Application Report
SPRA560B- February 2000

TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Target Side Design

Xiangdong Fu
C6000 Applications

ABSTRACT

This application report provides an overview of the target system 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 the TMS320C6201/C6701 EVM. All algorithm components are compliant with the eXpressDSP™ Algorithm Stand and are run-time configurable.

Contents

1 Introduction 2
2 System Overview 3
3 Chip Support and Board Support Library 4
4 Channel Manager Library 5
5 Algorithm Components 7
6 FW Module 8
7 Host Command Server 10
8 References 12
Appendix A Channel Manager API Reference 14
A.1 FRM_OBJ 14
A.2 CM_INFO 14
A.3 CM_Info 15
A.4 CM_ALG_INFO 15
A.5 CM_CHA_INFO 15
A.6 CM_Init 15
A.7 CM_Open 16
A.8 CM_Close 16
A.9 CM_SetAlgs 16
A.10 CM_GetAlgs 17
A.11 CM_GetInstStatus 17
A.12 CM_SetInstStatus 17
A.13 CM_RetAlg 18
A.14 CM_Exec 18
A.15 CM_Control 18
Appendix B IALG Interface and XDAIS Programming Rules 19

eXpressDSP is a trademark of Texas Instruments Incorporated.

List of Figures

Figure 1. Multichannel Vocoder TDK 1.0 Target System Block Diagram 3
Figure 2. Execution Flow Driven by Data Related Events 4
Figure 3. Execution Flow Driven by Command Related Events 4
Figure 4. CSL and BSL 5
Figure 5. Multiple Channels in the Channel Manager 5
Figure 6. High Level View of a Channel 6
Figure 7. Block Diagram of the FW Module 9
Figure 8. FW Control Block 9
Figure 9. FW I/O Multiplexer 10
Figure 10. List of Host Commands 11
Figure 11. Command Server Execution Flow 12
Figure B–1. Definition of IALG_Fxns, the v-table 19

List of Tables

Table 1. Multichannel Vocoder TDK Algorithm Summary 8

1 Introduction

This application report provides an overview of the target system design of 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 the TMS320C6201/C6701 EVM. All algorithm components are XDAIS-compliant and are run-time configurable.

The organization of this application report is as follows: An overall system overview is followed by detailed descriptions on the implementation of each individual functional module.

2 System Overview

The block diagram of the Multichannel Vocoder TDK target system is illustrated in Figure 1 below.

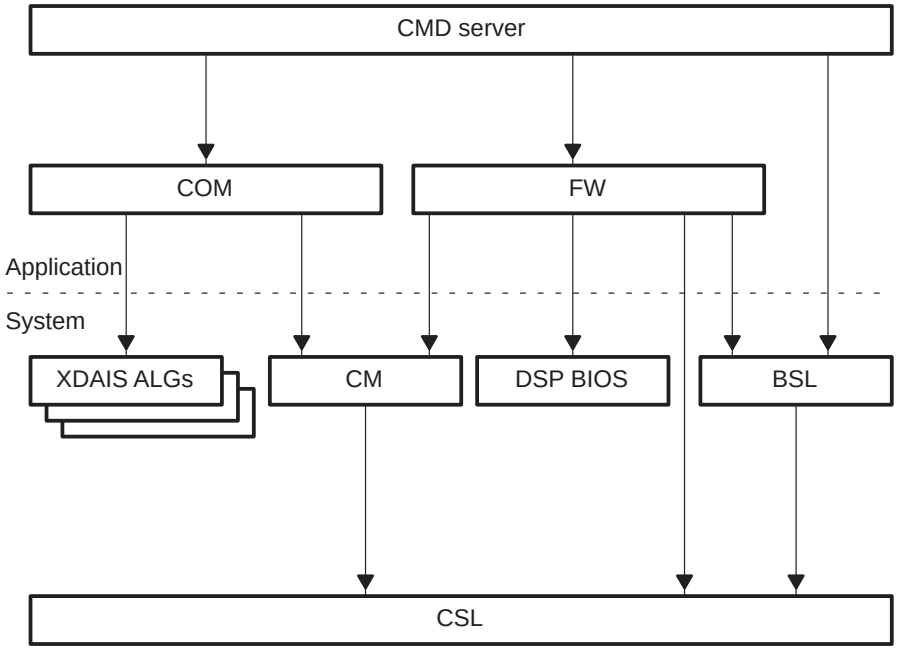

Figure 1. Multichannel Vocoder TDK 1.0 Target System Block Diagram

The major system components include the following:

- XDAIS compliant algorithms: XDAIS compliant vocoder algorithms, including ITU G.723.1, G.729/A, G.726, GSM-EFR and EVRC.
- BSL and CSL: the Multichannel Run-Time Environment (MCRTE) chip support and board support library, which provide all chip level and board level peripheral accesses, including DMA, serial port, audio codec and HPI, etc.
- CM: the MCRTE channel manager library, which is a C6000 optimized XDAIS algorithm framework that provides APIs for run-time context switching,
- DSP BIOS, which is used for scheduling and real time tracing and analysis,

On top of these generic libraries, there are three application specific modules:

- CMD Server: Host Command Server. CMD interprets and executes the commands from the Host PC. This is the interface module between the host and the DSP. Commands include those to get/set channel configuration, to get/set algorithm parameters, to query available algorithm components and to manipulate the audio codec on the ’C6201 EVM board.
- COM: the glue module between individual XDAIS algorithm components and the channel manager.

