sampling rates with the DSP running at 144 MHz. For example, to get exactly 8 kHz, we can choose M=15, N=5, P=5 so that *[48 MHz / (16 x 15 x 5 x 5)] = 8 kHz*. It is possible to achieve 16-kHz sampling frequency, but the DSP speed would only be 128 MHz out of a possible 160 MHz.

**Table 4. DSK5416_AIC20EVM Sampling Frequency Settings**

| SAMPLING FREQUENCY | M | N | P | DSP PLLMUL | DSP PLLDIV | DSP SPEED | DSP CLKOUT (AIC20 MCLK) |
|---|---|---|---|---|---|---|---|
| 8 kHz | 15 | 5 | 5 | 8 (+ 1) | 3 | 144 MHz | 48 MHz |
| 12 kHz | 10 | 5 | 5 | 8 (+ 1) | 3 | 144 MHz | 48 MHz |
| 16 kHz | 10 | 5 | 5 | 7 (+ 1) | 2 | 128 MHz | 64 MHz |
| 24 kHz | 5 | 5 | 5 | 8 (+ 1) | 3 | 144 MHz | 48 MHz |

Note: Achieving 16-kHz sampling rate requires DSP speed and CLKOUT to be reconfigured from the default values.

#### 8.3.1.3 Interrupt Service Routines

*ISR Initialization Code*

The interrupt service routines (ISRs) are initialized using TI's standard *Chip Support Library (CSL)* APIs. These easy-to-use API's allow the device driver's `DSK5416_AIC20EVM_init()` function to dynamically plug the ISR into the vector table as well as enable global interrupts.

```
DSK5416_AIC20EVM_DeviceHandle DSK5416_AIC20EVM_init()
{
   Uint16   index;

   // Set up the SWWSR, BSCR, SWCR registers
   EBUS_config(&DSK5416_AIC20EVM_myMemConfig);

   // Check for Motherboard connection and force reset if it's there
   if (DSK5416_DC_REG & DSK5416_DC_DETECT) {
      DSK5416_DC_REG &= DSK5416_DC_NO_RST;
      DSK5416_DC_REG |= DSK5416_DC_RESET;
      for (index = 0; index < EB_RESET_DELAY; index++)
         DSK5416_AIC20EVM_delay(EB_RESET_DELAY);
      DSK5416_DC_REG &= DSK5416_DC_NO_RST;
   }
   else
      return (FALSE);

   // Clear any pending interrupts (IFR)
   IRQ_clear(IRQ_EVT_RINT1);
   IRQ_clear(IRQ_EVT_XINT1);

   // Place the HWI hooks at the proper spots in the interrupt vector table
   // NOTE: only use IRQ_plug() when NOT using the DSP/BIOS HWI Dispatcher!!!
   IRQ_plug(IRQ_EVT_RINT1, &DSK5416_AIC20EVM_rcvXmtSample);
   IRQ_plug(IRQ_EVT_XINT1, &DSK5416_AIC20EVM_frameSync);

   // Enable interrupts globally (INTM)
   IRQ_globalEnable();

   // Device initialization successful
   return (TRUE);
}
```

The McBSP transmitter is initialized to generate an interrupt on every new FS detected. The McBSP receiver will be initialized to generate an interrupt on every *RRDY Event*, which means each time a new data word has been received at the McBSP and shifted into the data receive register (DRR). The CPU can directly access the DRR without much of a performance hit, since it is a memory mapped register (MMR) that resides in DSP internal data memory. The details of what happens during every McBSP Tx and Rx interrupt are discussed in the following section.

