TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Host Side Design

Xiangdong Fu
Zhaohong Zhang

C6000 Applications

ABSTRACT

This application report provides an overview of the host side design in the Multichannel Vocoder (MCV) Technology Demonstration Kit (TDK). Emphasis is given to overall system architecture and the major system components. The MCV TDK is currently running on TMS320 C6201/C6701 EVMs. All algorithm components are compliant with the eXpressDSP™ Algorithm Standard and are run-time configurable.

The MCV TDK consists of two parts:

- Host application (host)
- DSP application (target)

The host application sends commands to the digital signal processor (DSP) for run-time channel and I/O configuration and for streaming data between the host and DSP. The host is a Win32 application built for Windows 95/98 and Windows NT.

This application report highlights the host-side design of the MCV TDK. The DSP-side design is described in the application report, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Target Side Design* (SPRA560) [1].

Three major function modules are implemented in the host-side application.

- Host command module
- Real-time data transmission module
- Instrumentation module

This document begins with a brief introduction followed by a detailed discussion of the design for each function module.

Contents

List of Figures

1 Introduction

Figure 1 shows the high-level architecture of the multichannel vocoder technology demonstration kit. This demonstration presents a fully functional multichannel, multifunction application running on the Texas Instruments (TI™) TMS320C6000 DSP. The demonstration consists of two distinctive parts:

- DSP-side application (target)
- Host-side application (host)

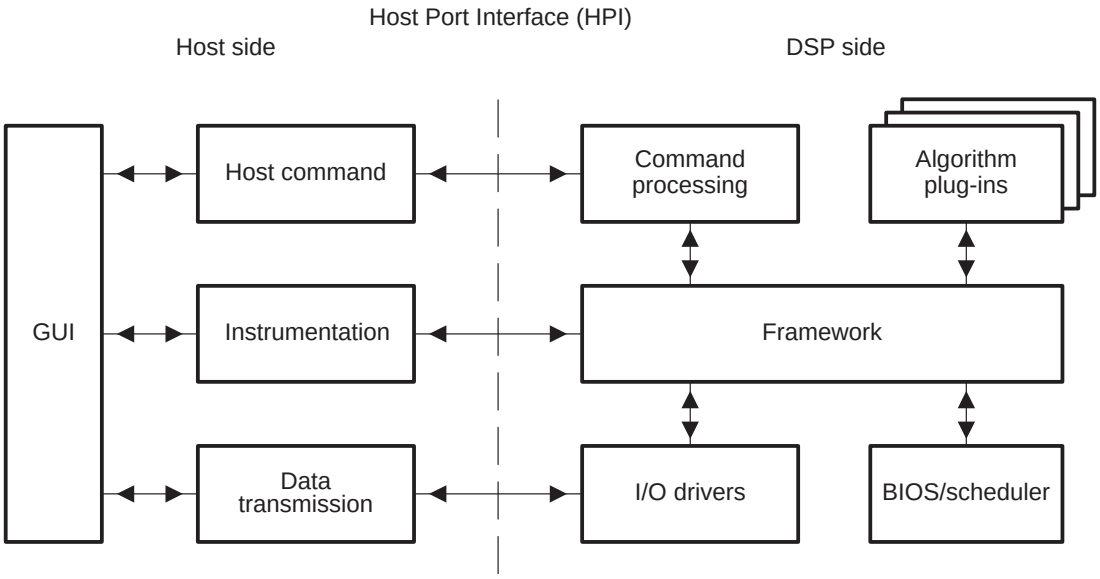

Figure 1. System Diagram of the Multichannel/Algorithm Framework Demonstration

The host application sends commands to the DSP for run-time channel and I/O configuration and for real-time data transmission between the host and DSP. The host is a Win32 application built for Windows 95 and has an interactive GUI to allow run-time system configuration as well as channel instrumentation.

The DSP application consists of up to 10 concurrently active processing channels. Each active channel has at least one input I/O stream, one output I/O stream, and up to three different algorithms. For each channel, the underlying framework gets input data from the assigned I/O input port, uses assigned algorithm/algorithms to process the data, and sends the outputs to the assigned I/O output port. The host performs all reconfigurations of I/O ports and algorithms to a certain channel.

This application report highlights the host-side design of the MCV TDK. The DSP-side design is described in the application report, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Target Side Design* (SPRA560) [1].

Figure 2 shows the three major function modules implemented in the host processor.

