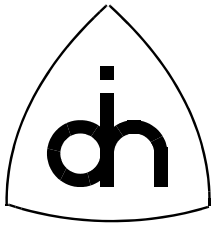
Odin TeleSystems Inc.



Programmer's Guide for OTX DSP C54x Software Development Kit

Doc. No. 1412-1-SAA-1007-1

Rev. 1.1

December 23, 2008

Changes

The material in this document is for information only and is subject to change without notice. While reasonable efforts have been made in the preparation of this document to assure its accuracy, Odin TeleSystems Inc., assumes no liability resulting from errors or omissions in this document, or from the use of the information contained herein.

Odin TeleSystems Inc. reserves the right to make changes in the product design without reservation and notification to its users.

Warranties

THE SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED “AS IS” AND WITHOUT WARRANTY OF ANY KIND. ODIN TELESYSTEMS EXPRESSLY DISCLAIMS ALL THE WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE. ODIN TELESYSTEMS DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET ANY REQUIREMENTS, OR THAT THE OPERATIONS OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS WILL BE CORRECTED. FURTHERMORE, ODIN TELESYSTEMS DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE SOFTWARE OR ITS DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVISE GIVEN BY ODIN TELESYSTEMS OR ODIN TELESYSTEMS’ AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY.

UNDER NO CIRCUMSTANCE SHALL ODIN TELESYSTEMS INC., ITS OFFICERS, EMPLOYEES, OR AGENTS BE LIABLE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS, PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE AND ITS DOCUMENTATION, EVEN IF ODIN TELESYSTEMS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT WILL ODIN TELESYSTEMS’ LIABILITY FOR ANY REASON EXCEED THE ACTUAL PRICE PAID FOR THE SOFTWARE AND ITS DOCUMENTATION. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL AND CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Odin TeleSystems Inc.
http://www.OdinTS.com

This document is published by:
Odin TeleSystems Inc.
800 East Campbell Road, Suite 334
Richardson, Texas 75081-1873
U. S. A.

Printed in U. S. A.

1. Table of Content

2. Introduction

The OTX (Odin Telecom FrameworX) DSP (Digital Signal Processor) C54x Software Development Kit (SDK) enables the users of OTX Hardware to develop their own DSP applications. The OTX DSP C54x SDK supports the Texas Instruments TMS320C54x family of Digital Signal Processors. The SDK can be used to develop DSP applications for the applicable OTX ASM Resource modules, such as Vidar-5x4-ASM-PRO and Vidar-5x4-ASM-EX, or for the OTX DSP NIC and Resource boards, such as Thor-2-PCMCIA-PRO and Thor-2-PCMCIA-EX. The different OTX products may utilize different members of the C54x DSP family. For example, Vidar-5x4-ASM-PRO utilize TMS320C5416 DSPs, while Vidar-5x4-ASM-EX utilize TMS320C5410A DSPs. This document uses TMS320C5416 in all of its examples. Although certain technical details, such as amount of memory, processor speed, etc., may vary between the different DSPs, the fundamental concepts remain valid for the whole DSP family.

The OTX DSP C54x SDK provides the software to allows user written DSP applications to be loaded by and communicate with user written host applications, as illustrated in Figure 1. More specifically, the SDK provides the user with the following:

- Minimal DSP Kernel which allows user applications to be loaded and run through the host bus (PCI, PCIe, or PCMCIA)
- Demo applications providing examples and full source code for working DSP applications.

- Command passing mechanism for communication between applications running in the host and in the DSP.

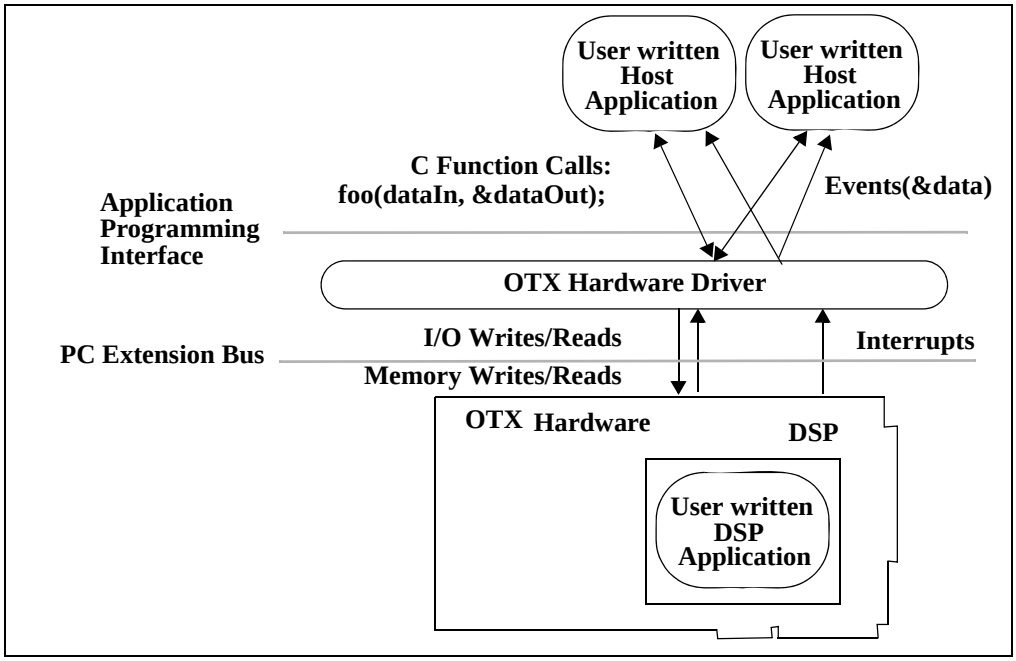


Figure 1. The DSP SDK allows user written DSP applications to be loaded by and communicate with a user written host application.

3. Distribution

The DSP SDK is distributed in a pkzipped file named OtxDspSdk<Rev>.zip, where Rev is the Revision number of the SDK. To use the DSP SDK, you will also need the Software Development Kit for the OTX Driver, as well as the Driver itself. The needed software is listed in Table 1.

TABLE 1. Odin Software needed for DSP Application Development

Odin Product Number	Description
SAA-1007-1	OTX DSP C54x SDK
SCA-1002-1	OTX Adapter Family Driver Set.
SAA-1006-1	OTX Hardware Driver Software Development Kit

It is important the all the used software, i.e. the Driver, the Driver SDK, and the DSP SDK are of the same revision. Always check the Odin homepage at* http:// www.OdinTS.com/sw.htm *for updates to the drivers. If you are updating the software, always update all of them to the same revision state.

4. Needed DSP Products, Tools, and Documentation

Before you venture into the DSP application development, you should verify that you have the needed tools and documentation available. First of all, you should have access to the Texas Instruments documentation on the TMS320C54x family of Digital Signal Processors. Recommended minimum documentation set is listed in Table 2.

TABLE 2. Recommended Texas Instruments TMS320C54x Documentation

Texas Instruments Literature Number	Description
SPRU131D	TMS320C54X DSP Reference Set. Volume 1: CPU and Peripherals
SPRU172	TMS320C54X DSP Reference Set. Volume 2: Mnemonic Instruction Set
SPRU179A	TMS320C54X DSP Reference Set. Volume 3: Algebraic Instruction Set
SPRS039A	TMD320C54x Fixed Point Digital Signal Processors Data Sheet

This documentation can be downloaded from the Texas Instruments Web site (**http://www.ti.com**) or they can be attained from your local Texas Instruments re-seller or distributor.