- FW: the module that can be best viewed as the BIOS extension of channel manager. It defines each channel as a BIOS software interrupt and uses BIOS PIP object to connect the I/O driver with the channel manager. Its API supports run-time configuration of channel setup and I/O connections.

Running under DSP BIOS, this real-time multichannel application is primarily event (interrupt) driven. System events trigger hardware interrupts (HWIs) threads that in turn post software interrupts (SWIs) threads. The DSP BIOS kernel schedules all those threads to run according to their priorities.

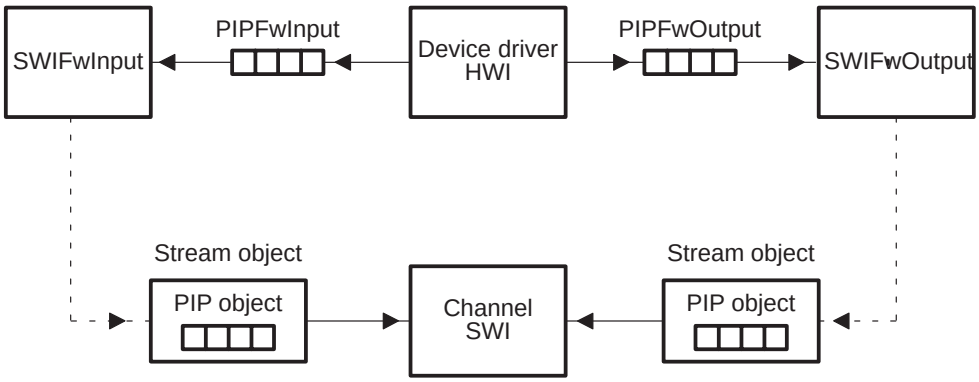

Figure 2. Execution Flow Driven by Data Related Events

There are two types of independent events that could happen in the system, the first type is data related and the second type is command related. Data related events happen periodically. These events notify the system that data is available for processing. The command-related events, on the other hand, happen randomly, depending on user activities. These events notify the system to execute a host command. Figure 2 and Figure 3 show the execution flow driven by these two events, respectively.

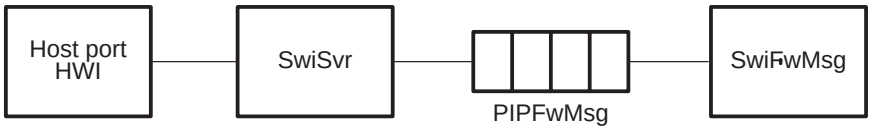

Figure 3. Execution Flow Driven by Command Related Events

3 Chip Support and Board Support Library

The chip support library (CSL) and the board support library (BSL) provide easy access to chip-level and board-level hardware resources through C callable APIs. The CSL is a collection of functional modules used to manage the various DSP on-chip peripherals such as the DMA controller, McBSP, EMIF, cache, etc. The BSL, on the other hand, is a collection of modules written for peripherals on a particular board containing the DSP, such as Texas Instruments (TI™) EVM. The BSL may interface into the CSL but not vice versa. Please refer to [1] and [2] for more information on CSL and BSL.

TI is a trademark of Texas Instruments Incorporated.

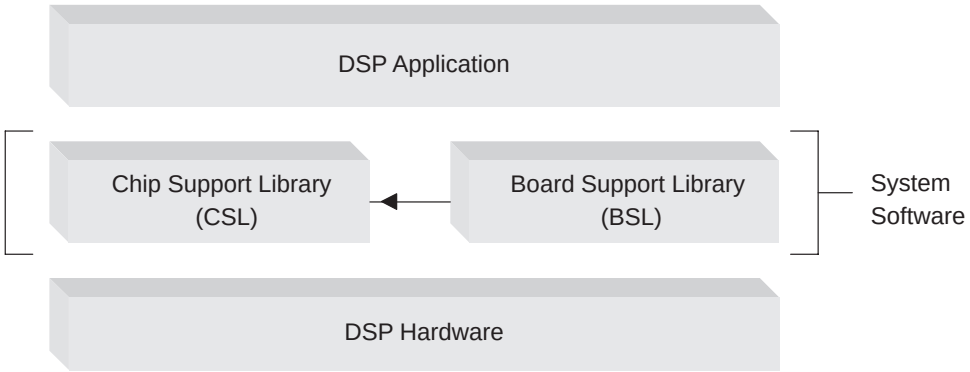

Figure 4. CSL and BSL

4 Channel Manager Library

The channel manager library (CM) provides APIs to manage multiple processing channels. Each processing channel consists of one or more algorithms such that the output of one algorithm is the input to the next algorithm, as illustrated in figure 5 below.