- Host command module
- Real time data transmission module
- Instrumentation module

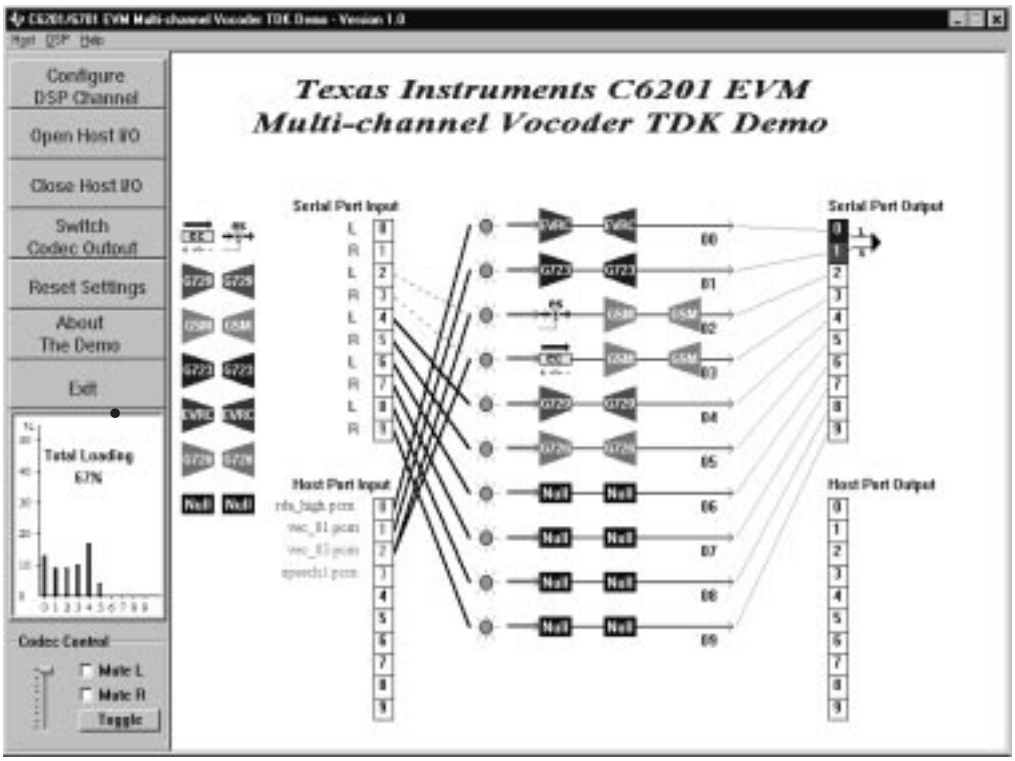

Figure 2. GUI of the Multichannel/Algorithm Framework Demonstration

The host command module sends commands to the DSP to perform hand shaking, to configure channels and I/Os, etc.

The real-time data transmission module streams data between the host and DSP. The data are voice-related—either raw PCM or coded bit-stream.

The instrumentation module gathers instrumentation data from the DSP to calculate and display a loading meter for each and every channel as well as the total loading for the DSP.

As a whole, the host application serves as the brain of the target application and provides its link to the outside world.

The following three sections discuss in detail the design of each module. Flowcharts and source code listings are used to illustrate the designs.

2 Host Command Module

2.1 Overview

The multichannel vocoder TDK provides run-time configuration of channel components (algorithms) and I/Os. For example, channel No.1 is originally idle and is reconfigured at run time to run G.729 voice compression algorithm, plus G.165 echo chancellor with input from serial port channel #1 and output to serial port channel #3. A set of commands is dedicated to configuration. To synchronize the DSP and host processor after reset, hand shaking must be performed, and a set of commands is defined for this purpose. There are also commands to allow the host processor to get updated information from the C6201 DSP.

All commands are initiated from the host-side application. The host is a Win32 application with an interactive graphic user interface to allow users to easily configure channel components and I/Os on the fly. The DSP acts as a pure slave.

On the host side, when a user initiates a command though the GUI, it is parsed and sent to the host command module. The host command module then packs the command and associated data into the host command buffer, transmits the buffer to the DSP though the host-port interface (HPI), and waits for the DSP to respond to, for example, the processing of PERFORM_HAND_SHAKING, as shown in Figure 3.

On the DSP side, the command-processing module is called in the idle loop. Inside this module, the NewCommandFlag in the host command buffer is checked. If it is equal to 0, no new command is waiting for processing; the call simply returns. If the flag is equal to 1, there is a new command and the command ID and associated data is interpreted and processed accordingly. This is illustrated in Figure 4, ”PERFORM_HAND_SHAKING”.

