Application Report
SPRA895 – February 2003

DSP/BIOS, RTDX and Host-Target Communications

Harish Thampi S / Jagan Govindarajan *Software Development Systems*

ABSTRACT

Real-Time Data Exchange (RTDX™) provides real-time, continuous visibility into the way target applications operate in the real world. RTDX allows system developers to transfer data between target devices and a host without interfering with the target application. This application note describes how DSP/BIOS™ uses RTDX technology for real-time host-target communication. Also covered is the DSP/BIOS HST module with and without the streaming I/O. Included is an example for streaming data transfer between a host and target for the TMS320C6711 DSK. It is assumed that the reader has some exposure to DSP/BIOS.

Contents

List of Figures

1 Introduction

Real Time Data Exchange (RTDX) is a technology that allows users to transfer data between a host and target devices without interfering with the target application. This helps the user to get a realistic view of how the system will work. RTDX consists of both target and host components; an RTDX software library runs on the target application and the target application makes function calls to this library's API in order to pass data to or from it. The RTDX software library also transfers data to/from the host in the background while the target application is running.

On the host platform, an RTDX host library operates in conjunction with Code Composer Studio™. Displays and analysis tools communicate with RTDX via an easy-to-use COM API to obtain the target data and/or to send data to the target application. Designers may use their choice of standard software packages to retrieve, analyze, and/or display data such as Microsoft's Visual C++, Visual basic, and even Excel. The RTDX technology is explained in detailed in section 2.1.

DSP/BIOS is a scalable real-time kernel that supports real-time scheduling, synchronization, and real-time instrumentation. DSP/BIOS provides preemptive multi-threading, hardware abstraction, and real-time analysis. DSP/BIOS makes use of RTDX to transfer data for its real-time analysis tools and allows the user to insert and configure RTDX by providing an RTDX interface in its configuration tool. It also provides two other interfaces called the host channel (HST) and the host link driver (DHL) for communication between the host and target. These interfaces internally use RTDX to communicate with the host. This application note will focus on how DSP/BIOS uses RTDX for host-target communication, and includes with an example that works on TMS320C6711 DSK.

2 DSP/BIOS Real-Time Analysis with RTDX

2.1 RTDX – The Technology

RTDX enables real-time, continuous visibility into the way target applications operate in the real world. RTDX allows system developers to transfer data between a host and the target devices without interfering with the target application. Figure 1 shows how RTDX works. This section talks about how the RTDX data transfer is done from the target to host and vice versa.

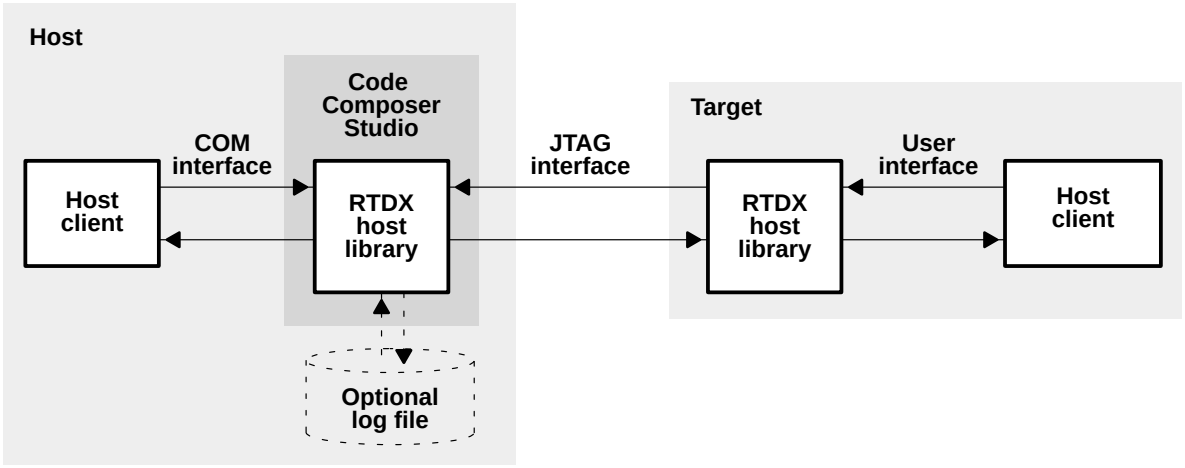

Figure 1. RTDX Block Diagram

In target-to-host communication, an output channel should be configured. Data is written to the output channel using the routines defined in the RTDX user interface. This data is immediately recorded into a target buffer defined in the RTDX target library. The data from this buffer is then sent to the host through the JTAG interface. The RTDX host library receives this data from the JTAG interface and records it into either a memory buffer or an RTDX log file. The transfer of data through JTAG from target to host is done without halting the target's processor. This data recorded by the host can then be collected by the host application and displayed in a meaningful way. On the Microsoft Windows™ operating system, the host interface is provided as a COM interface.