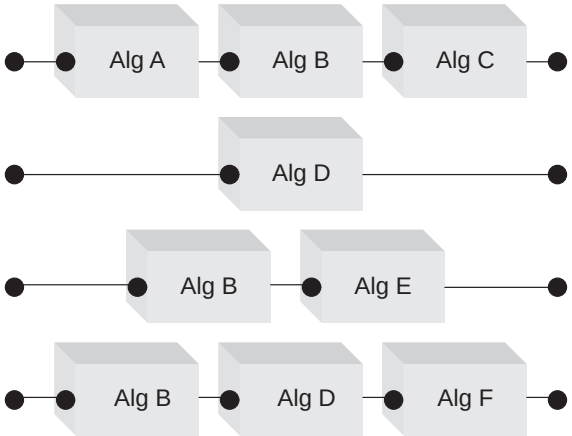

Figure 5. Multiple Channels in the Channel Manager

The channel manager library allows users to create these channels and assign algorithms to them. The algorithms are required to be compliant with the eXpressDSP Algorithm Standard (XDAIS) which the channel manager understands. All XDAIS compliant algorithms are reentrant and hence multiple instances of any given algorithm may be instantiated and placed into different channels.

The channel manager API provides routines and data types to:

- Open and close channels
- Configure channels
- Execute channels
- Assign algorithms to channels
- Register algorithms with the library
- Get and set algorithm parameters in a channel

Once a channel is opened and algorithms are assigned to it, it may be executed by calling the execute routine. The channel manager executes a channel from beginning to end in one atomic step and is not reentrant.

If we consider just one channel and bind some IO to it, we end up with what is shown in Figure 6. Here we have an input driver that puts input data into some device independent object denoted as an input stream. This IO object may be a DSP BIOS PIP object or maybe even a SPOX SIO object. Generally, once data is placed into the input stream, some event is triggered and a task runs that calls the channel manager to execute the channel. The channel manager will then read the input from the input stream, process the data through the channel algorithms, then put the final output data into the output stream. For the multichannel case, picture several of these hypothetical channels running together, all managed by the channel manager.

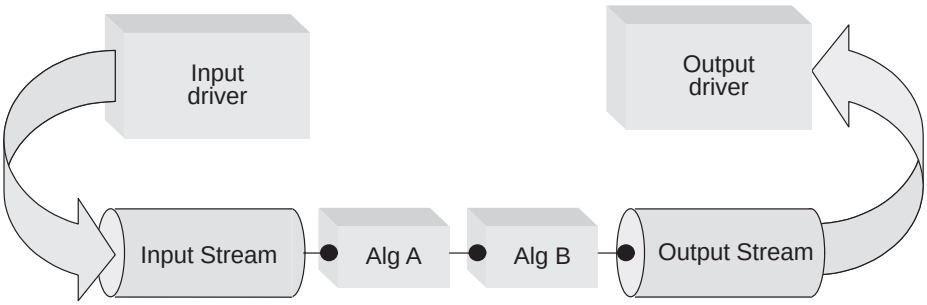

Figure 6. High Level View of a Channel

Although channels may be configured on the fly at run time, generally they are configured once at initialization then executed repeatedly based on input availability. Executing a channel is a fairly complex process. To maximize throughput, a software pipeline is established. In actuality, there are two nested software pipelines. The outer pipeline is at the channel boundary whereas the inner pipeline is at the algorithm boundary. The goal is to parallelize the data movement with the processing which is possible using the DSPs DMA controller.

Each algorithm has a certain set of data blocks associated with it.

- Input data
- Output data
- Context data
- Read-only table data

Input and output data is obvious. Context data is where the algorithm instance stores its private state information. Every algorithm instance has its own private context buffer, which stores state information used for future processing, for example, the IIR filer delay line. The data block an algorithm may optionally have is read-only table data. This data is needed by the algorithm but never changes; an example might be filter coefficients. Read-only data is treated specially because it needs to be transferred on-chip for use but does not have to be transferred back to off-chip when the algorithm is done with it.

For API functions of the channel manager library, see Appendix A.

5 Algorithm Components

The following algorithm components are currently in the MCV TDK target system. All of them are compliant with the eXpressDSP Algorithm Standard[16][17].

- ITU-T recommendation G.723.1, dual rate speech coding algorithm operating at 5.3/6.3kbps.[3][4][5]
- ITU-T recommendation G.729, a CS-ACELP speech coding algorithm operating at 8kbps. And G.729 Annex A, a reduced complexity version of G.729, yet fully bit-stream compatible with it. [6][7][8]
- ITU-T recommendation G.726, an adaptive differential pulse code modulation (ADPCM) speech coding algorithm operating at 16/24/32/40kbps. [9][10]
- IS-127 Enhanced Variable Rate Speech Coding algorithm (EVRC), an international standard for “Speech Service Option 3 for Wide-band Spread Spectrum Digital Systems”.[11][12]
- Enhanced Full Rate GSM speech coding algorithm, an international standard for wireless communications defined by the European Telecommunications Standards Institute (ETSI).[13][14]
- Line Echo Cancellation algorithm, compliant with ITU-T recommendation G.165.[15][16]

The following table shows the code size, data size and performance data for the algorithms above.