2.2 Host Commands

The enumeration variable CID defined below includes all commands currently in use.

```
typedef enum{
/* commands for Host/DSP hand shaking                  */
  PERFORM_HAND_SHAKING = 0X10000001,
  CONFIRM_HAND_SHAKING = 0X10000002,

/* commands for set/get channel configuration          */
  SET_CHANNEL_CONFIG = 0X20000001,
  GET_CHANNEL_CONFIG = 0X21000001,

/* commands for set/get I/O configuration              */
  SET_IO_CONFIG      = 0X20000002,
  GET_IO_CONFIG      = 0X21000002,
/* commands for set/get algorithm name & configuration */
  SET_ALG_PARAM          = 0X20000003,
  GET_ALG_PARAM          = 0X21000003,
  GET_ALGORITHM_NAME     = 0X21000004,
/* commands for audio codec manipulation                 */
  MUTE_CODEC_LEFT        = 0X30000001,
  MUTE_CODEC_RIGHT       = 0X30000002,
  SWITCH_CODEC_OUTPUT    = 0X30000003,
  SET_CODEC_GAIN         = 0X30000004,
  /* commands for get host IO buffer configuration      */
  GET_HOSTIO_CONFIG      = 0X40000001,
  GET_HIO_CTRL_ADDR      = 0X40000003
  /* No command                                         */
  NO_COMMAND             = 0X00000000,
} CID;
```

2.3 Host Command Buffer

The centerpiece of the host command protocol is the following data structure.

```
typedef struct{
  volatile int NewCommandFlag;   /* set by Host, clear by DSP      */
  volatile int CommandFinishFlag;/* set by DSP, clear by Host      */
  volatile CID CommandID;        /* command type, set by DSP       */
  volatile int CommandData[COMMAND_DATA_SIZE];
                                 /* data associated with a command */
                                 /* modified by both Host and DSP  */
} HCBUFFER;                      /* command buffer                 */
```

Notice all of the variables in the above data structure are defined as volatile. The command buffer is located in a pre-determined space in the DSP memory known to both the host and DSP; the host maintains a shadow of this buffer in its own memory. The host fills the shadow buffer with the proper command ID and associated data before copying it to the DSP through the HPI. Some commands require feedback from the DSP after the command is executed successfully. Figure 3 and Figure 4 show the flowcharts of the execution process for the command PERFORM_HAND_SHAKING. Other host commands are executed in a similar fashion.

2.4 Host Command Module Flow Charts

Figure 3 and Figure 4 show the flowcharts for processing the command PERFORM_HAND_SHAKING for the DSP side and the host side, respectively.

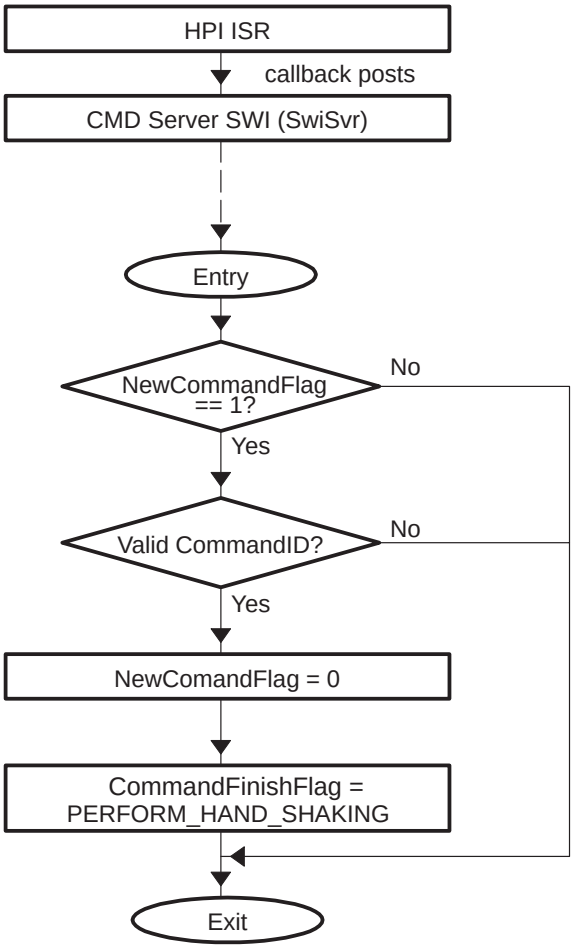

Figure 3. Handshaking (DSP Side)