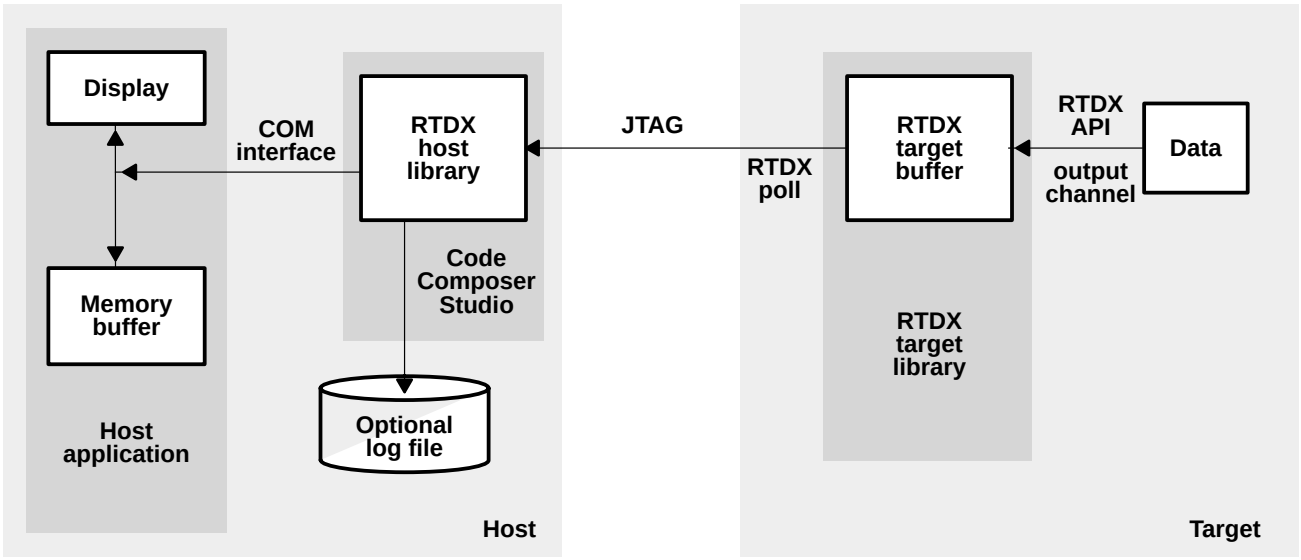

Figure 2. Target-to-Host Communication

For the target to receive data from the host, you must declare an input channel on the target side. The target requests data from the input channel using routines defined in the user interface. This request is recorded into the target buffer and then sent to the host through the JTAG interface. All the data to be sent to the target is written to a memory buffer inside the RTDX host library through a COM interface. When the RTDX host library receives a read request from the target application, the data in the host buffer is sent to the target through the JTAG interface. The data is then written to the requested location on the target in real time. The host then notifies the RTDX target library when the operation is complete.

For more information on RTDX or examples of its use, see the RTDX reference in the online help of Code Composer Studio.