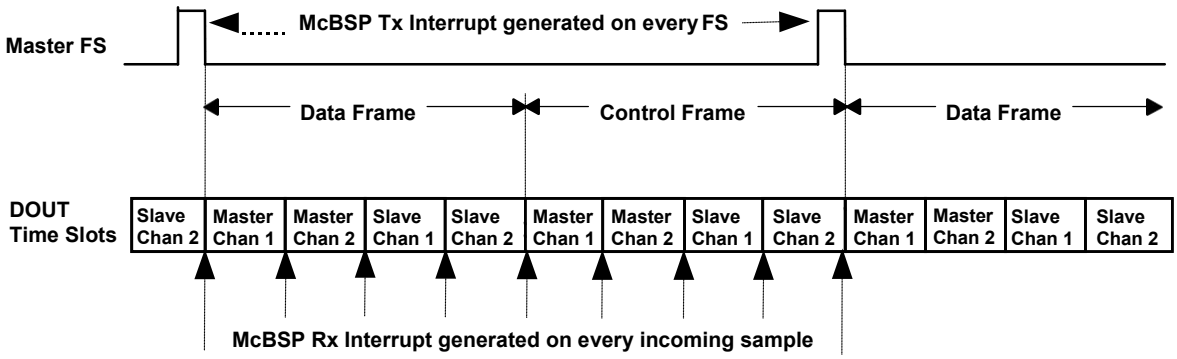


**Figure 18. McBSP Interrupt Service Routines Configuration**

*McBSP Transmit ISR (Tx Event = FSX Detected)*

The FS signal from the master AIC20 device is connected to the FSX and FSR inputs of the McBSP. This configuration allows the DSP to detect the FS at the McBSP, generate an interrupt, and have the interrupt serviced. The function `DSK5416_AIC20EVM_frameSync()` is implemented to increase a global frame sync counter, as well as tell the DSP that the current data word coming into the McBSP DRR corresponds to the first timing slot of the FS period. A global index array is used to point to the current time slot. Each time the FS interrupt occurs, this index is simply set to 0 which serves as the pointer to the first array element of the receive buffer. The McBSP Rx ISR relies on the Tx ISR to reset the buffer index each time a new FS signal is detected at the McBSP FSX input.

```
interrupt void DSK5416_AIC20EVM_frameSync()
// Called when FSX detected
{
   // Update counter to signal that another FS has just been detected
   DSK5416_AIC20EVM_gFsCounter++;
   // Reset timing slot pointer
   DSK5416_AIC20EVM_gBufferIndex = 0;
}
```

*McBSP Receive ISR (Rx Event = RRDY Detected)*

The McBSP's receive mode can be configured to generate a special interrupt each time a new data word has been read at the McBSP. Therefore, the logical function for the associated ISR would be to just read in the current contents of the DRR. The global buffer index is always pointing to the current time slot which just corresponds to a specific position in the receive buffer array. Once the data has been read and written to the receive buffer, the buffer index is incremented for the next data word to be read at the McBSP. When the index reaches the frame buffer size, that signals that the buffer is full and needs to be processed. A DSP/BIOS software interrupt (SWI) is posted which invokes the `processBuffer()` function where the data can be consumed by the application framework.

In a DSP/BIOS-based scheduling system, the highest priority SWI runs, but can be preempted by all hardware interrupts (HWIs). In this example, the hardware interrupts are used to read and write the actual data values while the SWIs are used to perform the less urgent (but still real-time critical) DSP processing functions on the filled receive buffers. The McBSP receive ISR keeps track of how many words have been written into the current receive ping-pong buffer, and when the buffer is full (i.e. FS period completed), the SWI processing function is posted and runs in the context of the DSP/BIOS scheduler when no HWIs are being serviced.

Since both receive and transmit modes are driven by the same serial clock (SCLK) of the AIC20 cascade, it would make sense for the device driver to transmit an output sample for every input sample that is received. So, the McBSP receive ISR immediately writes out a sample from the current Tx buffer right after a new sample has been read into the current Rx buffer. However, since the McBSP transmit mode is double-buffered, whatever data word is written to the data Xmit register (DXR) appears on the data bus exactly 2 time slots in the future. So, the ISR must *look ahead* two channels and get that channel's data to write out during each current read cycle (triggered by an RRDY event).

For example, if the current received word is master channel 1's data, then the Tx data for slave channel 1 must be written to the McBSP immediately after the read (buffer index 1➔2➔[3]). If the current Rx timing slot is slave channel 1's CR contents, then the Tx data for master channel 1 (buffer index 7➔0➔[1]) must be immediately written to the McBSP to assure it falls within the correct timing slot 2 cycles in the future, due to the double-buffered nature of the McBSP transmitter. See the following Table for the *lookahead* decision-making process on which Tx buffer sample must be sent out depending on the current Rx buffer index. In summary, every read cycle (i.e. every RRDY receive event) must include one read and one write operation by the host processor to keep the TDM DIN and DOUT data streams continuous. The `DSK5416_AIC20EVM_rcvXmtSample()` function is plugged into the vector table as the ISR to run for every McBSP RRDY event.

```
interrupt void DSK5416_AIC20EVM_rcvXmtSample() // Called for every McBSP RRDY Receive event
{
   static short DSK5416_AIC20EVM_pingOrPong = PING;

   if (DSK5416_AIC20EVM_pingOrPong == PING) {
      // Read the current DOUT word
      DSK5416_AIC20EVM_gBufferRcvPing[gBufferIndex] = MCBSP_read16(C54XX_MCBSP_hMcbsp);
      // Write out the DIN word for the corresponding future timing slot
      MCBSP_write16(DSK5416_AIC20EVM_gBufferXmtPing[(gBufferIndex+RXTXOFFSET)%(BUFFSIZE)]);
      // Increment timing slot pointer for next read
      DSK5416_AIC20EVM_gBufferIndex++;
      // Post SWI if frame buffer full
      if (DSK5416_AIC20EVM_gBufferIndex == DSK5416_AIC20EVM_BUFFSIZE) {
         SWI_or(&processBufferSwi, PING);
         DSK5416_AIC20EVM_pingOrPong = PONG;
      }
   }
   else { // DSK5416_AIC20EVM_pingOrPong == PONG
      <repeat above code exactly but for the PONG Rx & Tx buffers>
   }
}
```