In addition to the basic documentation on C54x DSPs, you should have access to Odin's OTX adapter boards containing TMS320C54x DSP resources. Examples of OTX boards with these resources are:

- Thor-2-PCMCIA-PRO: 2 TMS320C5416s
- Vidar-5x4-ASM-CST: 4 TMS320C5416s
- Vidar-5x4-ASM-PRO: 4 TMS320C5416s
- Vidar-5x4-ASM-EX: 4 TMS320C5410A

The needed Odin Hardware is listed in Table 3.

TABLE 3. Odin Hardware needed for Custom DSP Application Development

Odin Product Number	Description
HAA-xxxx-y	OTX Adapter Board with on-board TMS320C54x DSP resources.
HMA-1057-1	Hermod-JTAG Code Composer Debug Probe (Used with OTX PCI Adapters)

And finally, to actually produce executable DSP applications, you will need software development tools, like Assembler, C Compiler and Debugger. The OTX adapters support standard Texas Instruments C Development tools. These tools can be purchased from any TI reseller. The corresponding original manufacturer's product numbers are listed in Table 4.

TABLE 4. Development tools from Any TI and DSP Research re-sellers

Manufacturer / Product Number	Product Description
Texas Instruments / TMDS324L855-02	TMS320C5000 PC C Compiler/Assembler/Linker v3.83 for Windows
Spectrum Digital. / 701905	XDS510 USB PLUS JTAG Emulator

You can find a listing of Texas Instruments distributors at **http://www.ti.com** and a listing of Spectrum Digital products at at **http://www.spectrumdigital.com**

5. C54x DSP Overview

This chapter provides a brief introduction to the Texas Instruments (TI) TMS320C54x DSP technology and to the terminology used later in this document. For in-depth information on the architecture, functionality, and operation of the C54x DSPs, please refer to the Texas Instruments documentation listed in Table 2 on page 6.

The processors in the TI TMS320C54x family are 16-bit fixed-point digital signal processors. The C54x DSPs feature a Harvard architecture with separate program and data buses providing a high-degree of parallelism during execution. The performance of the processors vary between 40 and 200 MIPS (Million Instructions Per Second) with faster and faster devices being introduced to the family.

5.1 Memory Organization

The C54x memory consists of three separate memory spaces:

- Program Space
- Data Space
- I/O Space

All C54x devices contain Random Access Memory (RAM) and certain C54x devices may also contain Read Only Memory (ROM). The on-chip ROM can be part of the program space or data space. On certain DSPs the ROM memory contains look-up tables for A-law and u-law companding.

The RAM memory comes in two different flavors:

- Dual Access RAM (DARAM)
- Single Access RAM (SARAM).

DARAM can be accessed twice per machine cycle which means that the DSP can read from and write to a single DARAM block during one processor clock cycle. SARAM can only be accesses once for read or for write during one processor clock cycle.

Different DSP versions within the C54x family contain different amounts of internal memory and support different amounts of external memory. The data memory space of the TMS320C5416 device addresses up to 64K of 16-bit words. This device uses a paged extended memory scheme in program space to allow access of up to 8192K of program memory. The device contains 64K-word´ 16-bit of on-chip dual-access RAM (DARAM) and 64K-word´ 16-bit of on-chip single-access RAM (SARAM). The DARAM is composed of eight blocks of 8K words each. Each block in the DARAM can support two reads in one cycle, or a read and a write in one cycle. Four blocks of DARAM are located in the address range 0080h-7FFFh in data space, and can be mapped into program/data space by setting the OVLY bit to one. The other four blocks of DARAM are located in the address range 18000h-1FFFFh in program space. The DARAM located in the address range 18000h-1FFFFh in program space can be

mapped into data space by setting the DROM bit to one. The SARAM is composed of eight blocks of 8K words each. Each of these eight blocks is a single-access memory. For example, an instruction word can be fetched from one SARAM block in the same cycle as a data word is written to another SARAM block. The SARAM is located in the address range 28000h-2FFFFh, and 38000h-3FFFFh in program space.

For more details of the TMS320C5416 memory map please refer to TI document SPRS095.The internal memory is always available. Certain OTX DSP boards also provide external memory. For the exact amount of memory available on your OTX product, please refer to the Technical Description of the specific OTX product.

5.2 Host Port Interface

The Host Processor communicates with the OTX on-board DSPs through a Host Port Interface (HPI). HPI is a access mechanism implemented in certain C54x DSPs which allows an easy information exchange between the host CPU and the DSP. The HPI is either an 8-bit or a 16-bit parallel interface port (depending on the DSP model) . The HPI allows both the host CPU and the DSP to share a portion of the on-chip memory. On TMS320C5416 DSPs there is 2 kWords of HPI memory available and it is located between addresses 1000h - 2000h.

The OTX Driver and the OTX SDK utilizes the on-chip HPI memory to pass control messages and user data between the host CPU and DSP.

5.3 Serial Ports

DSPs utilize serial ports to communicate with external serial devices. The TI TMS320C54x DSPs provide 3 types of serial ports:

- Standard synchronous serial ports
- Buffered serial ports
- Time-Division multiplexed (TDM) serial ports

On OTX Adapters the DSP serial ports are connected Time-Division Multiplexed (TDM) Highways. These highways transmit user data in to the DSPs for processing and out from after processing. The DSPs on OTX adapters utilize Buffered Serial Ports (BSPs). The buffered serial ports automatically transfers incoming data into the DSP memory and automatically sends outgoing data from the DSP memory. This autobuffering feature greatly reduces processor overhead for data transfer.

5.4 Program Loading

The OTX DSPs run a small kernel (operating system) which is provided with the OTX Driver. When the DSPs are booted up, the kernel is loaded to the DSP through the Host Port Interface by the host CPU. Once the kernel is running, then new application programs can be loaded through the HPI by host. Once the application program is run-

ning, the kernel is removed from the memory and the application program retains a full control of the processor.

An important feature of OTX adapters is that all the DSP code is uploaded by the host CPU and that no code is “hard-coded” in any type of persistent on-board devices. This makes the OTX adapters very maintainable as many features can be added or enhanced with simply changing the software.

6. Introduction to DSP Telecom Applications

Digital Signal Processors in telecom are used to manipulate digitized signals using mathematical algorithms. The applications supported by OTX DSPs can be roughly divided into three 3 categories: Data Receivers, Data Senders, and Data Converters. Each of these categories of DSP applications are briefly introduced in the following chapters.

6.1 Data Receiver

Data receivers are applications that receive serial data and apply an algorithm to the data. This type of DSP Data Receiver application can be thought of containing 3 interfaces: Data interface, Control Interface and Event interface (Figure 2). The receiver application receives serial data through a serial port (data interface). The receiver application is controlled by the host application through the control interface and it can send asynchronous notification towards the host through the event interface.

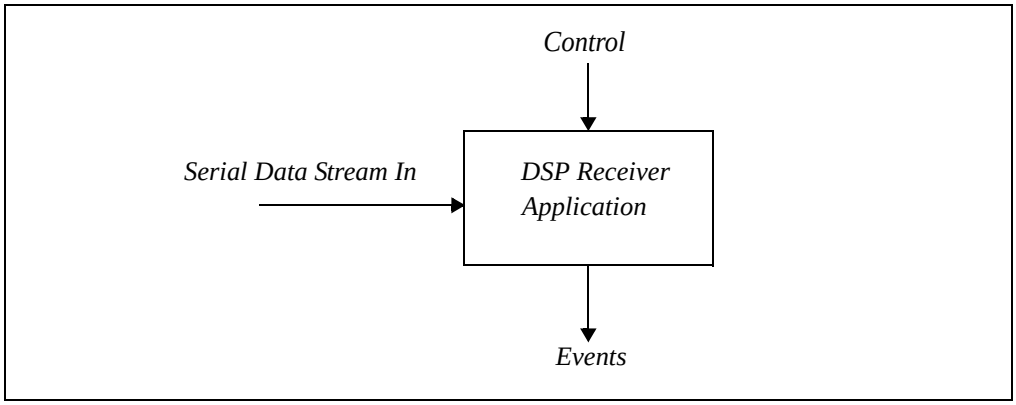


Figure 2. DSP Data Receiver Application.