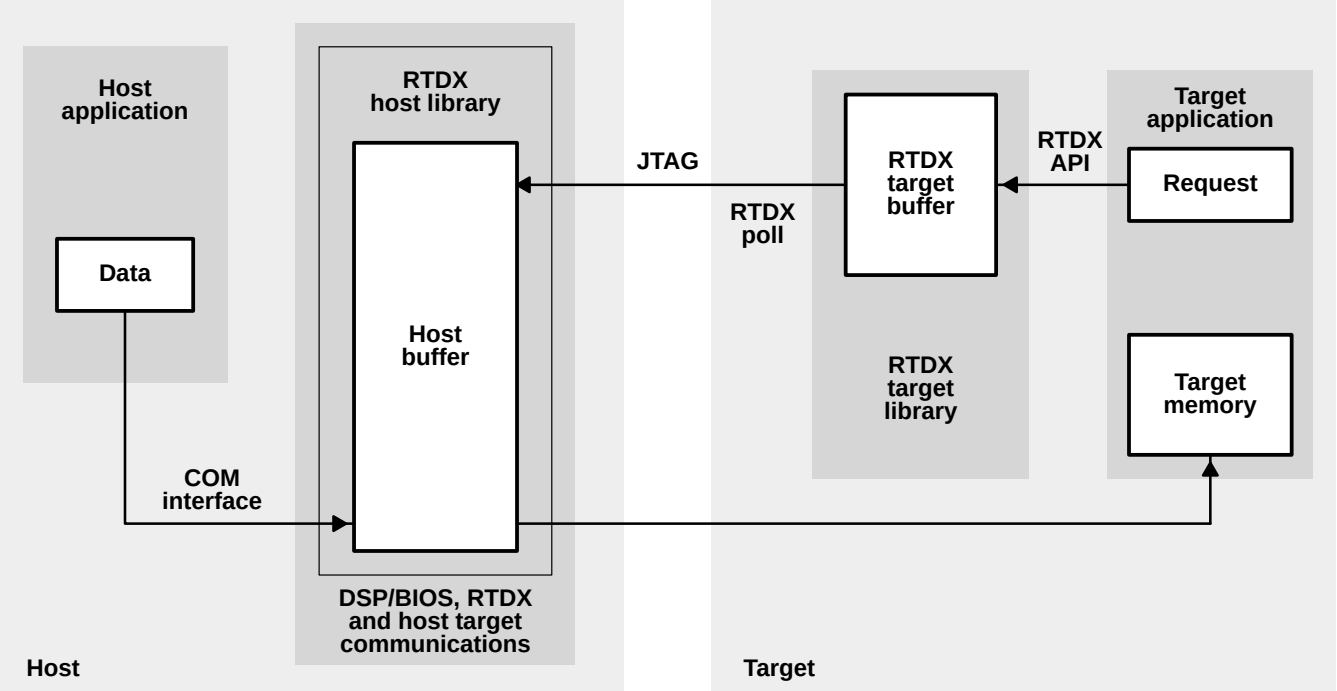

Figure 3. Host-to-Target Communication

2.2 RTDX Without DSP/BIOS

For a non-DSP/BIOS project, the user must manually configure the interrupt vectors in the target application to support RTDX. The linker command file of the project should allocate the ".rtdx_text" and ".rtdx_data" sections into the available memory area. The ".rtdx_data" section contains the buffer where the data is stored temporarily before it is sent to the host.

The user must also define the variable "_RTDX_interrupt_mask" in the linker command file to designate to RTDX which interrupts must be disabled before entering an RTDX critical section. Upon entering a critical section, RTDX will apply this mask to the interrupt enable/mask register to temporarily disable the indicated interrupts. The interrupt control register is restored upon leaving the RTDX critical section. The algorithm is similar to the following:

```
cs_enter:
        oldGIE = CSR.GIE
        CSR.GIE = 0             // next operations must be atomic
        oldIER = IER;
        IER = IER & RTDX_interrupt_mask
        CSR.GIE = oldGIE
cs_exit:
        IER = oldIER
```

Any interrupt handlers that call RTDX_read/write functions should be added to the mask to prevent corruption of the RTDX global data structures by simultaneous access from multiple RTDX clients. This setting permits users to disable only those interrupt, which may result in an RTDX function call. Other interrupts in the system will be unaffected and will be permitted to fire from within the critical section. Any ISRs that do not call an RTDX routine need not be added to this mask. An example of linker command file is shipped with Code Composer Studio and can be found in the following directory \ti\examples\<target>\rtdx\shared.

The implementation of RTDX differs for each supported ISA and the type of RTDX in use (i.e., JTAG, SIM, HS). In all cases, the RTDX_poll function maintains communication between the host and the target. In interrupt-driven implementations, this function is called within an ISR used by RTDX. In polling-driven implementations, it must be called regularly by the target application. The macro RTDX_POLLING_IMPLEMENTATION identifies whether this implementation of RTDX is interrupt-driven or polling. The macro is defined as 0 (zero) if an interrupt driven form of RTDX is used. If the macro is defined as 1 (one), it indicates a polling implementation and it will be necessary to explicitly call RTDX_Poll from your target application at regular intervals to maintain communication with the host. Failure to call RTDX_Poll may cause the debugger to hang. Do NOT explicitly call RTDX_Poll from your application if the RTDX_POLLING_IMPLEMENTATION macro is defined as a 0 (zero). Details specific to each supported ISA for the JTAG implementations of RTDX are given in the following sections.