Table 1. Multichannel Vocoder TDK Algorithm Summary

Algorithm	Frame Size (ms)	Bit Rate (kbps)	Stack Size (Bytes)	Code Size (Bytes)	Context Size (Bytes)		Table Size (Bytes)	C6201 MHz
					Enc.	Dec.		
G.723.1	30	5.3 /6.3	7964	73600	1488	452	19000	14.4 /23.2
G.729/A	10	8	2764	86720	1948	1644	6032	25.6 /16.3
G.726	Configurable	16/24/32/40	196	24384	1188	1188	232	3.2
GSM EFR	20	12.2	5664	65536	3320	1740	10204	13.1
EVRC	20	Variable Rate	5748	91232	2772	1412	14312	26.0
G.165 LEC (32ms)	Configurable	NA	256	4512	2092		0	3.2

NOTE: The above algorithms are compiled from their respective C codes, some without any hand optimization. The program is running from off-chip memory using the internal program memory as cache on 'C6201 EVM.

In order for an algorithm to be XDAIS compliant, it must implement the IALG interface defined by the standard. In addition to that, it must also implement an algorithm specific interface defined by an algorithm interface creator, which extends the IALG interface in order to run the algorithm. The algorithm also needs to obey certain rules to be fully XDAIS compliant. Refer to [17][18][19][20][21] for more information on how to make an algorithm XDAIS compliant and on how to use XDAIS compliant algorithms in real-time DSP applications. For a quick reference on the IALG interface and programming rules, see Appendix B.

6 FW Module

The FW module primarily serves as a glue module between the Board Support Library and the Channel Manager. The FW module provides API calls for mapping device driver I/O to processing channels as well as API calls to change algorithm configuration of a underlying processing channel. It also contains the system error-handling task.

The FW module is BIOS dependent. It make uses of BIOS objects such as data pipe (PIP), software interrupt (SWI), log object (LOG_Obj) and the statistics object (STS_Obj) for I/O data transmission, message passing, scheduling, error-handling and real-time analysis.

There are actually two functional blocks in the FW module, as shown below:

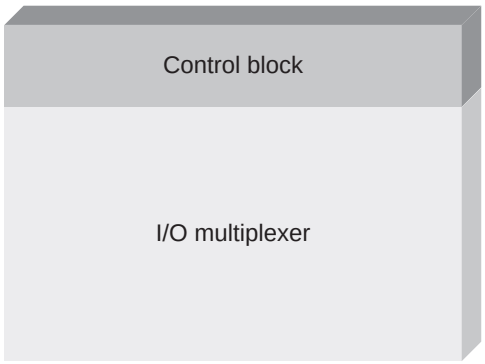

Figure 7. Block Diagram of the FW Module

The upper level control block of the FW module provides API function calls to access the underlying I/O multiplexer as well as the channel manager. The lower level I/O multiplexer maps I/O device drivers with the processing channel. It extends the processing channel object of the channel manager library to include links with the I/O device driver by using BIOS data pipes. The extended channel is then packaged into one software interrupt (SWI), which is scheduled by DSP BIOS. Figure 8 and Figure 9 below shows the block diagram of the FW control block and the I/O Multiplexer.