**Table 5. DSK5416_AIC20EVM McBSP Write Decision per Receive Interrupt**

| BUFFER INDEX (RX) | CURRENT READ (DOUT) TIME SLOT BASED ON CURRENT RX BUFFER INDEX VALUE | TX BUFFER ARRAY LOCATION | (DIN) TIME SLOT DATA TO IMMEDIATELY WRITE TO DXR AFTER CURRENT READ (DOUT) FROM DRR |
|---|---|---|---|
| 0 | SLAVE CHANNEL #2 CONTROL | 2 | MASTER CHANNEL #2 DATA |
| 1 | MASTER CHANNEL #1 DATA | 3 | SLAVE CHANNEL #1 DATA |
| 2 | MASTER CHANNEL #2 DATA | 4 | SLAVE CHANNEL #2 DATA |
| 3 | SLAVE CHANNEL #1 DATA | 5 | MASTER CHANNEL #1 CONTROL |
| 4 | SLAVE CHANNEL #2 DATA | 6 | MASTER CHANNEL #2 CONTROL |
| 5 | MASTER CHANNEL #1 CONTROL | 7 | SLAVE CHANNEL #1 CONTROL |
| 6 | MASTER CHANNEL #2 CONTROL | 0 | SLAVE CHANNEL #2 CONTROL |
| 7 | SLAVE CHANNEL #1 CONTROL | 1 | MASTER CHANNEL #1 DATA |

### *8.3.2 Coding Conventions, File Structure, and Packaging*

The *DSK5416_AIC20EVM* device driver code follows the standard coding conventions used in all eXpressDSP™ components (XDAIS, DSP/BIOS, RF) to allow for ease of integration and easy readability. All global symbols are prefixed with the <BOARD>_<DEVICE>_ API designation (e.g. DSK5416_AIC20EVM_) prefix to maintain uniqueness of symbol names so that the code can co-exist with all other system code and avoid symbol clashes. The interface itself also follows a uniform naming convention so that it is always obvious if a label is a constant, data type, function name, field within a structure, a function parameter, etc.

**Table 6. DSK5416_AIC20EVM Example Naming Conventions**

| Label Type | Convention | Example |
|---|---|---|
| SYMBOLIC CONSTANTS | All UPPERcase, single word (no underscores) after prefix | DSK5416_AIC20EVM_**NUMCHANS** |
| All data types | Titlecase (no underscores) after prefix | DSK5416_AIC20EVM_**B**ufferIndex |
| Structure fields | Begins with lowercase after prefix (no underscores after the prefix) | DSK5416_AIC20EVM_chanConfigParams[ ].**r**eg[ ] |
| function parameters | | DSK5416_AIC20EVM_writeControlWords(Uint16 **m**aster, Uint16 **s**lave2, Uint16 **s**lave1, Uint16 **s**lave0) |
| variables | | DSK5416_AIC20EVM_**h**Device |
| function names | Single-word (no underscores), begins with lowercase after prefix | DSK5416_AIC20EVM_**i**nit() |

Note: Never use the underscore ('_') character to separate words after the <BOARD>_<DEVICE>_ prefix; use Titlecase words instead to separate multiple worded names (e.g. *DSK5416_AIC20EVM_read**C**ontrol**D**ata**W**ords*)

The *DSK5416_AIC20EVM* device driver files also follow the standard eXpressDSP™ device driver packaging and delivery conventions. The main interface to the device driver is aptly named *dsk5416_aic20evm.h* (following the <board>_<device>.h naming convention), and only the relevant data types and APIs are exposed to the outside world. The following is a summary of the files that make up the *DSK5416_AIC20EVM* device driver:

- <aic20.h> – contains control register configuration data types and default settings
- <dsk5416_aic20evm.h> – contains all relevant constants, data types, and APIs used to interact with the device driver

- *dsk5416_aic20evm.c* – contains all device-specific code for basic AIC20EVM operations
- *dsk5416_aic20evm_ctrl.c* – contains (optional) run-time control code to change channel operating parameters on the fly

Typically, eXpressDSP™ device drivers are packaged as a single library file which follows the <board>_<device>.l<dspcore> naming convention. The library file can then be linked into the application just as if it were any other foundation library used for building the project. For this application note example, the device driver object code is packaged in a single file named *dsk5416_aic20evm.l54f*, where the *f* stands for far calls and returns on the C54x-based object code. For an application that uses the near call/return memory model, the device driver source code could be rebuilt with the appropriate options and named *dsk5416_aic20evm.l54* (no *f* in the suffix) to designate that the library is to be used in a near memory model system only.

**NOTE**: The *l* in the *.l54f file name suffix is a lower-case letter *L*. This is how TI distinguishes file names that are library archives.

## 8.4 Changing Device and Channel Parameters During Run-Time

The DSK5416_AIC20EVM device driver code *out of the box*, in its currently released form, configures the AIC20 devices for programming mode (vs continuous mode). Programming mode means that for every FS period, there is a specific time slot to either send a command to read/write a specific control register of a specific channel. For example, if there are four audio channels in the cascade, then each FS period consists of eight timing slots (first four timing slots are for reading/writing the actual data sample, while the remaining four timing slots read or write to a single control register for the time slot's channel).