2.2.1 TMS320C5400 Target Variations

In TMS320C5400 applications, a polling implementation is used to maintain the communication between the target and the host. The target application must call RTDX_Poll regularly and often for RTDX to maintain communication with the host. Each call to RTDX_Poll may result in an analysis trap, causing the CPU to jump to the interrupt vector ATRAP_V (analysis trap, offset 03Ch. This interrupt is reserved for use with real-time RTDX monitor interrupts and must call the handler “ATRAP_H” within the RTDX target library. An example of how this vector should be defined can be found in the file “intvecs.asm”, in the directory “\ti\examples\<target>\rtdx\shared” where “<target>” is dsk5402 for the C5402 DSK and dsk5416 for the C5416 DSK.

2.2.2 TMS320C5500 Target Variations

On the TMS320C5500 target, the interrupt vector DLOGIV (IV25) executes the interrupt service routine “datalog_isr” for handling RTDX. This interrupt is generated by the target on DMA completion for target to host transfer. The RTDX implementation is DMA based and so only one interrupt is generated on completion of the transfer. Example of how this vector is handled can be found in the file intvecs.asm. This file can be found in the directory \ti\examples\evm5510\rtdx\shared.

2.2.3 TMS320C6000 Target Variations

For TMS320C6000 applications, the interrupt vector MSGINT (offset 060h) must be defined to point to the RTDX interrupt handler RTEMU_msg, which is a special interrupt entry to save and restore context around the RTDX_Poll function. The implementation of RTDX for C6000 is interrupt-driven and there is roughly one interrupt from the host for each word of transfer. This interrupt is reserved for use with real-time monitor interrupts. An example of how this vector should be defined can be found in the file intvecs.asm. This file can be found in the directory \ti\examples\<target>\rtdx\shared, where “<target>” is dsk6211 for the C6211 DSK, dsk6711 for the C6711 DSK and evm6201 for the C6201 EVM.

2.3 DSP/BIOS Support for RTDX

RTDX can be configured using the DSP/BIOS configuration tool. DSP/BIOS exposes the underlying JTAG RTDX channels for data transfer that allows DSP/BIOS users to specify and launch application-specific threads or functions in response to data transfer events. DSP/BIOS built-in instrumentation uses this to paint the analysis plug-ins in real time. The advantage of using DSP/BIOS is that the following details are handled automatically.

- Allocation and location of the .rtdx_text and .rtdx_data sections
- Defining the _RTDX_interrupt_mask variable.
- Inserting the appropriate ISRs required for RTDX in the interrupt vector table

In the configuration file (*.cdb) some interrupts are reserved for RTDX. For C6000 targets, two interrupts (HWI_INT3, HWI_RESERVED1) are reserved for RTDX. For C5400 targets, the HWI_SINT29 and HWI_SINT30 are reserved for RTDX. For C5500, HWI_DLOG is reserved for RTDX. The user cannot modify reserved interrupts. Section 3.3 explains this in more detail.

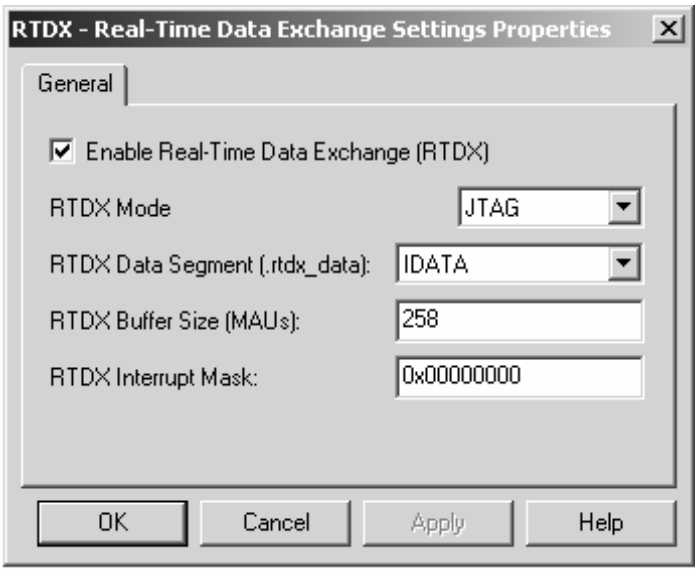

Figure 4. RTDX Manager Properties and RTDX Object Properties

Figure 4 shows the RTDX setting properties dialog. The “Enable Real-Time Data Exchange (RTDX)” check box should be checked to link RTDX support into the application. By default JTAG is selected as RTDX mode for most of the targets. If you are using the simulator, change the mode to “Simulator.” If the target has support for High-Speed-RTDX, then you can set this mode to HS-RTDX for high-speed communication between the target and host. For more information on setting RTDX mode to HS-RTDX, refer to *How to Use High-Speed RTDX Effectively* (SPRA821).