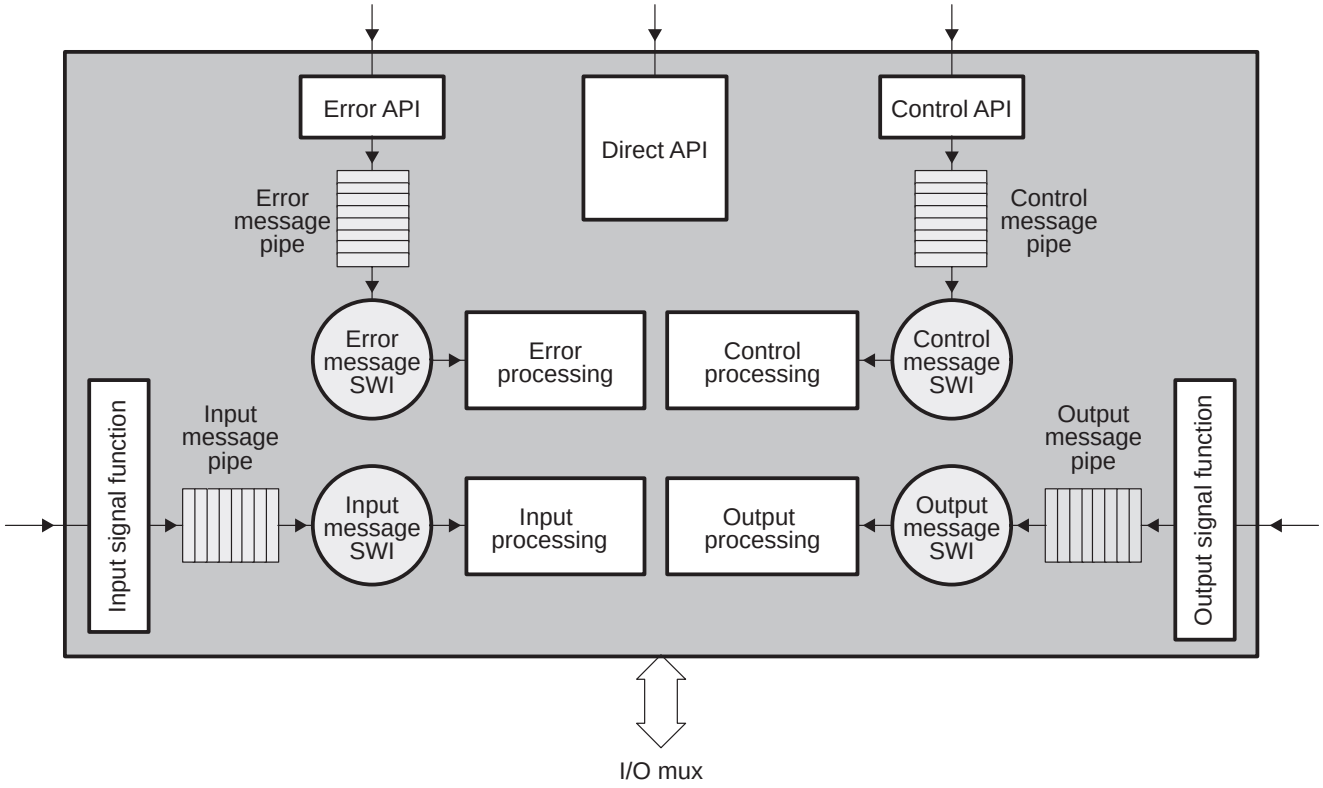

Figure 8. FW Control Block

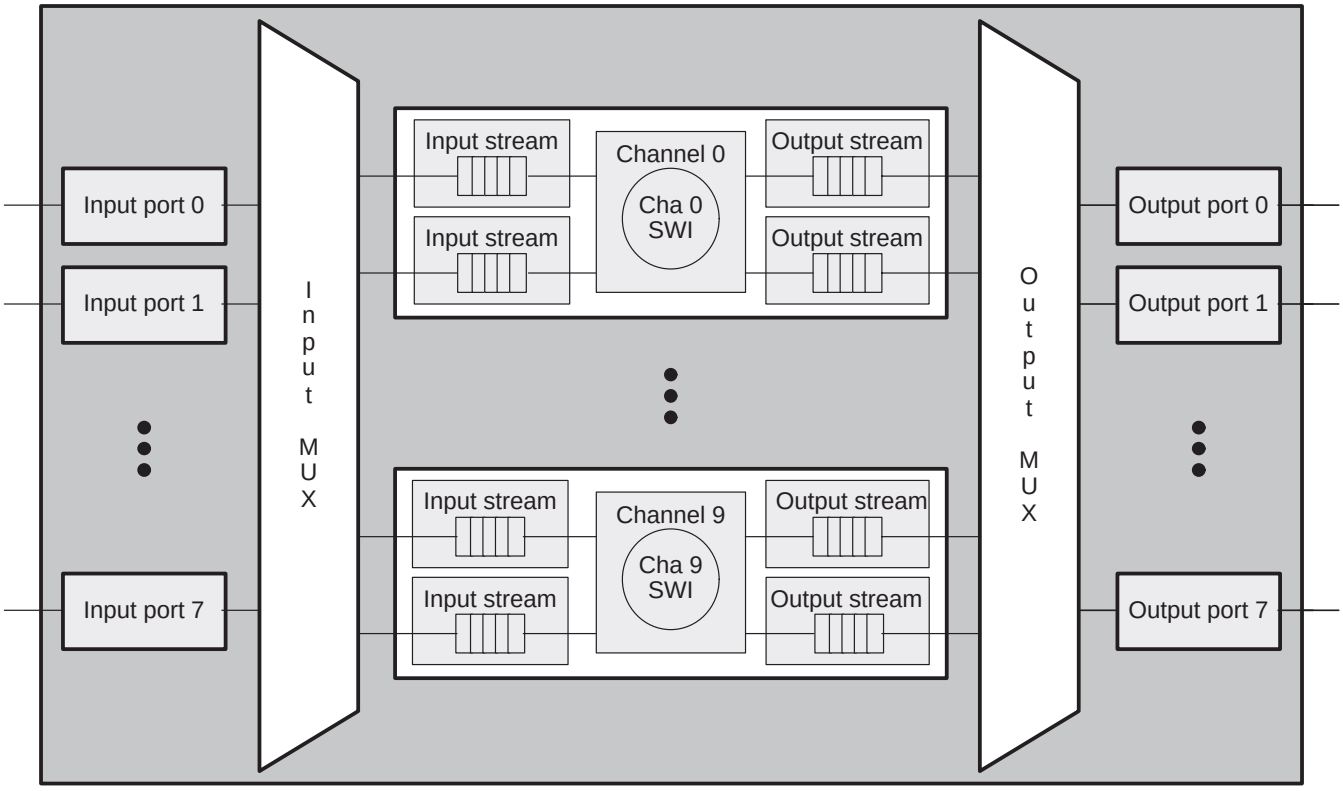

Figure 9. FW I/O Multiplexer

7 Host Command Server

The Host Command Server interprets and executes commands sent from the Host PC. This is the interface module between the Host GUI and the DSP application. Commands include those to get/set channel configuration, to get/set algorithm parameters, to query available algorithm components and to manipulate the audio codec on the ’C6201 EVM board. The following data structure shows 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  /* command ID */
```

Figure 10. List of Host Commands

To communicate with the DSP, the Host uses the HPI interrupt, which it shares with the Host data driver. The HPI ISR calls a callback function provided by the application at initialization time which posts a DSP BIOS SWI. The SWI then runs the host command server.

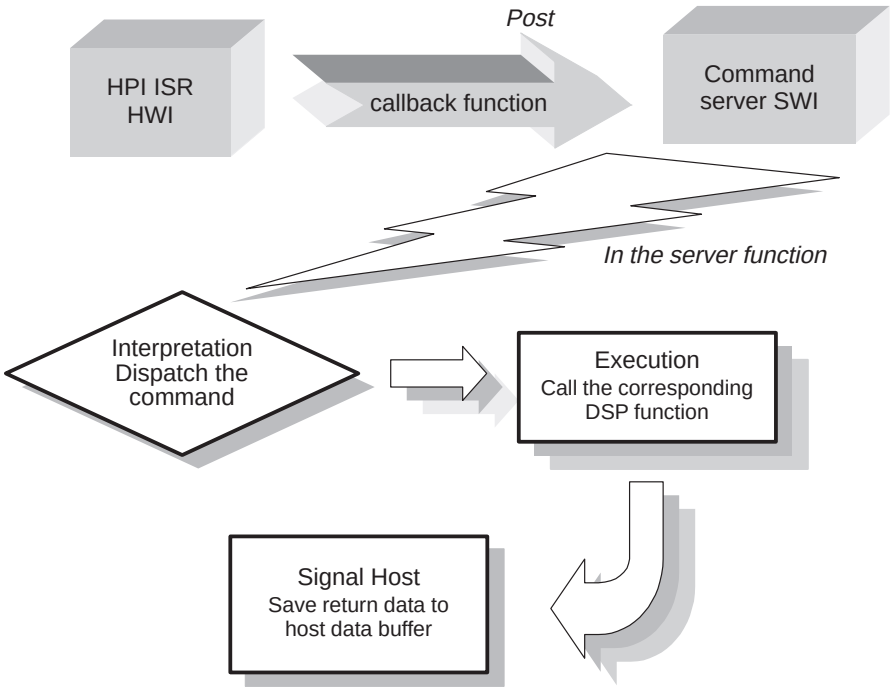

Figure 11. Command Server Execution Flow

Refer to [22] for the host side (client) implementation of the command-processing module.

8 References

Refer to the following documents and application reports to learn more about the MCV TDK on TMS320C6000.

1. *TMS320 C6000 Chip Support Library API Reference Guide*, to be published.
2. *TMS320 C6000 Board Support Library API Reference Guide*, to be published.
3. Thomas J. Dillon, Jr., *G.723.1 Dual Rate Speech Coder: Multichannel TMS320C6000 Implementation*, SPRA562.
4. *ITU-T Recommendation G.723.1 - Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*.
5. *ITU-T Recommendation G.723.1 - Annex A: Silence compression scheme*, November 1996.
6. Chiouguey Chen, *G.729 CS_ACELP Speech Coder: Multichannel TMS320C6000 Implementation*, SPRA564.