Typical examples of Receiver applications include detectors, such as DTMF Detector, MF Detector, and Data Receivers, such as HDLC Receiver, Fax/Modem Receiver, etc.

6.2 Data Sender

Data Senders are applications that generate serial data using mathematical algorithms. A typical Data Sender is shown in Figure 3 illustrating how the application generates outgoing serial data based on host control.

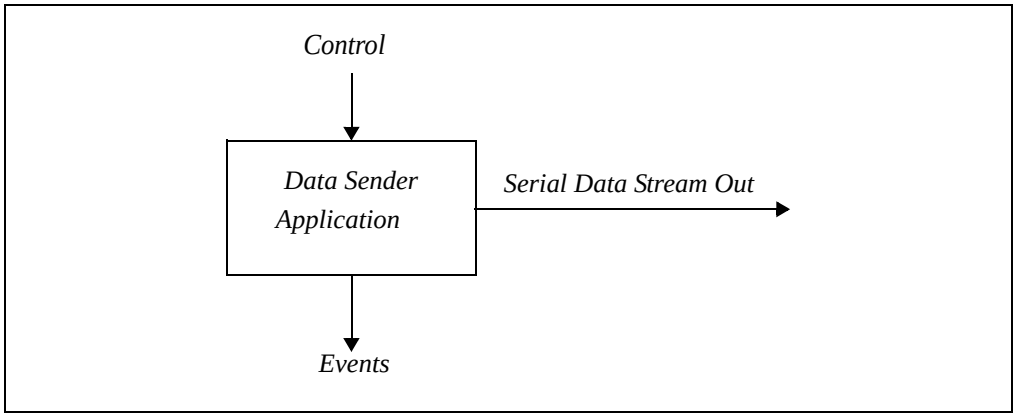


Figure 3. DSP Data Sender Application.

Examples of Data Sender applications include various Tone Generators, HDLC Senders, Modem/Fax Senders, etc.

6.3 Data Converter

Data Converters are applications that take in serial data, apply algorithms to the data, and output a serial stream of data in another format. A Data Converter application is shown in Figure 4 illustrating how serial data in is converted into another format under host control.

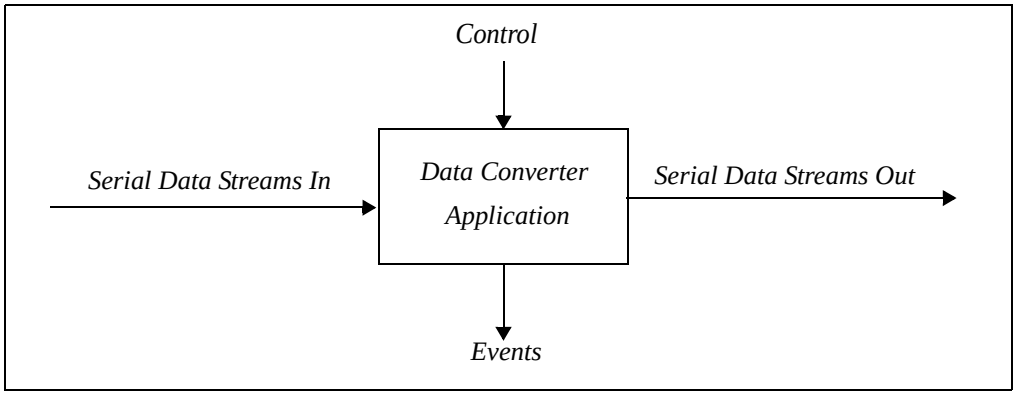


Figure 4. DSP Data Converter Application.

Examples of Data Converters include A-law to u-law converters, voice codecs, etc.

7. OTX DSP Data Architecture

This chapter describes how the serial TDM data is transferred on the OTX boards between the on-board highways and the DSP memory. As was mentioned in the previous chapter, the OTX boards utilize the buffered DSP serial ports to connect to the TDM highways. The buffered serial ports automatically transfer incoming data into the DSP memory and transmit outgoing data from the memory without almost any processor overhead. The overview of the data architecture (C5416) is illustrated in Figure 5.

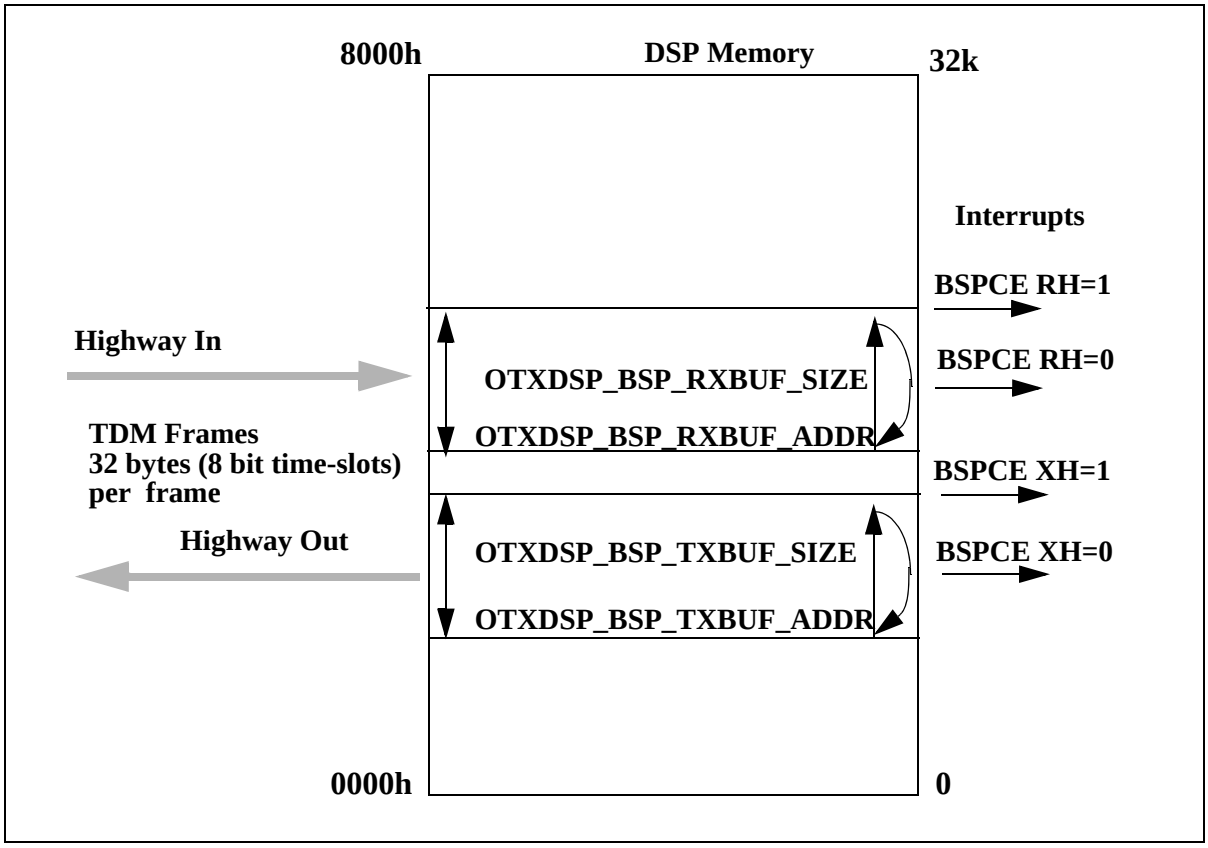


Figure 5. TMS320C5416 Data Buffers.