To send either a read or write control register command to a specific channel, the framework needs to write the command in the appropriate location in the Xmt ping-pong buffer. If no command is to be sent out, then the value of the command should be set to zero since it is not desired to write a random value to the control register timing slot and inadvertently change it.

### *8.4.1 Run-Time Control Functions*

The sample source file *dsk5416_aic20evm_ctrl.c* contains a group of ready-to-use wrapper functions that modify control register contents in the channel configuration array using convenient-to-use high-level APIs called by the framework. The channel configuration array can be treated as *shadow registers* of the actual AIC20 device registers. The framework only needs to call the functions on the correct channels by writing to the appropriate control register timing slot(s) in the Xmt ping-pong buffer.

```
DSK5416_AIC20EVM_gBufferXmtPing[MST_CHAN1_C] =
   DSK5416_AIC20EVM_enableFIR(hDevice, MST_CHAN1);
DSK5416_AIC20EVM_gBufferXmtPing[MST_CHAN2_C] =
   DSK5416_AIC20EVM_muteHandset(hDevice, MST_CHAN2, DSK5416_AIC20EVM_ENABLE);
DSK5416_AIC20EVM_gBufferXmtPing[SLV_CHAN1_C] =
   DSK5416_AIC20EVM_setSpeakerGain(hDevice, SLV_CHAN1, SPKG_DB02);
DSK5416_AIC20EVM_gBufferXmtPing[SLV_CHAN2_C] =
   DSK5416_AIC20EVM_setSidetoneGains(hDevice, SLV_CHAN2, SIDETONEMUTE, STG_NDB27);
```

In the above example, by just writing to the appropriate control register timing slots into the Xmt buffer of the device driver, the following configuration parameters are changed within the same FS period:

- Master channel #1 FIR filter is ENABLED (IIR filter is disabled automatically)
- Master channel #2 handset output is MUTED
- Slave channel #1 speaker gain is set to +2 dB
- Slave channel #2 sidetone gains are set (Analog is MUTED & Digital = –27dB)

Whenever no control register changes are needed, it is strongly recommended to write 0's for the control register timing slots to avoid inadvertently writing out random control data.

```
DSK5416_AIC20EVM_gBufferXmtPong[MST_CHAN1_C] = 0; // Master Chan 1 ctrl
DSK5416_AIC20EVM_gBufferXmtPong[MST_CHAN2_C] = 0; // Master Chan 2 ctrl
DSK5416_AIC20EVM_gBufferXmtPong[SLV_CHAN1_C] = 0; // Slave Chan 1 ctrl
DSK5416_AIC20EVM_gBufferXmtPong[SLV_CHAN2_C] = 0; // Slave Chan 2 ctrl
```

The following is a complete list of the run-time AIC20 control functions supplied with the existing *DSK5416_AIC20EVM* driver module (also found in the header file <dsk5416_aic20evm.h>:

Control Register #1

```
Uint16 DSK5416_AIC20EVM_enableFIR(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                  DSK5416_AIC20EVM_Channel  chan);
Uint16 DSK5416_AIC20EVM_enableIIR(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                  DSK5416_AIC20EVM_Channel chan);
Uint16 DSK5416_AIC20EVM_setAnalogLoopback(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                          DSK5416_AIC20EVM_Channel chan,
                                          DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setDigitalLoopback(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                           DSK5416_AIC20EVM_Channel chan,
                                           DSK5416_AIC20EVM_Cmd cmd);
```

Control Register #2

```
Uint16 DSK5416_AIC20EVM_setTurboMode(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel chan,
                                     DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setDIFbypass(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel chan,
                                     DSK5416_AIC20EVM_Cmd cmd);
```

Control Register #3A

```
Uint16 DSK5416_AIC20EVM_powerdownADC(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel  chan);
Uint16 DSK5416_AIC20EVM_powerdownDAC(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel chan);
Uint16 DSK5416_AIC20EVM_powerdownALL(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel chan);
Uint16 DSK5416_AIC20EVM_reset(DSK5416_AIC20EVM_DeviceHandle hDevice,
                              DSK5416_AIC20EVM_Channel chan);
```

Control Register #3B

```
Uint16 DSK5416_AIC20EVM_set8KBPF(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                 DSK5416_AIC20EVM_Channel chan,
                                 DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_muteHandset(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                    DSK5416_AIC20EVM_Channel chan,
                                    DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_muteHeadset(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                    DSK5416_AIC20EVM_Channel chan,
                                    DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_muteLineOutput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                       DSK5416_AIC20EVM_Channel chan,
                                       DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_muteSpeaker(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                    DSK5416_AIC20EVM_Cmd cmd,
                                    DSK5416_AIC20EVM_Channel chan);
```

**NOTE**: Control registers #4A and #4B are used to set the M, N, P values for configuring the sampling frequency. It is not advisable to reconfigure the sampling frequencies during run-time, especially since those values may depend on reconfiguring the DSP speed and CLKOUT frequencies as well. Hence, no run-time control functions are supplied for these registers.