7. *ITU-T Recommendation G.729 - CS-ACELPD*.
8. *ITU-T Recommendation G.729 Annex A – Reduced complexity CS-ACELPD*, March 1996.
9. Chiouguey Chen, *G.726 ADPCM Speech Coder: Multichannel TMS320C6000 Implementation*, SPRA563.
10. Peter Dent, *Implementation of G.726 ADPCM on Echo Control for G165/DECT on Texas Instruments TMS320C62xx DSP*, Application Report BPRA066, October 1997.
11. Min Wang, *IS-127 Enhanced Variable Rate Speech Coder: Multichannel TMS320C6000 Implementation*, SPRA566.
12. TIA/EIA/IS-127, *Enhanced Variable Rate Codec, Speech Service Option 3 for Wide-Band Spread Spectrum Digital Systems*, January 1997.

13. Min Wang, *GSM Enhanced Full Rate Speech Coder: Multichannel TMS320C6000 Implementation*, SPRA565.
14. Peter Dent, *Implementation of Echo Control for G165/DECT on Texas Instruments TMS320C62xx Processors*, Application Report, May 1998.
15. Zhaohong Zhang, *Performance Analysis of Line Echo Cancellation Implementation Using TMS320C6201*, SPRA421.
16. *eXpressDSP Algorithm Standard (Rules & Guidelines)*, SPRU352.
17. *eXpressDSP Algorithm Standard (API Reference)*, SPRU360.
18. Stig Torud, *Making DSP Algorithms Compliant to the eXpressDSP Algorithm Standard*, SPRA579.
19. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Static DSP System, SPRA577*.
20. Carl Bergman, *Using the eXpressDSP Algorithm Standard in a Dynamic DSP System*, SPRA580.
21. Xiangdong Fu and Zhaohong Zhang, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit Host Side Design*, SPRA558.
22. Xiangdong Fu, *TMS320C6000 Multichannel Vocoder Technology Demonstration Kit,* SPRA567.

Appendix A Channel Manager API Reference

This section serves as a reference into the channel manager API library.

A.1 FRM_OBJ *Data Structure*

Typedef