The buffered serial port autobuffers occupy two memory regions: Receive buffer starting at *OTXDSP_BSP_RXBUF_START* and transmit buffer *OTXDSP_BSP_TXBUF_START*. The autobuffers are circular buffers which are continuously being filled and emptied as the data from the external highway comes in (receive buffer) and goes out (transmit buffer). The sizes of the buffers are *OTXDSP_BSP_RXBUF_SIZE* and *OTXDSP_BSP_TXBUF_SIZE*, which in the SDK

demo applications are set to 800 words. This means that both buffers can contain up to 50 2.048 Mbit/s PCM frames (50 frames x 32 bytes/frame = 1600 bytes = 800 words).

The DSP generates interrupts when the buffers are half-full and when they are full. The receive interrupts are used to activate the corresponding Interrupt Service Routines (ISRs) which will then copy data from the receive autobuffer to the user's receive buffer freeing room for new data to be received. Respectively, the transmit interrupts are used to activate an ISR to fill data from the user's transmit buffer to the transmit autobuffer.

7.1 Data Formats

The user data on the highways are typically transmitted in companded 8-bit A-law or u-law formats. A-law and u-law (ITU-T G.711) are standard companding techniques which allow 13-bit (A-law) or 14-bit (u-law) samples to be represented with 8-bit values. However, most algorithms are designed to be used with linear values, thus before the companded 8-bit samples can be processed, they have to be decompanded.

Internally, C54x DSPs process data in 16-bit samples. Thus, before processing the incoming data is typically uncompanded into 16-bit linear samples. Correspondingly, the internal 16-bit samples are companded before they are transmitted out. A typical flow of data through a DSP application is illustrated in Figure 6.

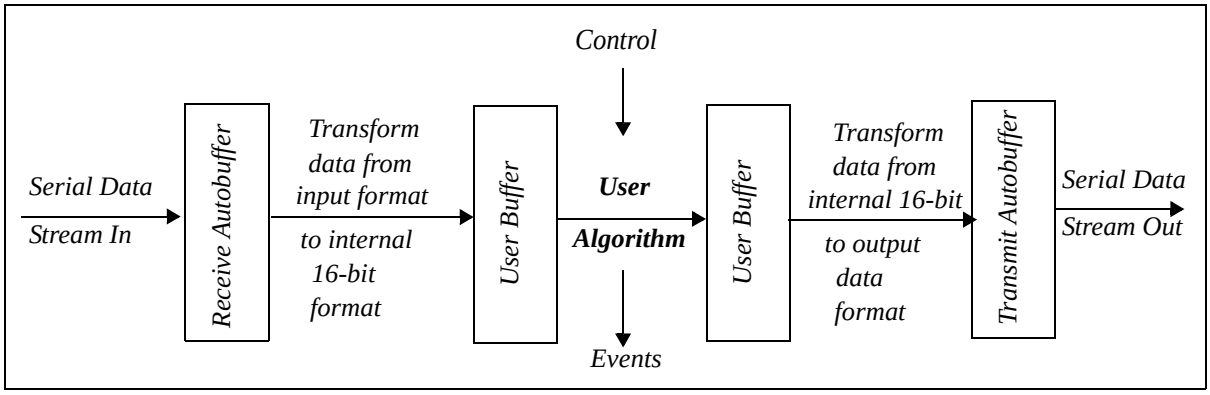


Figure 6. Typical data flow through a DSP application.

8. OTX DSP Control Architecture

When developing host and DSP applications, the user will utilize two APIs:

- the Host Driver API for host applications
- the DSP SDK API for DSP applications

8.1 OTX Host Driver API

This chapters describes host API used to write host application that communicate with DSP applications. For an in-depth discussion on the host API, please refer to *"Programmer's Guide for OTX Hardware API,"* Odin document number 1412-1-SAA-1006-1.

8.1.1 DSP Initialization

To utilize DSP resources on OTX boards, the application must first initialize the DSPs. The initialization is performed in two steps.

1. Open Physical DSP Devices

2. Load DSP software

To open a physical DSP Device, the application uses the *OtxDrvOpenPhysicalDevice()* function, as illustrated in Example 1.

```
#define DSP_SOURCE_ID 0
#define DSP_NO 0
OTX_RESULT nResult;
OTX_HANDLE hMyOTXBoard;
OTX_HANDLE hMyDsp;
OTX_HANDLE hMyEventQueue;

:

// Open a Physical DSP Device #0 on an OTX Board
// Returns handle to the DSP Device
nResult  =  OtxDrvOpenPhysicalDevice(hMyOTXBoard,  OTX_DEVICE_DSP,
DSP_NO, DSP_SOURCE_ID, hMyEventQueue, &hMyDsp);
```

Example 1. Opening of a physical DSP device.

After the DSP device has been opened and the application has attained a handle to the device, the Digital Signal Processor can now be booted up to run a program using the *OtxDspRunProgram()* function. The *OtxDspRunProgram()* function takes three parameters: A handle to the DSP device, name and path of the application file to be loaded, and label for the entry point of the application. The loading of the OTX standard SPM1 module is illustrated in Example 2.

```
// Loads and runs the standard OTX SPM1 application on
// a DSP device
nResult = OtxDspRunProgram(hMyDsp, "OtxSpm1.out", "_c_int00");
```

Example 2. Loading of DSP program.

Once the DSP program is running, it can now make a new set of logical devices available for the user. For example, the OTX Signal Processing Module 1 (OtxSpm1) program packages (which is distributed with the OTX Hardware driver) provides logical devices such as DTMF Detector and Generator, HDLC Sender and Receiver, etc. These can now accessed and controlled as any other logical devices in the system, and do not even have to be aware that these specific logical devices have been implemented with DSP software.

If a user written DSP application is loaded, then the host application can request a creation of a special logical device: *OTX_LDEVICE_USER_APPLICATION*, as shown in Example 3.

```
#define LOGIC_DEV_SOURCE_ID 1
OTX_HANDLE hMyLogicDev;

result = OtxDrvCreateLogicalDevice(
          hMyDsp,
          OTX_LDEVICE_USER_APPLICATION,
          hMyEventQueue,
          LOGIC_DEV_SOURCE_ID,
          &hMyLogicDev);
```

Example 3. Creation of a User Written DSP Logical Device.

The function *OtxDrvCreateLogicalDevice()* returns a handle to the user implemented logical device. From the host application point of view, this handle can now be used like any other handle to a logical device.