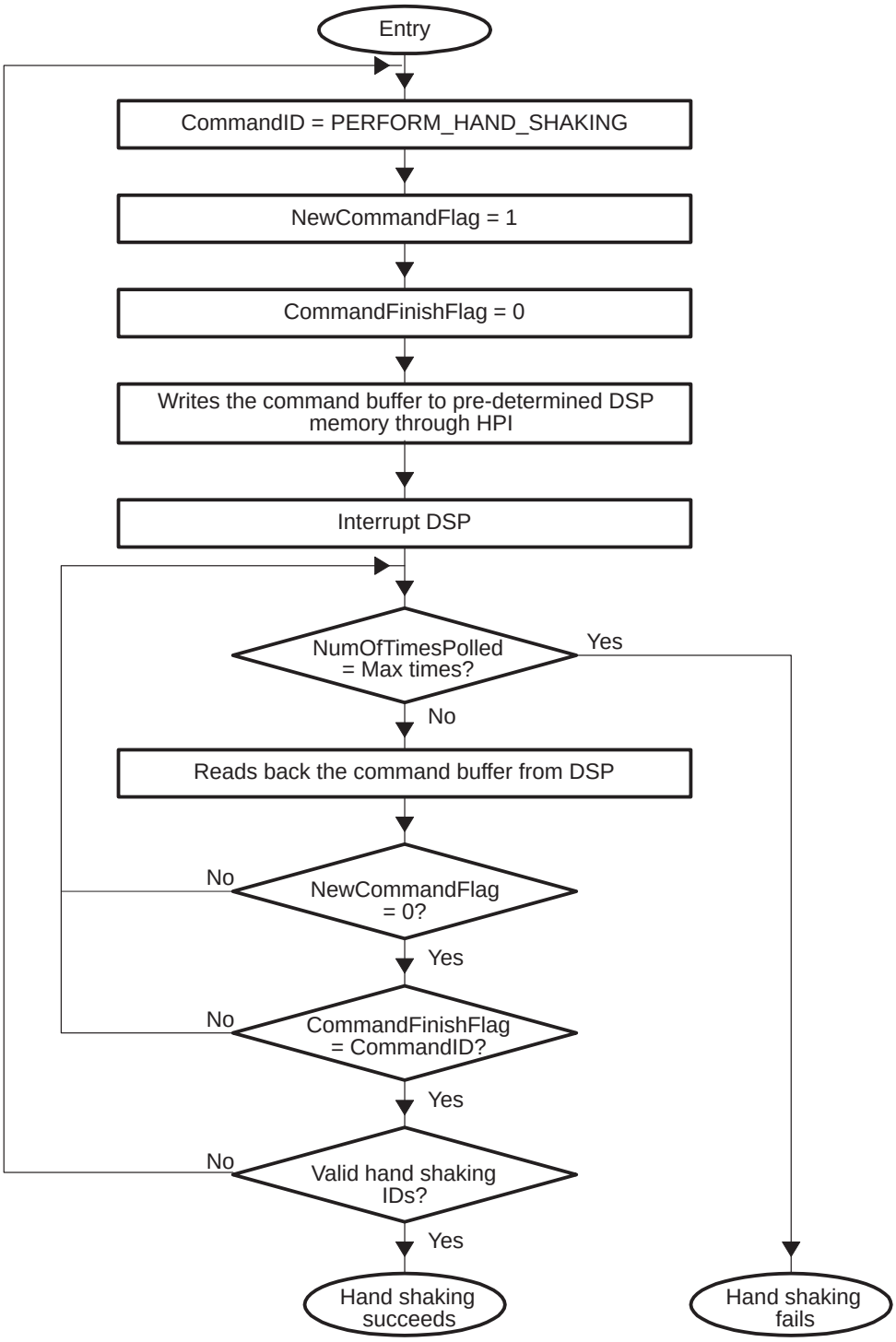

Figure 4. Hand Shaking (Host Side)

3 Real-Time Data Transmission

3.1 Overview

In a telecommunication system such as a voice-over-internet protocol (VoIP) gateway, The C6000 DSP receives voice data from the TDMA bus through its serial port. The DSP then processes the data by using appropriate algorithms (for example, line echo cancellation plus G.729 voice coder) and sends the compressed data to the host through its host port interface (HPI). The host processor then packs the data to TCP/IP or UDP packets before sending them to the Internet.

Similarly, packets come from the network to the host processor, where they are unpacked and sent to the DSP, decompressed back to voice and enhanced, and finally sent out to the TDMA bus. Figure 5 shows the data flow.