Control Register #5A

```
typedef enum DSK5416_AIC20EVM_A2DGain { /* e.g. DB07_5 = +7.5 dB A/D Gain */
  A2DMUTE, DB54_0, DB48_0, DB42_0, DB40_5, DB39_0, DB37_5, DB36_0, DB34_5, DB33_0,
  DB31_5, DB30_0, DB28_5, DB27_0, DB25_5, DB24_0, DB22_5, DB21_0, DB19_5, DB18_0,
  DB16_5, DB15_0, DB13_5, DB12_0, DB10_5, DB09_0, DB07_5, DB06_0, DB04_5, DB03_0,
  DB01_5, DB00_0
} DSK5416_AIC20EVM_A2DGain;
```

```
Uint16 DSK5416_AIC20EVM_setADPGA(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                 DSK5416_AIC20EVM_Channel chan,
                                 DSK5416_AIC20EVM_A2DGain gain);
```

Control Register #5B

```
typedef enum DSK5416_AIC20EVM_D2AGain { /* e.g. NDB07_5 = -7.5 dB D/A Gain */
  D2AMUTE, NDB54_0, NDB48_0, NDB42_0, NDB40_5, NDB39_0, NDB37_5, NDB36_0, NDB34_5,
NDB33_0, NDB31_5, NDB30_0, NDB28_5, NDB27_0, NDB25_5, NDB24_0, NDB22_5, NDB21_0,
NDB19_5, NDB18_0, NDB16_5, NDB15_0, NDB13_5, NDB12_0, NDB10_5, NDB09_0, NDB07_5,
NDB06_0, NDB04_5, NDB03_0, NDB01_5, NDB00_0
} DSK5416_AIC20EVM_D2AGain;
```

```
Uint16 DSK5416_AIC20EVM_setDAPGA(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                 DSK5416_AIC20EVM_Channel chan,
                                 DSK5416_AIC20EVM_D2AGain gain);
```

### Control Register #5C

```
typedef enum DSK5416_AIC20EVM_SidetoneGain { /* e.g. STG_NDB24 = -24 dB s/t gain */
  SIDETONEMUTE, STG_NDB27, STG_NDB24, STG_NDB21, STG_NDB18, STG_NDB15, STG_NDB12,
STG_NDB09
} DSK5416_AIC20EVM_SidetoneGain;

Uint16 DSK5416_AIC20EVM_setSidetoneGains(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                         DSK5416_AIC20EVM_Channel chan,
                                         DSK5416_AIC20EVM_SidetoneGain gainAnalog,
                                         DSK5416_AIC20EVM_SidetoneGain gainDigital);
```

### Control Register #5D

```
typedef enum DSK5416_AIC20EVM_SpeakerGain { /* e.g. SPKG_DB01 = +1 dB speaker gain */
  SPKG_DB00, SPKG_DB01, SPKG_DB02, SPKG_DB03
} DSK5416_AIC20EVM_SpeakerGain;

Uint16 DSK5416_AIC20EVM_setSpeakerGain(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                       DSK5416_AIC20EVM_Channel chan,
                                       DSK5416_AIC20EVM_SpeakerGain gain);
```

### Control Register #6A

```
Uint16 DSK5416_AIC20EVM_setHeadsetIO(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel chan,
                                     DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setHandsetIO(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel chan,
                                     DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setCallerID(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                    DSK5416_AIC20EVM_Channel chan,
                                    DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setLineInput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                     DSK5416_AIC20EVM_Channel chan,
                                     DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setMicInput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                    DSK5416_AIC20EVM_Channel chan,
                                    DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setHandsetInput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                        DSK5416_AIC20EVM_Channel chan,
                                        DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setHeadsetInput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                        DSK5416_AIC20EVM_Channel chan,
                                        DSK5416_AIC20EVM_Cmd cmd);
```

Control Register #6B

```
Uint16 DSK5416_AIC20EVM_setAnalogSidetoneHeadset(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                                 DSK5416_AIC20EVM_Channel chan,
                                                 DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setAnalogSidetoneHandset(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                                 DSK5416_AIC20EVM_Channel chan,
                                                 DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setSpeakerOutput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                         DSK5416_AIC20EVM_Channel chan,
                                         DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setLineOutput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                      DSK5416_AIC20EVM_Channel chan,
                                      DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setHandsetOutput(DSK5416_AIC20EVM_DeviceHandle hDevice,
                                         DSK5416_AIC20EVM_Channel chan,
                                         DSK5416_AIC20EVM_Cmd cmd);
Uint16 DSK5416_AIC20EVM_setHeadsetOutput(DSK5416_AIC20EVM_DeviceHandle,
                                         DSK5416_AIC20EVM_Channel chan,
                                         DSK5416_AIC20EVM_Cmd cmd);
```

### *8.4.2 Run-Time Control Thread*

A program on the host PC could be created to control each AIC20 channel during run-time of the system. This program could simply send a high-level command to the target via RTDX™ and then the framework would just need to write the command to the appropriate channel's control register time slot. The device driver sends out the control register command in the correct time slot like it does with the normal audio sample data that is transmitted to the DIN line of the AIC20 device, as shown in the above code samples.