8.1.2 Host to DSP Communication

Once a user written DSP application is running, the host can now send data to the DSP application using the *OtxDspIoControl()* function. The OtxDspIoControl function is modeled after the Win32 System Services *DeviceIoControl()* function. The *OtxDspIoControl()* function prototype is shown in Example 4.

```
OTX_RESULT OtxDspIoControl(
  IN  OTX_HANDLE hDspDevice,  // Handle to the physical DSP device
  IN  OTX_UINT32 nIoControlCode,  // I/O Control Code to be passed
  IN  void *pInBuf, // Pointer to the user allocated input Buf
  IN  OTX_UINT32 nInBufSize,  // Size of the input Buffer
  IN  void *pOutBuf, // Pointer to the user allocated output Buf
  IN  OTX_UINT32 nOutBufSize, // Size of the output Buffer
  OUT OTX_UINT32 *pnBytesReturned,   // Number of Bytes returned
  IN  OTX_TASK_REF nTaskRef,  // Task Ref for non-blocking calls
  IN  OTX_TIME nWaitMaxMs     // Max blocking wait time
);
```

Example 4. *OtxDspIoControl()* function prototype.

The *nIoControlCode* parameter in the *OtxDspIoControl()* function are used to indicate the action to be performed by the DSP application. Some of the codes reserved for standard actions performed automatically by the host driver. It is the responsibility of the DSP application developer to ensure that the used codes are coordinated between the host application and the DSP application.

The *pInBuf* parameter in *OtxDspIoControl()* can be used to pass arbitrary data to the DSP application. If the DSP application needs to return data to the host application, then the data will be copied to the user allocated *pOutBuf*. Also, *pnBytesReturned* will return the size of the returned data.

The *OtxDspIoControl()* function can be called in two modes:

- **Blocking**: The driver will wait for the DSP software to complete the whole operation before returning control back to the calling program. A time-out value can be provided to the function to force a return upon timeout expiry and to prohibit a potential dead-lock situation.
- **Non-Blocking:** The driver function returns immediately. If the function was able to complete operation without waiting, it will return the result of the operation. Otherwise the function will return *OTX_S_PENDING* and schedule a task to be completed when the needed data becomes available. The application can now continue its operations while the requested task will be completed in the background. Once the task has been completed, a driver event will be provided to the application through the appropriate event queue. The event will contain a reference value that was provided in the non-blocking function call.

The *OtxDspIoControl()* function call can be made blocking by providing *OTX_TASK_SYNC* as the task reference. An example of a blocking function call with a 5 second time-out and without a time-out are shown in Example 6.

```
OTX_RESULT nResult;
OTX_HANDLE hMyDsp;

// Blocking call with 5 second time-out
nResult = OtxDrvIoControl(hMyDsp, ..., OTX_TASK_SYNC, 5000);

// Blocking call without time-out
nResult =
OtxDrvIoControl(hMyDsp, ..., OTX_TASK_SYNC, OTX_TIME_INFINITY);
```

Example 5. Examples of Blocking calls to OtxDspIoControl( ).

A positive (>= 1) task reference value will cause a non-blocking function call and a creation of a background task. Once the task has completed or if the time-out has expired, a driver event will be provided to the user. The driver event contains the task reference value as given to the function. This allows the application to match the events to the tasks. An example of a non-blocking function call is shown in Example 6.

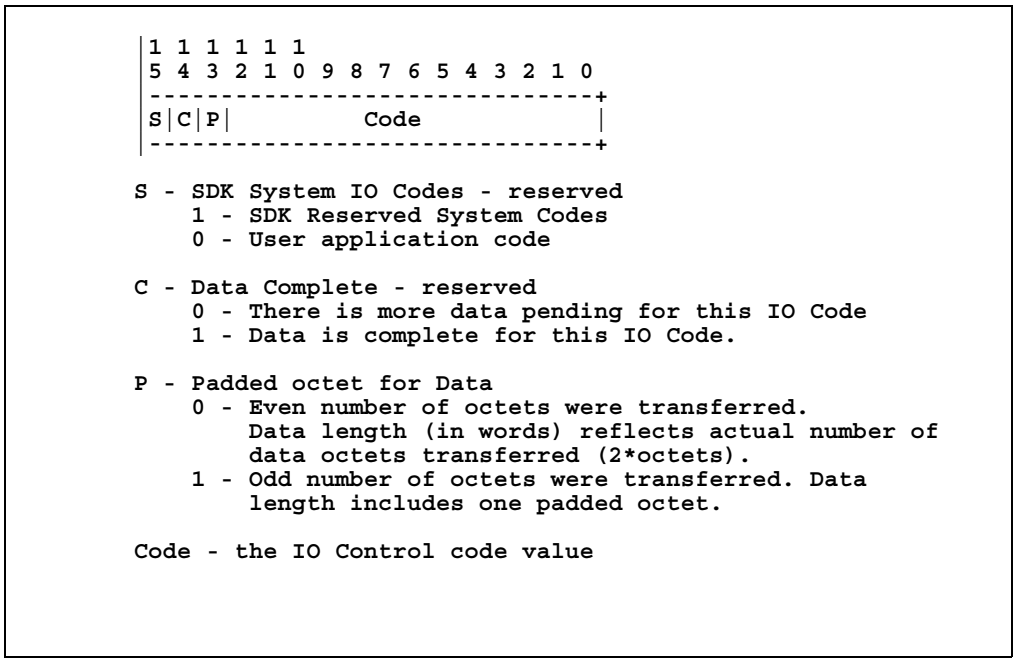
```
#define MY_TASK_REF
OTX_RESULT nResult;
OTX_HANDLE hMyDsp;

// Non-Blocking Function call ...
nResult = OtxDspIoControl(hMyDsp, ..., MY_TASK_REF, 5000);

if (nResult == OTX_S_PENDING) {
   // Task created, execution continues immediately
} else {
   // Function call completed, Success or failure
}
```

Example 6. An example of a non-blocking function call *OtxDspIoControl( )*.

If the user application does not care to keep track of what tasks have been completed, the application can use a special *OTX_TASK_ASYNC* task reference, or alternatively the same positive task reference can be used over and over again.

8.1.2.1 I/O - Control Codes

The I/O-Control codes used with the *OtxDspIoControl()* function are 16-bit codes with the 3 most significant bits reserved for system use (See Example 7).



```
|1 1 1 1 1 1
|5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
|-------------------------------+
|S|C|P|         Code            |
|-------------------------------+

S - SDK System IO Codes - reserved
    1 - SDK Reserved System Codes
    0 - User application code

C - Data Complete - reserved
    0 - There is more data pending for this IO Code
    1 - Data is complete for this IO Code.

P - Padded octet for Data
    0 - Even number of octets were transferred.
        Data length (in words) reflects actual number of
        data octets transferred (2*octets).
    1 - Odd number of octets were transferred. Data
        length includes one padded octet.

Code - the IO Control code value
```

Example 7. I/O Control Code Structure.

The user codes must have the most significant bit set to ‘0’. Certain I/O-control codes are reserved for use with standard API functions. The reserved codes are listed in Table 5.

TABLE 5. Reserved I/O Control Codes.

Code Macro	Value	Meaning
OTXDSP_SDK_IO_NO_COMMAND	0x8000	Reserved. Not to be used an IO Control Code
OTXDSP_SDK_IO_CREATE	0x8001	Issued by the driver when the standard *OtxDrvCreateLogicalDevice()* API function is called.
OTXDSP_SDK_IO_ENABLE	0x8002	Issued by the driver when the standard *OtxDrvEnable()* API function is called.
OTXDSP_SDK_IO_DISABLE	0x8003	Issued by the driver when the standard *OtxDrvDisable()* API function is called.
OTXDSP_SDK_IO_CONNECT	0x8004	Issued by the driver when the standard *OtxDrvConnectLogicalDevice()* API function is called. Data passed is struct *OtxDspSdkIoInConnectS*
OTXDSP_SDK_IO_DISCONNECT	0x8005	Issued by the driver when the standard *OtxDrvDisconnectLogicalDevice()* API function is called. Data passed is struct *OtxDspSdkIoInDisconnectS*
OTXDSP_SDK_IO_RESET	0x8006	Issued by the driver when the standard *OtxDrvReset()* API function is called.