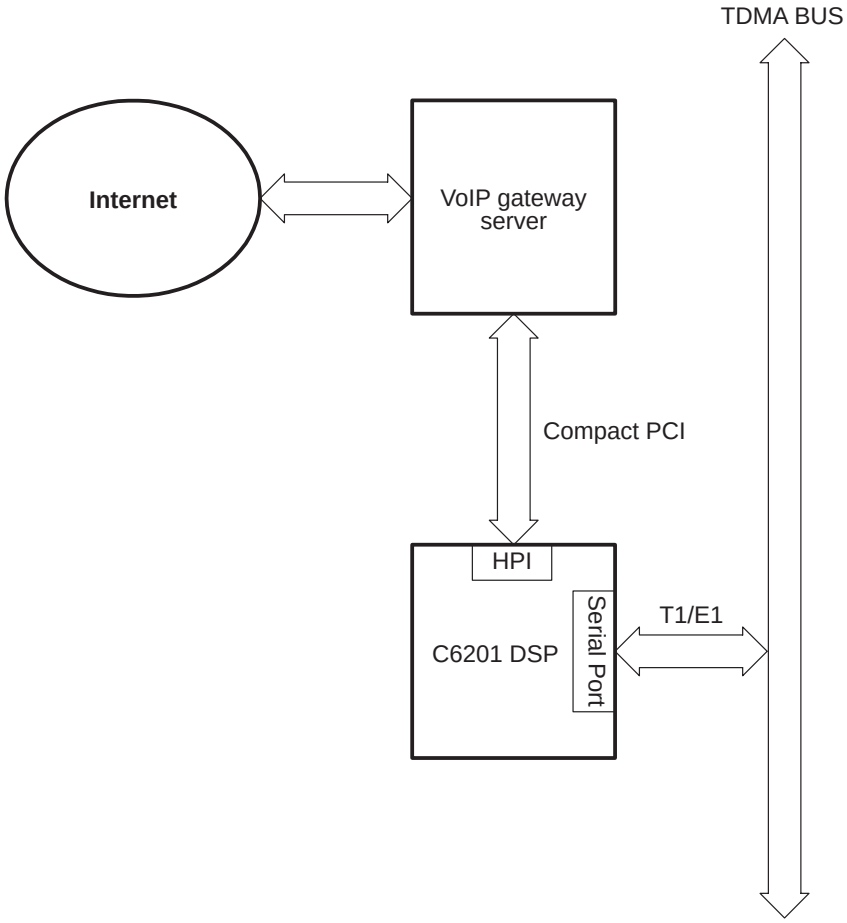

Figure 5. Simplified VoIP Gateway

In the multichannel vocoder demo, the data flow is simulated between the VoIP gateway and the DSPs (highlighted arrow in). Outbound data (DSP to host) is stored on the hard disk of the host PC as data files instead of going out to the network. Inbound data (host to DSP) is read from data files before being sent to the DSP.

The code discussed here is actually the host counterpart of the host data driver in the Board Support Library. Please refer to [2] for more details.

3.2 Major Design Concerns

Real-time data transfer between the host and DSP is not a trivial task. The major concerns are:

- Bandwidth restrictions
- Synchronization
- Signaling

3.2.1 Bandwidth Restrictions

Because the data transmission module shares the DSP's host port interface with the host command module and the instrumentation module, the question arises, does the C6201 HPI have enough bandwidth to handle all of the data flows? From [2], we know the maximum data transfer speed between the host and the C6201 200-MHz DSP through the HPI is between 24 Mbps and 50 Mbps, assuming the data goes to the external SD-RAM. There are 20 channels of host I/O now in the framework. The maximum bandwidth requirement is 1.28 Mbps (20×64 Kbytes/second). The bandwidth requirement of the command and instrumentation module is minimal compare to this; thus, the C6201 HPI has enough bandwidth to handle all of the data flows, which is also proved by the demonstration itself.

3.2.2 Synchronization

To avoid underflow and/or overflow, data flow must be synchronized between the host and DSP. This is no problem for outbound data, which comes from the synchronized TDMA bus. However, this is not the case for inbound data because data from the Internet is generally asynchronous.

To simulate this nonsynchronization, a Win32 timer is used as the source to trigger data transmission. The Win32 timer's best resolution is greater than 50 ms and the response delay to the timer by the OS kernel also varies wildly. According to our experiment, the delay can be as large as 300 ms. Thus, Using the WIN32 timer is a good simulation of the highly asynchronous data traffic from the Internet.

Finally, to avoid data overflow and/or underflow, a large jitter buffer is needed for each host I/O channel. Currently the jitter buffer size is up to 900 ms.

3.2.3 Signaling

Whenever the host finishes reading/sending a chunk of data to/from the DSP, it signals the DSP immediately. The DSP then updates the status of the host I/O buffers accordingly.

Whenever the host signals the DSP, DSP must respond promptly. Delays in response may cause false overflow/underflow conditions. To ensure minimum delay, the DSP's host interrupt is used as the signal.

3.3 Core Data Structures

3.3.1 DSP-Side Data Structures

3.3.1.1 I/O Data Buffer

```
static far volatile char HostInBuff[PORT_CT][BLOCKCT * BLOCKSZ];
static far volatile char HostOutBuff[PORT_CT][BLOCKCT * BLOCKSZ];
```

The I/O data buffer is a super buffer defined and partitioned into sub-buffers for each I/O port, including the serial port I/Os and host port I/Os.