```
typedef struct {
  void   *Addr;
  UINT32  Len;
  UINT32  Cap;
} FRM_OBJ;
```

Members

`Void *Addr`	Starting address of the data frame.
`UINT32 Len`	Length of the data frame (Bytes)
`UINT32 Cap`	Maximum size of memory containing this data frame (Bytes)

Description Structure definition of an I/O data frame

A.2 CM_INFO *Data Structure*

Typedef

```
typedef struct {
  const UINT32 ChaCt;
  const UINT32 AlgCt;
  const UINT32 AlgInstCt;
  const UINT32 MaxChaCt;
  const UINT32 MaxAlgCt;
  const UINT32 MaxAlgInstCt;
} CM_INFO;
```

Members

`const UINT32 ChaCt`	Number of CM channels currently open.
`const UINT32 AlgCt`	Number of algorithms registered with the library.
`const UINT32 AlgInstCt`	Number of algorithm instances currently in existence.
`const UINT32 MaxChaCt`	Maximum number of channels that may be open at once.
`const UINT32 MaxAlgCt`	Maximum number of algorithms that may be registered with the library.
`const UINT32 MaxAlgInstCt`	Maximum number of algorithm instances that may exist at once.

Description Structure definition of global channel manager library information.

A.3 CM_Info — *Global Variable*

Declaration `CM_INFO CM_Info;`

Description Global variable containing up-to-date information about the channel manager library.

A.4 CM_ALG_INFO — *Data Structure*

Typedef

```
typedef struct {
  char   *Name;
  UINT32  InputCt;
  UINT32  OutputCt;
} CM_ALG_INFO;
```

Members

`char *Name`	The name of the algorithm
`UINT32 InputCt`	Number of inputs the algorithm has.
`UINT32 OutputCt`	Number of outputs the algorithm has.

Description Used in CM_Control() to retrieve algorithm information from its handle.

A.5 CM_CHA_INFO — *Data Structure*

Typedef

```
Typedef struct {
  char    *Name;
  UINT32   AlgCt;
  UINT32   CopyMode;
} CM_CHA_INFO;
```

Members

`char *Name`	The name of the channel
`UINT32 AlgCt`	Number of algorithm instances currently in the channel.
`UINT32 CopyMode`	Data copy mode currently being used for the channel: • `CM_MEMCPY` • `CM_NOCOPY` • `CM_DMA`

Description Used in CM_Control() to retrieve channel information from its handle.

A.6 CM_Init — *Function*

Prototype `BOOL CM_Init();`

Arguments none

Return Value

`BOOL`	• TRUE - function succeeded • FALSE - function failed

Description Initializes the Channel Manager CM API module. Must be called at least once before calling any other CM API functions. This function calls SVC_Init().

A.7 CM_Open *Function*

Prototype

```
HANDLE CM_Open( char         *Name,
                CM_COPYMODE  CopyMode,
                SIG_OBJ      *Signal    );
```

Arguments

`char *Name`	Name of the channel
`CM_COPYMODE CopyMode`	Data copy mode currently being used for the channel: • `CM_MEMCPY` • `CM_NOCOPY` • `CM_DMA`
`SIG_OBJ *Signal`	Signal object used to post application upon each completion of running the channel.

Return Value `HANDLE` — Returns a handle to the open channel. `INV` is returned upon failure.

Description Opens a processing channel into the channel manager.

A.8 CM_Close *Function*

Prototype

```
void CM_Close(HANDLE hCm);
```

Arguments `HANDLE hCm` — Handle to previously opened channel.

Return Value None

Description Closes a CM channel.

A.9 CM_SetAlgs *Function*

Prototype

```
BOOL CM_SetAlgs(HANDLE hCm, UINT32 Count, HANDLE Algs[]);
```

Arguments

`HANDLE hCm`	Handle to an open channel.
`UINT32 Count`	Number of algorithms to assign
`HANDLE Algs[]`	An initialized array of algorithm handles to assign

Return Value `BOOL`
- TRUE - function succeeded
- FALSE - function failed

Description Assigns a set of algorithms to the channel.

A.10 CM_GetAlgs *Function*

Prototype `UINT32 CM_GetAlgs(HANDLE hCm, HANDLE Algs[]);`

Arguments

`HANDLE hCm`	Handle to an open channel.
`HANDLE Algs[]`	Array of algorithm handles to be filled in by this function

Return Value `UINT32` — Returns number of algorithms placed in `Algs`

Description Gets an array of algorithm handles for each algorithm currently in the channel

A.11 CM_GetInstStatus *Function*

Prototype

```
BOOL   CM_GetInstStatus(HANDLE hCha,
                        int    InstNo,
                        void*  algParams)
```

Arguments

`HANDLE hCha`	Handle to the channel
`int InstNo`	Number to identify the algorithm instance in the channel
`void* algParams`	Pointer to the algorithm parameter structure

Return Value `BOOL`

- TRUE - function succeeded
- FALSE - function failed

Description Get the parameter settings of an algorithm instance in a channel

A.12 CM_SetInstStatus *Function*

Prototype