8.1.3 DSP to Host Communication

The DSP to host communication is handled through asynchronous driver events and through Event Queues. When the DSP application wants to signal the host application, it can place a driver event to the Event Queue associated with the physical DSP device. The host application can wait for these events using the *OtxDrvWaitForSingleEvent()* function, which will place the host thread to sleep until the driver has a notification event available. The events generated by the DSP application can be retrieved from the event queue with function *OtxDrvGetEventData()*, as demonstrated in Example 8.

```
OtxEventDataS eventData;
 :

// Wait for a driver event
OtxDrvWaitForSingleEvent(hMyNotificationEvent,OTX_TIME_INFINITY);

// Check for driver events from previously created event queue
nResult = OtxDrvGetEventData(hMyEventQueue, &eventData);

do {  // Loop to retrieve all the events
 result = OtxDrvGetEventData(hMyEventQueue, &eventData);

  if (result == OTX_S_OK) {
    // We have a driver event
     ...
  }
} while (result == OTX_S_OK);
```

Example 8. Retrieval of driver events from an event queue.

If the event has any data to be passed from the DSP to the Host associated with it, the data can be retrieved with the *OtxDspReadData()* function. The *OtxDspReadData()* function can be used to retrieve data associated with both spontaneously generated events and for events generated upon completion of non-blocking function calls. The *OtxDspReadData()* function prototype is shown in Example 9.

```
OTX_RESULT OtxDspReadData(
  IN  OTX_HANDLE hDspDevice,  // Handle to the physical DSP device
  IN  OTX_TASK_REF nTaskRef, // Reference to associated task
  IN  void *pOutBuf, // Pointer to the user allocated output buf
  IN  OTX_UINT32 nOutBufSize, // Size of the output buffer
  OUT OTX_UINT32 *pnBytesReturned // Number of Bytes returned
  OUT OTX_UINT32 *pnIoControlCode, // I/O Control Code the
                        // returned data is associated with
);
```

Example 9. *OtxDspReadData()* function prototype.

8.2 DSP SDK API

8.2.1 DSP Initialization

The DSP initialization within the DSP application involves initializing the processor, setting up interrupt vectors, initializing the serial ports, and the host port interface. The source code for performing this initialization is provided with the SKD and can be used “as is” by most DSP applications.

8.2.2 Host to DSP Communication

Once the host application has sent a command request using the *OtxDspIoControl( )* API function, the OTX Host driver will copy the command code and the data in to the DSP Host Port Interface (HPI) memory. The data is packed into the HPI memory starting from address *OTXDSP_SDK_IOCTL_CODE_ADDR* as specified in Figure 7. Only one I/O-control packet is present at the HPI memory at the time. Other pending packets are queued by the Host driver.

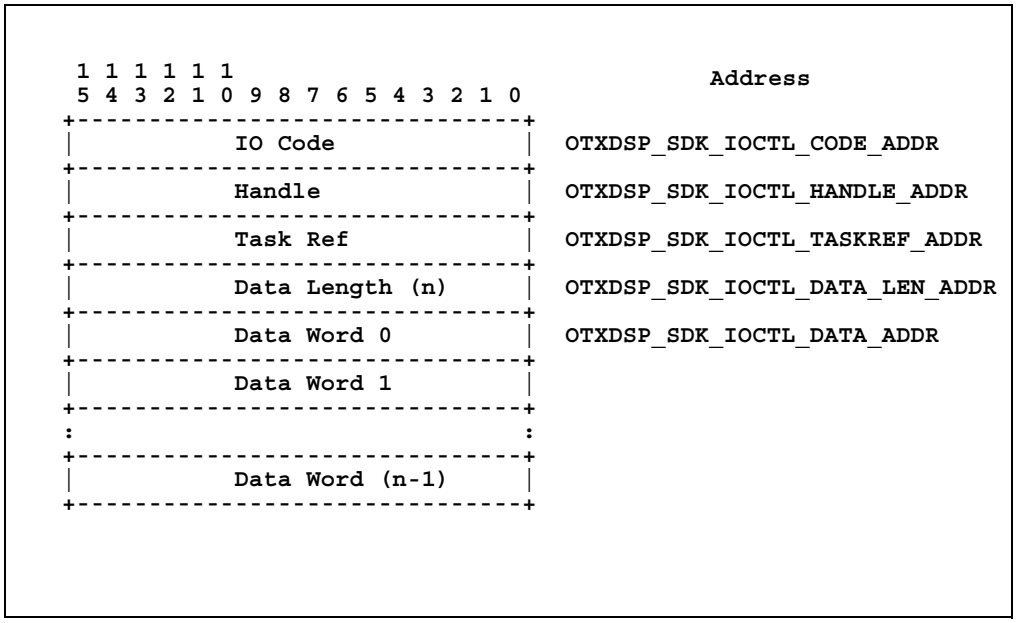


Figure 7. Host to DSP Data Packing in the HPI memory.

Once the host driver has copied a new I/O-control packet into the DSP HPI memory, the Host driver will interrupt the DSP and the DSP program execution will resume at the *OtxDspServiceIoControl( )* call-back function. The *OtxDspServiceIoControl( )* call-back function can be re-implemented by the user to perform the application specific operations (Example 10).

```
void OtxDspServiceIoControl(
/*-------------------------------------------*\
 Call-back routine called by the ISR when the host has issued
 a command to the HPI and requests our attention.

 TO BE RE-IMPLEMENTED BY THE USER.
\*-------------------------------------------*/
  void
)
{
  /* Add your code here .... */
}
```

Example 10. *OtxDspServiceIoControl( )* call-back function.

8.2.3 DSP to Host Communication

The DSP to host communication is handled through asynchronous driver events. The DSP application can send events and associated data to the Host by calling the *OtxDspSetHostEvent()* function. The prototype of the *OtxDspSetHostEvent()* function of is shown in Example 11.

```
short OtxDspSetHostEvent(
  unsigned short nEvent,   /* Event to be send to the host */
  unsigned short hHandle,  /* Handle to the user logical
                           .. device sending the event */
  unsigned short nTaskRef, /* Associated Task Reference */
  unsigned short *pData,   /* Pointer to the data to be sent to
                           .. the host */
  unsigned short nDataLen  /* Length of the data to be sent */
);
```

Example 11. *OtxDspSetHostEvent( )* DSP API function.

The *OtxDspSetHostEvent()* signals an event towards the Host and copied the data to be sent into a circular Event buffer in the HPI memory, as specified in Figure 8.