3.3.1.2 Buffer Control Arrays

```
static volatile int OffsetIn[PORT_CT];
static volatile int OffsetOut[PORT_CT];

static volatile int DataCtIn[PORT_CT];
static volatile int DataCtOut[PORT_CT];

static volatile int FrmSzIn[PORT_CT];
static volatile int FrmSzOut[PORT_CT];

static volatile unsigned int MaskIn[PORT_CT];
static volatile unsigned int MaskOut[PORT_CT];
```

3.3.2 Host-Side Data Structures

3.3.2.1 I/O Data Buffer

```
short HostDataBuffer[NUM_OF_HOST_IN+NUM_OF_HOST_OUT][MAX_BUF_SIZE];
```

Each host I/O port has its dedicated data buffer. For host input (outbound) channels, data is read from the hard disk and stored in the PC memory before being written to buffers in DSP memory. For the host output (outbound) channels, data flow is in the opposite direction, from DSP memory to PC memory and finally the hard disk.

3.3.2.2 Buffer Control Data structure

```
typedef struct{
  ULONG Start;
  ULONG End;
  ULONG Current;
}HIO;
```

3.3.2.3 Buffer Control Arrays

```
HIO       HostIn[NUM_OF_HOST_IN];
HIO       HostOut[NUM_OF_HOST_OUT];
int frameSize[NUM_OF_HOST_IN+NUM_OF_HOST_OUT];
int Mask[NUM_OF_HOST_IN+NUM_OF_HOST_OUT];
volatile int NumOfDataInHostBuffer[NUM_OF_HOST_IN+NUM_OF_HOST_OUT];
```

These definitions are similar to those defined for the DSP.

3.3.2.4 Signaling

```
typedef struct
{
  volatile int DSPResponseFlag;/* DSP sets this flag          */
  volatile int OpFlag;  /* Host sets this flag         */
                              /* 0 none, 1 write, 2 read, 3 flush */
  volatile int dataBuffer[100];/* Signaling data transmitting */
                              /* between Host and DSP        */

} HIOCTRL;
```

This is similar to the definition of the host command buffer but without the NewCommandFlag (see the section,). Because the host interrupt is used to signal the DSP, no polling flag is necessary.

3.4 Real-Time Data Transmission Flow Charts

3.4.1 Data Transmission Module (DSP Side)

Figure 6, Figure 7, and Figure 8 show the flow charts of the HPI interrupt service routine (HPI ISR). This routine first checks the ReadWriteFlag in HIO_CtrlBuf, an instance of HIOCTRL defined on the DSP, then acts accordingly.

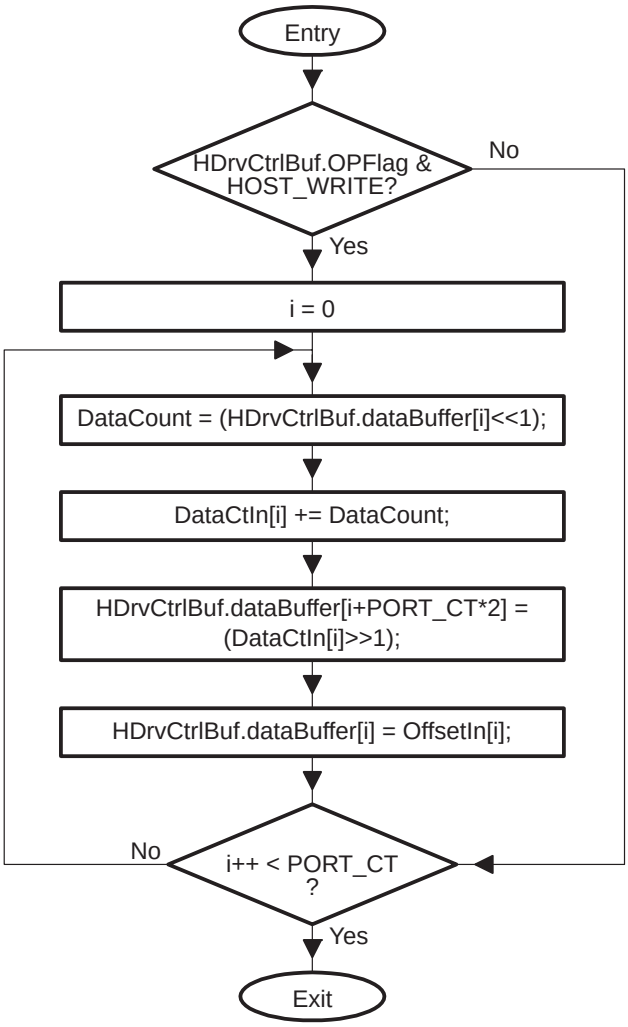

Figure 6. Data Transmission from the Host to the DSP (DSP Side)