In this reference framework example, a c*ontrol thread* is provided as a simple example of how the host program can send commands to the target to change its configuration such as algorithm and device parameters. The control thread is a periodic ISR which is run every 20 timer ticks in its current DSP/BIOS configuration. So if each timer tick is 1 ms (i.e. the default setting for DSP/BIOS), then the control thread is scheduled to run every 20 ms. The ISR accesses a global array of data words which represents something like a device I/O area. The array contains a specific location where if the value is nonzero, then that serves as a flag for the control thread to take the appropriate action based on the other values in the device I/O area.

The device I/O area array is set up to contain the gain percentage for each of the VOL_TI algorithm instances that belong to each data channel. So if the control thread sees that the host has just modified the device I/O area (by checking if the first word in the device I/O area array is non-zero), then the control thread ISR simply reads each new gain percentage value and changes the corresponding channel's VOL_TI algorithm object accordingly.

```
Int deviceControlsIOArea[] = {
    FALSE,   /* initially, no user action */
    0,       /* available for future use */
    100,     /* default volume gain % for channel #1 */
    100,     /* default volume gain % for channel #2 */
    100,     /* default volume gain % for channel #3 */
    100,     /* default volume gain % for channel #4 */
};
```

The purpose of the control thread is to detect hardware events such as a user pressing on the device's buttons and other controls, and applying them to data processing. (For example, changing the volume, modifying filter parameters, canceling a channel etc.). The mechanism is the following: a hardware event such as a button press triggers an interrupt, serviced by the following `thrControlIsr()` function.

```
Void thrControlIsr( Void ) {
    Int i;
    static Uns activationCount = 0;

    /*  We are really called from the CLK object upon every timer interrupt
     *  whereas user's action would occur after relatively long intervals,
     *  so we try to simulate that, too. Since interrupts occur every
     *  1ms (so is the CDB configured), we arbitrarily decide to actually
     *  do anything in this procedure every 20 interrupts, i.e. every 20 ms.
     *  If the if() clause below is removed, then the response to the user's
     *  action could happen in one millisecond. That would be the case with
     *  control ISR activated by a separate interrupt line. Such ISR would
     *  not need the if() clause below.
     */
    if (++activationCount < 20)   /* 20 ms */
        return;
    else
        activationCount = 0;

    /* now proceed with regular "I/O memory" reading actions */

    /* check if there has been any unread user input */
    if (deviceControlsIOArea[0] == FALSE)
        return;     /* there has not; return */

    /*  Read "volume" value for all channels and store
     *  the information in control thread's data structure. Interpretation
     *  is trivial in this case, we just copy host's data to control thread's
     *  data structure.
     */
    for (i = 0; i < DSK5416_AIC20EVM_NUMCHANS; i++)
        /* read "slider" for volume */
        thrControl.outputVolume[i] = deviceControlsIOArea[ 2 + i ];

    /* now post the control thread */
    SWI_post( &swiControl );

    /* and clear the "user input" flag */
    deviceControlsIOArea[0] = FALSE;
}
```

This procedure quickly reads the hardware parameters, interprets them, stores the result of the interpretation *in thrControl* thread's data structure, posts the *swiControl* thread, and exits. It does not modify any of the data processing parameters itself.

The *swiControl* thread is a SWI like the processing SWIs, and has exactly the same priority; it executes thread function `thrControlRun()`. Function `thrControlRun()`, based on the data presented by `thrControlIsr()`, modifies processing thread's data and/or XDAIS algorithms' parameters. The reason the encapsulate processing parameters modification logic is in a separate SWI thread is twofold:

- To keep the time spent in a hardware interrupt at a minimum;
- `thrControlIsr()` most likely interrupted a data processing thread; Do not modify processing parameters in the middle of processing activity.

By having *swiControl* have the same priority as the processing threads, we ensure that it does not prevent them, nor gets starved by them, so it has a low latency. The *swiControl* thread has the priority of the lowest-priority thread whose data parameters it modifies.

An alternative approach may be to check the device controls every certain period (for example, 20 ms), if there is no interrupt that would inform us about the hardware event. This choice depends on the application and the device.

In our example, the host writes into an area of memory that simulates I/O area where the buttons and other controls are located. There is a *clkControl* CLK object that runs the `thrControlIsr()` function; that is a timer interrupt which simulates a device control interrupt. This is used when developing and testing applications on a board, such as DSK5416, where no buttons are connected to any interrupt lines. Each channel's volume gain can be changed by accessing the GEL sliders found in the GEL column of the CCS Toolbar (GEL ➔ Application Control ➔ Set_channel_<x>_volume). More sliders can be added by modifying the provided *app.gel* file.