The RTDX data segment (.rtdx_data) is used for buffering target-to-host data transfers. The RTDX message buffer and state variables are placed in this segment. RTDX buffer size, in minimal addressable units (MAUs), is the size of the RTDX target-to-host message buffer. When RTDX_write is performed the data is temporarily stored in this buffer before sending it to the host. HST channels using RTDX are limited to this size.

The RTDX interrupt mask identifies the ISRs that call the RTDX APIs and protects RTDX critical sections. This permits users to only disable those interrupts that can make an RTDX function call. This defines the _RTDX_interrupt_mask in the linker command file for the DSP/BIOS application. See section 2.2 for more information on this setting.

By default, the RTDX_interrupt_mask is set to 0. Within DSP/BIOS, this is the safest setting for RTDX as DSP/BIOS is a preemptive kernel. While an interrupt itself may not directly call an RTDX function, it may unblock another thread that will call an RTDX function that does not run within the context of the ISR. In this case, the RTDX critical section handling alone is not sufficient for protecting the RTDX global data structures. If the user can guarantee that a high-priority interrupt will neither directly call an RTDX function itself, nor indirectly as a result of a task swap to a thread which calls an RTDX function, then the user may unmask the corresponding bit for that interrupt in the RTDX_interrupt_mask. Users that call RTDX functions directly from within a DSP/BIOS application must ensure mutual exclusion of RTDX activity or data corruption may result.

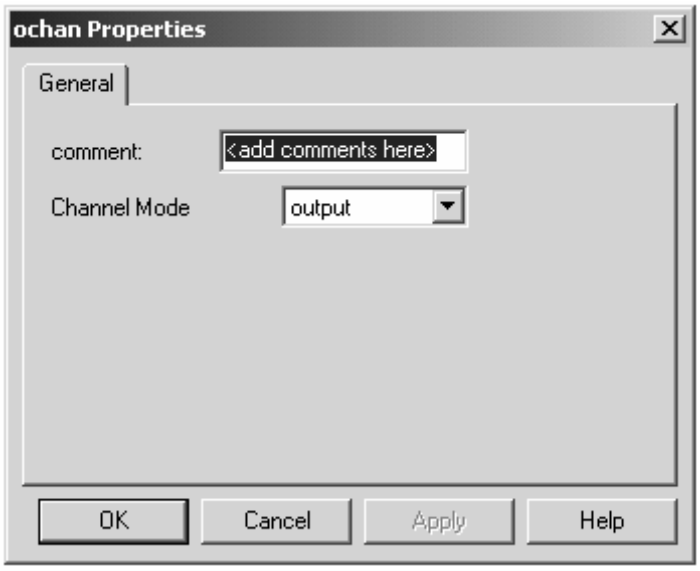

Figure 5. RTDX Manager Properties and RTDX Object Properties

For creating a channel (object) for communication via RTDX, right click on the RTDX – Real-Time Data Exchange Settings and select “Insert RTDX” in the configuration tool to bring up the RTDX object properties dialog (Figure 5) and select output if the RTDX channel handles output from the DSP to the host. Select input if the RTDX channel handles input to the DSP from the host. Setting the channel properties adds the following statement in the *cfg.h (* is the cdb file name) which is generated by the DSP/BIOS configuration tools.

```
extern RTDX_outputChannel ochan;     //ochan is the name of the RTDX object
```

If the generated *cfg.h is not included in the project then this statement should be added in the C source file. Enable the channel using the RTDX API (RTDX_enableOutput/RTDX_enableInput) before writing/reading to/from the channel.

2.4 RTA Update Modes: Stop Mode vs Real-Time Update

The DSP/BIOS Real-Time Analysis (RTA) plugins for Code Composer Studio supports two update modes: stop mode and real-time. When RTDX is available and enabled in the application from the DSP/BIOS configuration tool, then the RTA plugins will periodically update according to the designated refresh rate. The refresh rate of the RTA updates may be displayed and/or altered by right-clicking and selecting the Property dialog of the RTA Control Panel plugin window from within Code Composer Studio. The default refresh rate is 1 second.