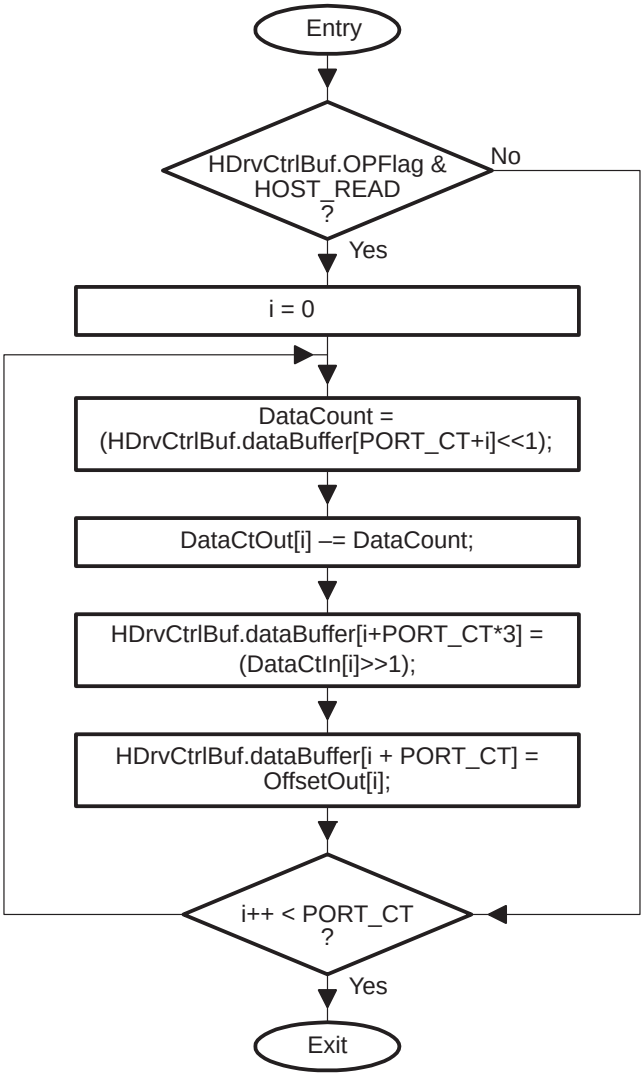

Figure 7. Data Transmission From the DSP to the Host (DSP Side)

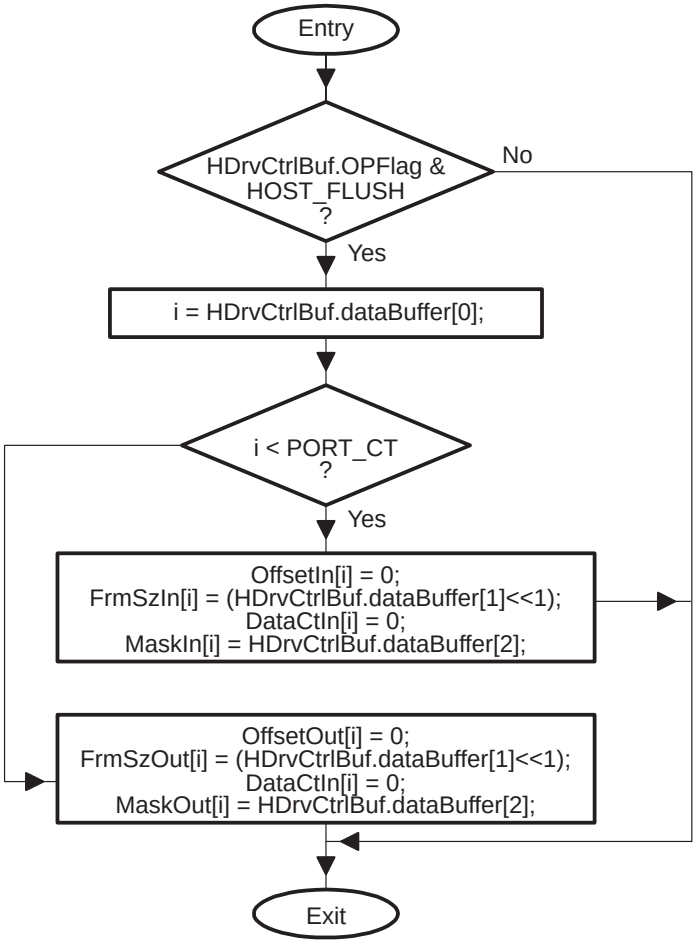

Figure 8. Flush a Host Output I/O Channel (DSP Side)

3.4.2 Data Transmission Module (Host Side)

Figure 9 shows the flow chart of Host Write action on the host side; that is, data is transmitted from the host to the DSP. Because Host Read is similar to Host Write, its flowchart is not shown.