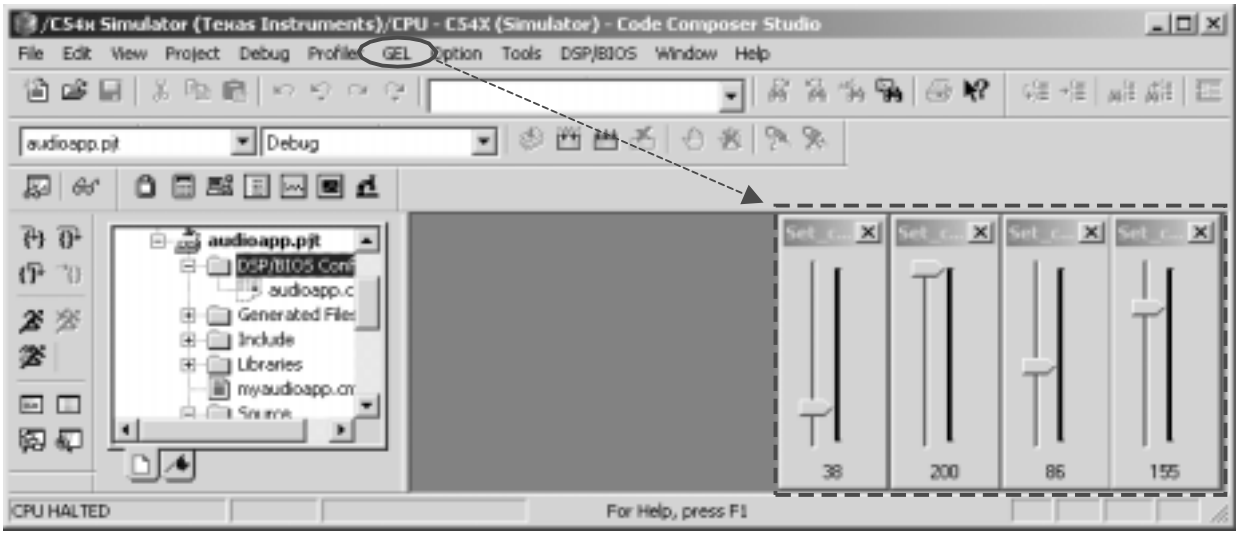


**Figure 19. Host CCS GEL Sliders for Changing Channel Volumes**

This reference framework example currently only allows the VOL_TI algorithm's gain percentage to be changed permanently during runtime, however, it should be easy for the developer to build additional functionality into the control thread to perform device-specific changes such as mute certain outputs of a channel, enable/disable the built-in HW audio filters, or increase/decrease the speaker gain of a specific channel. The control thread could simply write the commands directly into the control register timing slots of the Xmt ping-pong buffers so that the control data is sent out to the device by the device driver during normal operation. An example of this functionality is shown in the code example at the beginning of this section.