If RTDX is not available, RTA data may still be collected into data structures within the target application. As soon as the plugins detect that the target has halted, they will retrieve their data through a simple memory read operation from Code Composer Studio. This is referred to as a “stop mode” update.

3 HST – DSP/BIOS Host Channel Interface

3.1 The HST Module

DSP/BIOS provides an HST module that manages host channel objects. Host channel objects allow DSP/BIOS applications to stream data between the target and the host. Host channels are highly useful in the early development, especially when testing software interrupt processing algorithms. Programs can use host channels to input data sets and to output the results. Once the algorithm appears sound, the host channel objects can be replaced with I/O drivers for production.

The purpose of the HST module is to implement a host-to-target run-time communication link for DSP/BIOS applications. The HST module has an underlying LNK module. The HST and LNK modules are not involved in loading, starting, and stopping of the DSP. They are only used for run-time analysis operations like gathering instrumentation data, reading/writing target memory, and streaming data to/from host files. Figure 6 shows the block diagram of the HST interface between a PC host and a DSP target.

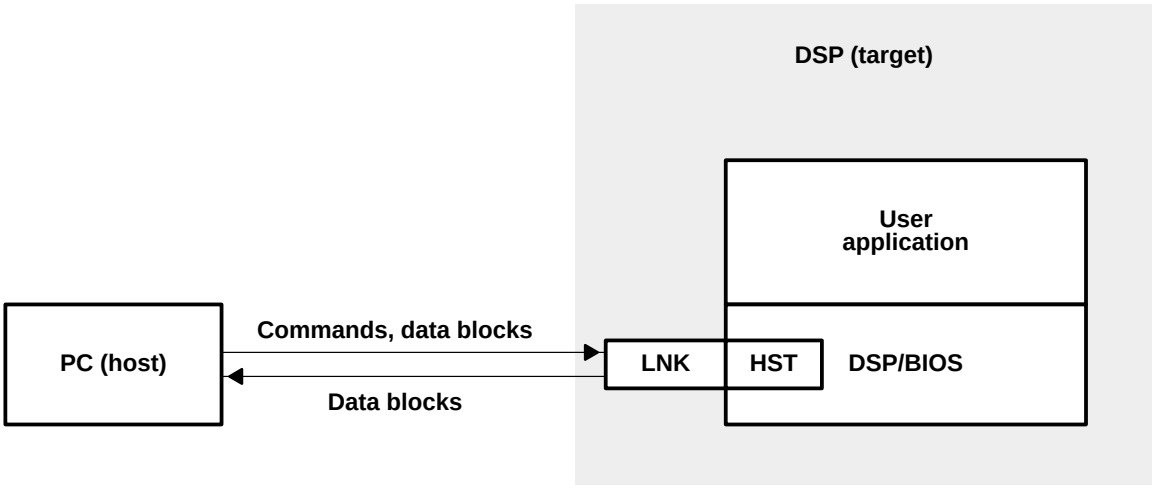

Figure 6. HST Interface Between Host and Target

HST communication is half-duplex and so at any given time, data can flow only in one direction. The HST channel used for real-time analysis is run in the IDL thread and is preempted by software interrupts (SWIs) and hardware ISRs, so the channel is only in use when the DSP has some cycles to spare for the IDL thread execution. The host channel is internally implemented using a data pipe. LNK is a board specific module that performs the handshaking between the host and target. The HST module is a generic part of DSP/BIOS that interfaces with the host using the LNK module and does not assume any particular implementation of LNK. The LNK module is the arbitrator of all host I/O and uses RTDX to communicate and exchange data with the host. It initiates and manages I/O for multiple channels and different channels can be in different states of I/O. If the client on the host side were not active, the initiated requests on the target side would stay dormant.

The host channel is internally implemented using a data pipe (PIP) object. To use a particular host channel, the program uses HST_getpipe to get the corresponding pipe object and then transfers data by calling the PIP_get and PIP_free operations (for input) or PIP_alloc and PIP_put operations (for output). DSP/BIOS also allows the use of SIO streaming between the host and target using HST. This will be covered in detail in chapter 4.

3.2 DSP/BIOS Real-Time Analysis Tools

The DSP/BIOS Real-Time Analysis (RTA) features provide developers and integrators unique visibility into their application by allowing them to probe, trace, and monitor a DSP application during its course of execution. These utilities piggyback upon the same physical JTAG connection already employed by the debugger and utilize this connection as a low-speed real-time communication link between the target and host.