Figure 9. Data Transmission from the Host to the DSP (Host Side)

4 Instrumentation

4.1 Overview

There is an instrumentation window in the GUI of the host-side application (see Figure 2). The window shows the average loading of each channel as well as the total loading of the DSP. The chart reflects average channel loading over the current two-second sampling period. In other words, the Host samples the DSP every two seconds. In that two-second period, Each time before and after a processing channel is processed, the DSP uses BIOS's clock mamager API function CLK_gethtime() to read the clock ticks of the on-chip high resolution timer. The DSP then calculates the difference between the two readings and adds that to an accumulator for that channel. Finally, the host reads the values in the accumulator at the end of the two-second period and reset it to zero.

4.2 Design Details

There is a counter defined on the DSP for each processing channel as well as a counter for the idle state, time spent outside any processing channel. Initially, all counters are zero.

On the DSP side, before executing a channel, a reading of the timer is taken by using CLK_gethtime(). The difference between this reading and the last one taken after executing another channel is accumulated into the counter for idle state. The channel is then executed and, immediately after the execution finishes, another reading to the timer is performed. The difference between this reading and the one taken before the execution of that channel is accumulated in to the counter for that particular channel.

On the host side, every 2 seconds the host reads values of the channel counters on the DSP. The host then uses the data read to calculate the loading of each channel as well as total loading of the DSP. It then resets the DSP channel counters to zero and updates the loading display on the GUI window.

4.3 Source Code Listings

Data structure definition

```
ACSBUFFER ActChaStatus; /* counters */
static int ActiveChaNum = FW_CCT;  /* FW_CCT refers to idle state */
static int CurTime  = 0;
```

Source code segment

```
/* measure current time in cycles, before channel execution*/
temp = CLK_gethtime();
/*update the counter for idle state, ActiveChaNum = FW_CCT here */
ActChaStatus.DSPActiveFlag[ActiveChaNum] += (temp -CurTime);
/* update Curtime */
CurTime = temp;
/* now ActiveChaNum points to current channel*/
ActiveChaNum = ChaNum;
/* execute the channel */
CM_Exec(ChannelTable[ChaNum].hCha, FrmPtr->InFrmPtr,
        FrmPtr->OutFrmPtr, SIG_DEF);
/* measure time again in cycles, after channel execution*/
temp = CLK_gethtime();
/* update the counter for current channel */
ActChaStatus.DSPActiveFlag[ActiveChaNum] += (temp -CurTime);
/* update Curtime */
CurTime = temp;
/* set ActiveChaNum back to idle state */
ActiveChaNum = FW_CCT;
```

Figure 10. Instrumentation: Source Code Listing 1 (DSP Side)

```
void GetDSPLoading()
{
  int i;
  double total = 0;
  ULONG readLength,writeLength;
  int trytimes = 0;
  /* begin critical section */
  g_cs.Lock();
  do{
    readLength = sizeof(ACSBUFFER);
    if(trytimes++ > 0){
      if(trytimes<TRY_TIMES) wait(10000);
      else{
        ErrorMessage();
        break;
      }
    }
  }while( !Board_hpi_read((ULONG *)(&ActiveChannelStatus),
       &readLength, DSPACSBuffer) ||readLength!=sizeof(ACSBUFFER));
  /* end critical section */
  g_cs.Unlock();

  /* calculate total cycle counts */
  for(i = 0; i<=NUM_OF_THREADS; i++){
    total += (unsigned long)ActiveChannelStatus.DSPActiveFlag[i];
  }
  /* calculate loadings for each channel */
  /* make the number in percentage */
  /* reset the loading counter for each channel */
  for( i = 0; i <= NUM_OF_THREADS; i++){
    DSPChannelLoading[i] =(int)((ActiveChannelStatus.
                    DSPActiveFlag[i] *100.0)/total);
    ActiveChannelStatus.DSPActiveFlag[i] = 0;
  }
  /* update the counters on DSP */
  do{
    writeLength = sizeof(ACSBUFFER);
    if(trytimes++ > 0) {
      if(trytimes<TRY_TIMES) wait(10000);
      else {
        ErrorMessage();
        break;
      }
    }
  }while( !Board_hpi_write((ULONG *)(&ActiveChannelStatus),
      &writeLength, DSPACSBuffer) ||writeLength!=sizeof(ACSBUFFER));
```

Figure 11. Instrumentation: Source Code Listing 2 (Host Side, called every 2 seconds)

5 Summary

This application report highlights the host-side design of the multichannel/algorithm framework based on the TMS320C6201 platform.

6 References

Refer to the following application reports to learn more about the multichannel vocoder TDK system on the TMS320C6000 DSP.

1. Xiangdong Fu and Zhaohong Zhang, *Multichannel Algorithm Implementation on the TMS320C6000 DSP,* SPRA556.
2. Xiangdong Fu, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Target Side Design*, SPRA560.
3. Xiangdong Fu, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit*, SPRA567.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.