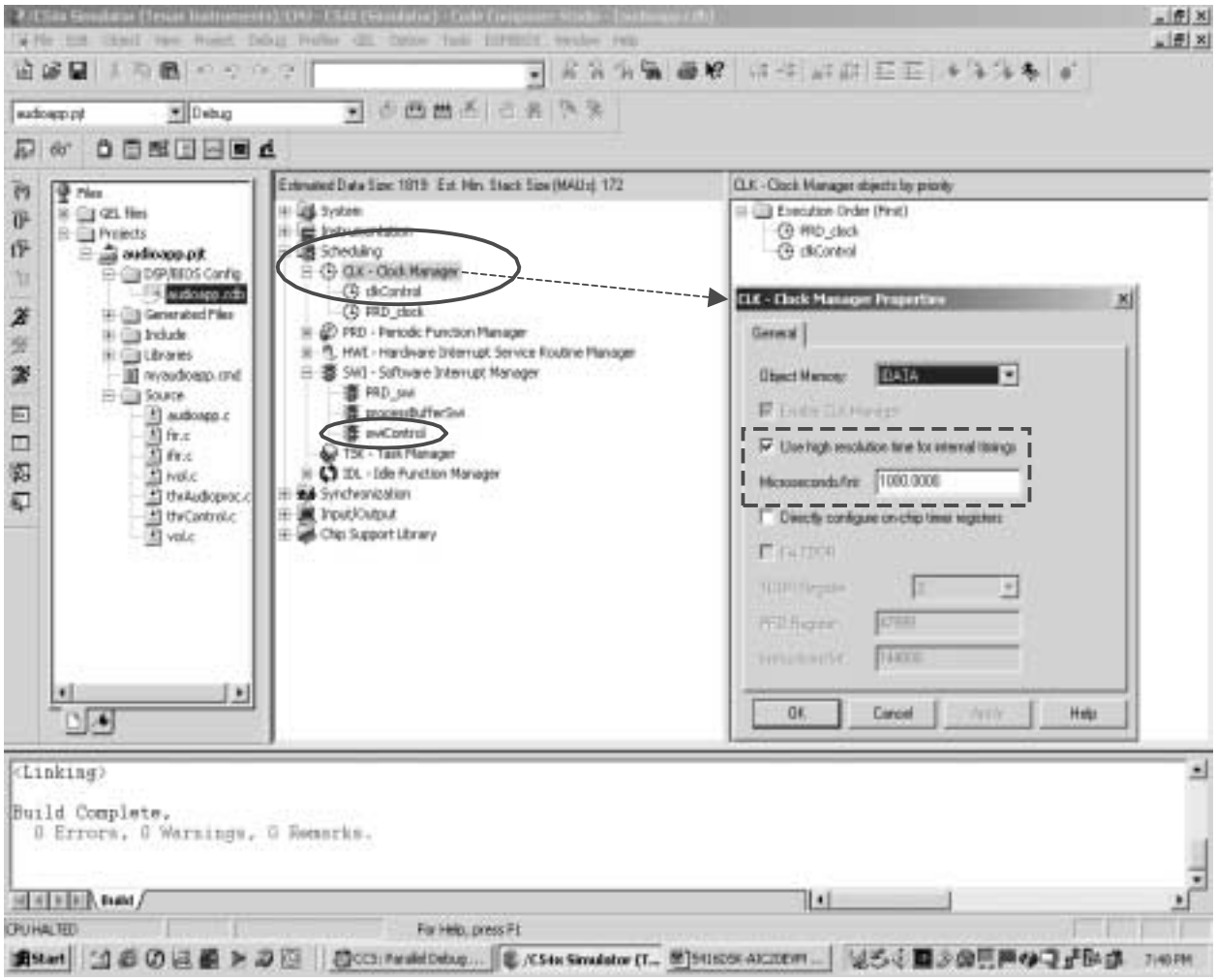


**Figure 20. DSP/BIOS Timer ISR and Control Thread Configuration**

The *clkControl* and *swiControl* threads are preconfigured in the provided reference framework's DSP/BIOS configuration (*.CDB) file. Every CLK object has an associated function which runs in the context of the timer ISR that has a period set in the clock manager properties (microseconds/Int). The `thrControlISR()` function is associated with the *clkControl* CLK object which is executed on each timer interrupt which occurs every 1000 µs (i.e. 1 ms).

### 8.5 Development of System-Specific AIC20 Device Drivers

The provided AIC20 device driver code can be completely reused to develop AIC20 device drivers for just about any specific target hardware platform. The best method would be to make a copy of the Code Composer Studio project that accompanies this application note, modify it for a specific target hardware platform, and then build a specific application on top of the provided framework. There is also a small CCS project file (dsk5416_aic20evm_l54f.pjt) located in the *drivers* subdirectory that can be used to rebuild the device driver library file. Use this project file to build your own custom AIC20-based driver libraries and follow the same naming conventions (i.e. <BOARD>_<DEVICE> prefixes for all file names, code labels, and global symbols).

## 9 Conclusion

Writing device driver code from scratch can be a tricky, tedious, and time-consuming process. It involves the lowest level of understanding details with respect to the host processor peripherals that connect to the device itself, and the precise timings must be understood on both sides to determine the correct interactions between host and device.

A reference framework with a reusable, portable, configurable, modular, production quality, and ready-to-run device driver has been developed to aid the DSP system designer who is evaluating or using a TMS320C54x-based processor and the TLV320AIC20 HPA voice-band data converter in the system design. The provided production-quality *C* source code, along with the plethora of software components and development tools offered through TI's eXpressDSP™ software strategy, allow the designer to get started quickly and even use the baseline code for actual production purposes to get to market faster. The sample framework is also useful for rapid prototyping, as well as evaluating various DSP algorithms working in conjunction with the TLV320AIC20 dual-channel voice-band codec on a TMS320C54x™ DSP platform, TI's most popular family of digital signal processors.

## 10 References


1. *Codec Evaluation System* (SLAA141)
2. *Demo/Test Codec Systems with TLV320AIC20/21/24/25 EVM* (SLAA153)
3. *TLV320AIC20, Low Power, Highly-Integrated Programmable 16-Bit 26-KSPS Dual Channel Codec* (SLAS363)
4. *TLV320AIC20, TLV320AIC21, TLV320AIC24, TLV320AIC25 EVM* (SLAU088)
5. *DSP-Codec Development Platform* (SLAU090)
6. *TMS320VC5416 Fixed-Point Digital Signal Processor* (SPRS095)
7. *Reference Frameworks for eXpressDSP Software: RF1, A Compact Static System* (SPRA791)
8. *Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System* (SPRA793)
9. *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795)
10. *TMS320C54x DSP/BIOS Users Guide* (SPRU326)
11. *TMS320C5000 DSP/BIOS API Reference* (SPRU404)
12. *DSP/BIOS by Degrees: Using DSP/BIOS Features in an Existing Application* (SPRA591)
13. *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802)

14. *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
15. *TMS320 DSP Algorithm Standard API Reference Guide* (SPRU360)
16. *TMS320C54x DSP Reference Set, CPU and Peripherals, Volume 1* (SPRU131)
17. *TMS320C54x DSP Reference Set, Enhanced Peripherals, Volume 5* (SPRU302)
18. *TMS320C54x, Chip Support Library API Reference Guide* (SPRU420)
19. *TMS320C54x Code Composer Studio Tutorial* (SPRU327)
20. *Real-Time Data Exchange: A White Paper* (SPRY012)
21. *How to Write an RTDX Host Application Using MATLAB* (SPRA386)

## IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265