DSP/BIOS RTA requires the presence of the DSP/BIOS kernel within the target system. In addition to providing run-time services to the application, the DSP/BIOS kernel provides support for real-time communication with the host through the physical link. By simply structuring an application around the DSP/BIOS APIs and statically created objects that furnish basic multitasking and I/O support, developers automatically instrument the target for capturing and uploading the real-time information that drives the visual analysis tools inside Code Composer Studio IDE. Supplementary APIs and objects allow explicit information capture under target program control as well. From the perspective of its hosted utilities, DSP/BIOS affords several broad capabilities for real-time program analysis.

When used in tandem with the Code Composer Studio IDE standard debugger during software development, the DSP/BIOS real-time analysis tools provide critical visibility into target program behavior at exactly those intervals where the debugger offers little or no insight – during program execution. Even after the debugger halts the program and assumes control of the target, information already captured through DSP/BIOS can provide invaluable insights into the sequence of events that led up to the current point of execution. Later in the software development cycle, regular debuggers become ineffective for attacking more subtle problems arising from time-dependent interaction of program components. The DSP/BIOS real-time analysis tools take an expanded role as the software counterpart of the hardware logic analyzer. This dimension of DSP/BIOS becomes even more pronounced after software development concludes. The embedded DSP/BIOS kernel and its companion host analysis tools combine to form the necessary foundation for a new generation of manufacturing test and field diagnostic tools. These tools will be capable of interacting with application programs in operative production systems through the existing JTAG infrastructure. The overhead cost of using DSP/BIOS is minimal, therefore instrumentation can be left in to enable field diagnostics, so that developers can capture and analyze the actual data that caused the failures.

3.3 The RTA Command Dispatcher

DSP/BIOS provides a built-in handler for commands originating from RTA plug-ins on the host side. This built-in handler is called the RTA command dispatcher. This dispatcher is implemented as an IDL function RTA_dispatcher. RTA_dispatcher is a real-time analysis server on the target that accepts commands from DSP/BIOS analysis tools, gathers instrumentation information from the target, and uploads it at run time. RTA_dispatcher sits at the end of two dedicated HST channels and its commands/responses are routed from/to the host via LNK dataPump.

The primary role of the RTA command dispatcher is to look for commands from the host and service them. It should be noted that all data transfers between target and host are initiated from the host side. The RTA dispatcher polls for any new command from the host and executes it. The RTA command dispatcher uses the underlying LNK driver to perform the I/O operations requested by the host commands. The RTA dispatcher is repeatedly called in the IDL loop to poll for a new command. The user-defined HST channels have nothing to do with the RTA command dispatcher. For the user-defined HST channels, DSP/BIOS does not provide any command handler. DSP/BIOS gets the data from the RTDX channel and makes it available to the user HST channels. It is the user's responsibility to decide what to do with the data coming in and going out through the HST channels. The user will have to provide a dispatcher function to process the data coming in and going out of the HST channel.

Note that the RTA dispatcher is running within the context of an IDL thread, which is the lowest priority thread in the system. If for any reason the IDL thread is starved by higher priority threads and is never permitted to execute, then the RTA plug-ins will not be able to communicate with the target. Therefore, IDL thread starvation may cause the RTA plug-ins to stop updating until the target is halted.

3.4 HST and RTDX – The HST Operation Sequence

The HST module uses RTDX for data transfer between the target and host. The HST channel has an underlying LNK module that does all the I/O operations. LNK can have several implementations, but DSP/BIOS has an RTDX implementation of LNK. It is the LNK module that communicates with the host through the RTDX channel. In DSP/BIOS, the host channel communications are performed during the IDL loop. The default IDL functions take care of data transfers between the target and the host. DSP/BIOS provides two HST objects by default. These are the RTA_fromHost and the RTA_toHost objects. These HST objects are used by the DSP/BIOS RTA plug-ins to send RTA commands to target and get RTA data from the target. The operational sequence of a HST object and its relation with RTDX along with the default HST objects is explained in Figure 7.