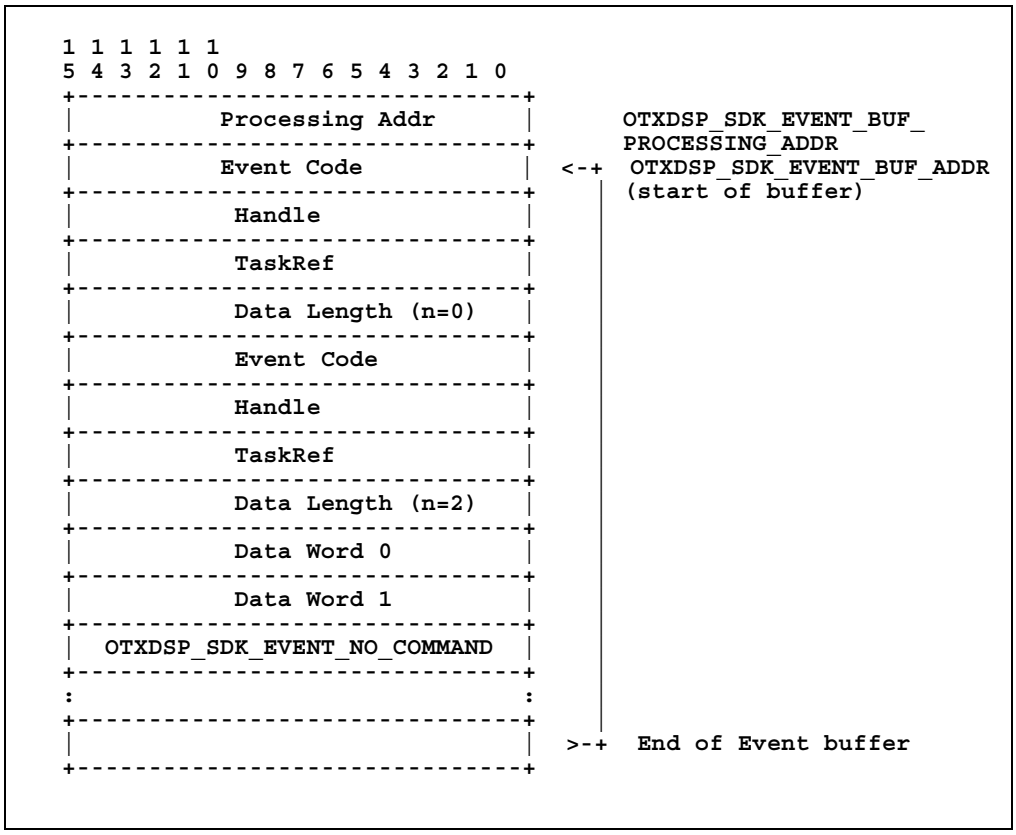


Figure 8. DSP to Host Event and Data Buffer in the HPI memory.

8.2.4 Data Access

As was discussed in Chapter 7: "OTX DSP Data Architecture", the incoming and outgoing data is stored in circular Auto Buffers which are mapped into the DSP memory. The DSP generates an interrupt when the buffers are half-full and full (Receive buffer) or half-empty and empty (transmit buffer). These interrupts will in-turn call user definable call-back functions *OtxDspServiceRxSerialPort( )* and *OtxDspServiceTxSerialPort( )*. The application developer can use these call-back functions to copy the data between the auto buffers and user buffers, or if the performance allows, even process the data directly in these auto buffers. The call-back function for Serial Port Receive Interrupts and for Serial Port Transmit Interrupts are shown in Example 12 and in Example 13.

```
void OtxDspServiceRxSerialPort(
/*-----------------------------------------------*\
 Call-back routine called when we have receive interrupt from the
 Buffered Serial Port indicating the receive buffer is either
 full or half-full.

  TO BE RE-IMPLEMENTED BY THE USER.
\*-----------------------------------------------*/
  void
)
{
  /* Add your code here .... */
}
```

Example 12. *OtxDspServiceRxSerialPort( )* call-back function.

```
void OtxDspServiceTxSerialPort(
/*-----------------------------------------------*\
 Call-back routine called when we have transmit interrupt from the
 Buffered Serial Port indicating the transmit buffer is either
 empty or half-empty.

  TO BE RE-IMPLEMENTED BY THE USER.
\*-----------------------------------------------*/
  void
)
{
  /* Add your code here .... */
}
```

Example 13. *OtxDspServiceTxSerialPort( )* call-back function.

9. SDK API

9.1 Api Functions

The DSP SDK API provides an implementation of the functions needed to communicate with the Host Driver. In addition, the SDK provides the implementation of certain useful utility functions, such as A-law and U-law companding and expanding. The API functions are listed in .

.

TABLE 6. DSP SDK API functions available to the application developer.

Function Name	Function Description
OtxDspInitProcessor( )	Sets up the processor to the desired operating mode
OtxDspInitInterrupts( )	Initialize and unmask specified interrupts
OtxDspInitHostPortInterface( )	Initialize the Host Port Interface

TABLE 6. DSP SDK API functions available to the application developer.

Function Name	Function Description
OtxDspInitBufferedSerialPorts( )	Initialize the Buffered Serial Ports (BSP)
OtxDspSetHostEvent( )	Generates a driver event for the host and copies the associated data to the Host through the HPI buffer.
OtxDspCommandCompleteRc( )	Generates a Command Complete driver event for the host and passes the return code from a IO Control call back to the host.
OtxDspCommandComplete( )	Generates a Command Complete driver event for the host.
OtxDspAlawCompress( )	Takes in a 14 bit 2-s complement value and returns a A-law compressed value ready to transmission (8-bits)
OtxDspULawCompress( )	Takes in a 14 bit 2-s complement value and returns a U-law compressed value ready to transmission (8-bits)
OtxDspLawExpand( )	Expands a U-law or A-law companded 8-bit sample and returns a 14-bit 2's complement value (in the 14 least significant bits of a 16-bit word)

9.2 Call-back functions

The DSP SDK API defines certain call-back functions that need to implemented by the application developer. The call-back functions are called from the DSP Interrupt Service routine and are listed in Table 7. The demo application provide good example implementations for these call-back functions.

TABLE 7. Call-back functions to be re-implemented by the application developer.

Function Name	Function Description
OtxDspServiceIoControl( )	Call-back routine called by the Interrupt Service Routine when the host has issued a command to the HPI and requests our attention.
OtxDspServiceRxSerialPort( )	Call-back routine call when we have receive interrupt from the Buffered Serial Port indicating the receive buffer is either full or half-full.
OtxDspServiceTxSerialPort( )	Call-back routine call when we have transmit interrupt from the Buffered Serial Port indicating the transmit buffer is either empty or half-empty.

10. Demo Applications

The C54x DSP SDK is provided with a full source code for two working demo applications (DspDemo1 and DspDemo2). The source code is provided for the DSP application programs (Dsp\DspDemo1\ and Dsp\DspDemo2\) as well as the source code for the corresponding Host applications (Demos\DslSdk\Pci\DspDemo1\ and Demos\DspSdk\Pci\DspDemo2\).

10.1 Getting Started

It is recommended that you will use an incremental approach to your DSP application development; i.e., start from the Demo applications and make small changed at a time and always verify that the changes work. The steps to get started in DSP application development are:

1. Verify that you can compile and link the demo applications using the TI C compiler.

2. Verify that you can use the corresponding Host application to load the DSP demo application and that the DSPs start running (the DSPs are running when the heart-beat LEDs are blinking).

3. Verify that the Host Demo application operates with the DSP demo application as documented. Use the Demo applications simple key-stroke interface to verify the operation.

4. Verify that you can debug (set breakpoints, step) the demo applications using the TI Code Composer debugger.

5. Begin experimenting by making small changes to the demo applications and always verifying that your changes work.

10.2 Included Files

10.2.1 DSP SDK Files

The C54x DSP SDK contains the files listed in Table 8.

TABLE 8. DSP SDK Common Files.

File Name	File Description
DspIoctl.h	Host <-> DSP Interface Specification header file.
OtxDef.h	Macro, Constant, and Type Definitions for the DSP SDK

TABLE 8. DSP SDK Common Files.

File Name	File Description
OtxApi.h	Macro, Constant, and Type Definitions for the DSP SDK
OtxApi.c	API function implementations.
OtxVecs.asm	Interrupt Vectors (Assembly)
OtxLaws.asm	Implementation A-Law, u-law expanding and companding routines in Assembly.