```
BOOL   CM_SetInstStatus(HANDLE hCha,
                        int    InstNo,
                        void*  algParams)
```

Arguments

`HANDLE hCha`	Handle to the channel
`int InstNo`	Number to identify the algorithm instance in the channel
`void* algParams`	Pointer to the algorithm parameter structure

Return Value `BOOL`

- TRUE - function succeeded
- FALSE - function failed

Description Set the parameter of an algorithm instance in a channel

A.13 CM_RetAlg *Function*

Prototype

```
HANDLE CM_RegAlg(char *Name,        void  *algFxns,
                 void (*process)(), void  *algParams,
                 UINT32 InputCt,    UINT32 OutputCt);
```

Arguments

`char *Name`	Name of the algorithm
`void *algFxns`	Pointer to XDAIS IALG function pointer table
`Void*(*process)()`	Function pointer to algorithm processing routine
`Void* algParams`	Pointer to XDAIS algorithm parameter structure
`UINT32 InputCt`	Number of algorithm inputs
`UINT32 OutputCt`	Number of algorithm outputs

Return Value

`HANDLE`	Returns a handle to the registered algorithm. Returns INV if algorithm could not be registered.

Description Registers an algorithm with the library.

A.14 CM_Exec *Function*

Prototype

```
BOOL CM_Exec(HANDLE hCm, FRM_OBJ *In[], FRM_OBJ *Out[],
UINT32 Post);
```

Arguments

`HANDLE hCm`	Handle to an open channel.
`FRM_OBJ *In[]`	An array of pointers to input frames.
`FRM_OBJ *Out[]`	An array of pointers to output frames
`UINT32 Post`	Post value

Return Value

`BOOL`	• TRUE - function succeeded • FALSE - function failed

Description Executes channel

A.15 CM_Control *Function*

Prototype

```
UINT32 CM_Control(CM_CTRL_ID Id, ...);
```

Arguments

`CM_CTRL_ID id`	Control ID, may be one of the following: • `CM_RESET` • `CM_GET_CHA_INFO` • `CM_GET_ALG_INFO`
`...`	Additional arguments dependent on control ID.

Return Value

`UINT32`	Return value depends on control ID.

Description Executes a CM control function.

Appendix B IALG Interface and XDAIS Programming Rules

The IALG interface is the core interface defined by XDAIS that the algorithm is required to implement. It defines types and constants as well as the v-table, `IALG_Fxns`. Every algorithm needs to define and initialize a variable of type, `IALG_Fxns`, to declare the v-table. Some of the functions in the v-table are optional, while `algAlloc()`, `algInit()`, and `algFree()` are required.

```
typedef struct IALG_Fxns {

       /* Unique pointer that identifies the module implementing this interface           */
    Void *implementationId

       /* Notification to the algorithm that its memory is ”active” and algorithm         */
       /* processing methods may be called. May be NULL; NULL => do nothing               */
    Void (*algActivate)(IALG_Handle);

       /* Apps call this to query the algorithm about its memory requirements.             */
       /* Must be non-NULL.                                                                */
    Int  (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);

       /* Algorithm specific control operations.                                           */
       /* May be NULL; NULL => no operations supported.                                    */
    Int  (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);

       /* Notification that current instance is about to be ”deactivated”.                */
       /* May be NULL; NULL => do nothing.                                                 */
    Void (*algDeactivate)(IALG_Handle);

       /* Query algorithm for memory to free when removing an instance.                    */
       /* Must be non-NULL.                                                                */
    Int  (*algFree)(IALG_Handle, IALG_MemRec *);

       /* Apps call this to allow the algorithm to initialize memory requested             */
       /* via algAlloc().  Must be non-NULL.                                               */
    Int  (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);

       /* Apps call this whenever an algorithms object or any pointer parameters           */
       /* are moved in real-time. May be NULL; NULL => object can not be moved.            */
    Void (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);

       /* Query algorithm for number of memory requests.                                   */
       /* May be NULL; NULL => number of mem recs is less then IALG_DEFMEMRECS.            */
    Int  (*algNumAlloc)(Void);

} IALG_Fxns;
```

Figure B–1. Definition of IALG_Fxns, the v-table

The following list is rules and guidelines to be observed by an algorithm for its XDAIS compliance.

- Rule 1: All algorithms must follow the run-time conventions imposed by TI's implementation of the C programming language.
- Rule 2: All algorithms must be reentrant within a preemptive environment (including time-sliced preemption).
- Rule 3: All algorithm data references must be fully relocatable (subject to alignment requirements). That is, there must be no "hard coded" data memory locations.
- Rule 4: All algorithm code must be fully relocatable. That is, there can be no hard coded program memory locations.
- Guideline 1: Algorithms should minimize their persistent data memory requirements in favor of scratch memory.
- Guideline 2: Each initialization and finalization function should be defined in a separate object module; these modules must not contain any other

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.

Copyright © 2000, Texas Instruments Incorporated