10.2.2 DSP Demo Application Files

The files listed in Table 8 implement the DSP SDK. These files should not be edited by the application developer. In addition to the standard files, the SDK also includes Demo application files listed in Table 9 and Table 10.

TABLE 9. Files specific for Demo Application 1

File Name	File Description
DspDemo1.c	Demo 1. Skeleton for DSP applications. Only implements blinking of the heart-beat LED and a simple command interface for controlling the blinking.
Demo1Io.h	IOCTL Codes used by DspDemo1
build.bat	Batch file for building the demo application

TABLE 10. Files specific for Demo Application 2

File Name	File Description
DspDemo2.c	Demo 2. Skeleton for DSP applications. Interfaces with system highways implementing an adjustable gain control.
Demo2Io.h	IOCTL Codes used by DspDemo2
build.bat	Batch file for building the demo application

10.2.3 Host Demo Application Files

The SDK also includes two demo host applications to be used with the DSP demo applications. The DSP Host Applications can be used with any OTX PCI boards that contain TI TMS320C54x DSP resources. The host application files included are listed in Table 11.

TABLE 11. Files for Host Demo Applications

File Name	File Description
DspDemo1.c	Host demo application to be used with the DSP demo application DspDemo1.out
DspDemo2.c	Host demo application to be used with the DSP demo application DspDemo2.out
DspDemo1.dsp	Visual C++ Project file for making DspDemo1.exe
DspDemo2.dsp	Visual C++ Project file for making DspDemo2.exe

10.3 Compiling and Linking

10.3.1 DSP Demo Applications

To compile and link the demo application, you can use the *build.bat* batch file provided with the SDK. If you wish invoke the C Compiler manually, *build.bat* contains the commands for compiling and linking of the files.

The linking process is controlled with a linker command file. The SDK provides an example linker command files for both demo applications (*DspDemo1.cmd* and *DspDemo2.cmd)*.

For more information on compiling and linking, please refer to the TMS320C54x Optimizing C Complier User Guide.

10.3.2 Host Demo Applications

The source directory of each host demo applications contain subdirectories for each supported platform, e.g. Win32 for Windows. A Microsoft Developer Studio 6.0 Project File (*.dsp) or a Makefile is located in each platform specific subdirectory. To compile the demo applications with Visual C++, insert the project file into a new or existing Workspace and add a path the OtxDrv.Lib (\Lib) and the DRV SDK header files (\Inc) to the project settings in the settings for this project.

10.4 Demo Program Flow

The DSP demo application follow the program flow shown in Figure 9.

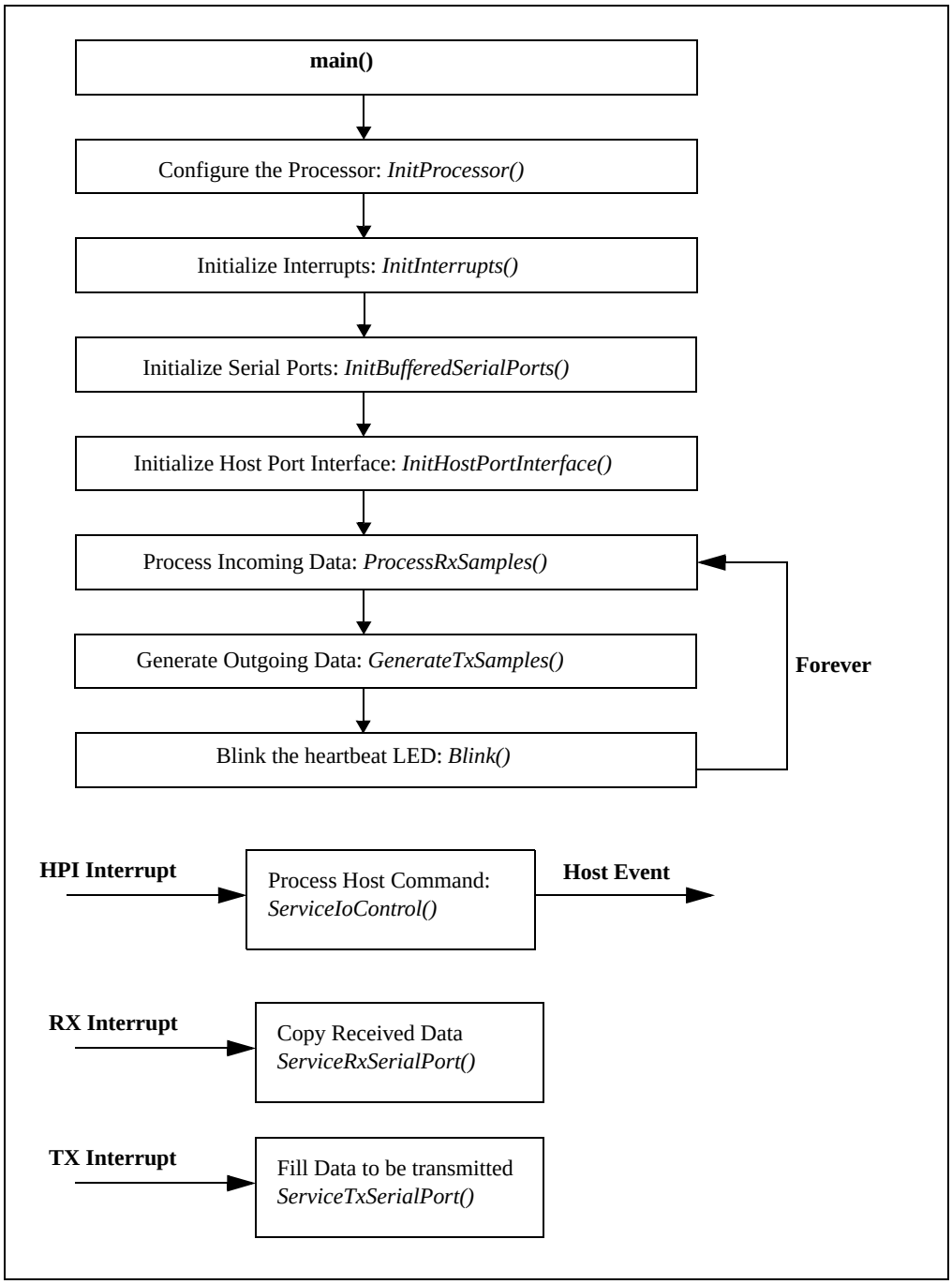


Figure 9. Program flow for SDK Demo Applications.

11. Debugging

Once the user application has been compiled and linked, it can be downloaded to the DSPs on OTX boards and executed or debugged with the TI Code Composer Debugger. For more information on how to use the Code Composer Debugger, please refer to the Code Composer User Guide.

To connect the Code Composer debugger to the OTX PCI Adapter boards, you need to use the Hermod-JTAG Code Composer Debug Probe, Odin product number HMA-1057-1. Connect Hermod-JTAG to the BJ3 connector on the OTX PCI adapter. The Code Composer emulator pod can then be connected to Hermod-JTAG port labeled "*emulator*".

Doc. No. 1412-1-SAA-1007-1
For more information on this product, please contact:

Odin TeleSystems Inc.
800 East Campbell Road, Suite 334
Richardson, Texas 75081-1873
U. S. A.

Tel: +1-972-664-0100
Fax: +1-972-664-0855
Email: Info@OdinTS.com
URL: http://www.OdinTS.com/