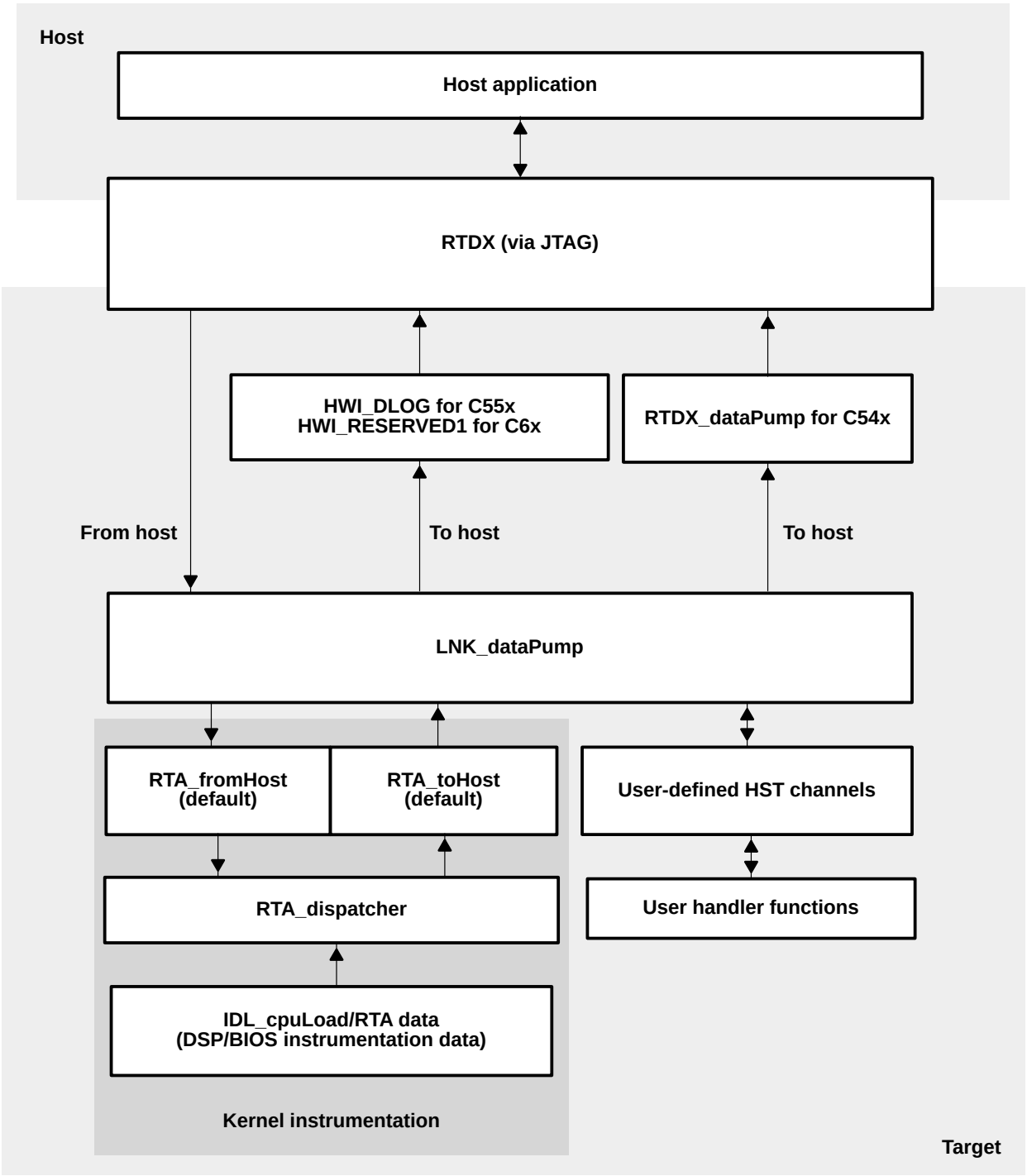

Figure 7. Data Flow for a HST Channel Using RTDX

DSP/BIOS has three default IDL functions. They are IDL_cpuLoad, LNK_dataPump and RTA_dispatcher. For C54x™, DSP/BIOS provides a fourth IDL function called the RTDX_dataPump. These IDL functions play a significant role in the real time analysis capabilities provided by DSP/BIOS. However, the IDL loop follows a lowest priority best fit approach, i.e., IDL loops have the lowest priority in the system and runs only when some cycles are spared for it. The LNK_dataPump function has an underlying RTDX link through which it fetches data from the host and makes it available for the HST channels (default and user) defined in the system. For user-defined HST channels, the user has to provide a handler function that will fetch the data from the HST module and perform the necessary processing. The RTA commands from the host arrive via the RTA_fromHost channel and are handled by the RTA_dispatcher. The RTA dispatcher executes the RTA command and the resulting data is sent to the host through the RTA_toHost channel. Data sent to the host may include execution details, log data, etc that is required by the DSP/BIOS plug-ins. The data transfer from the LNK_dataPump to the host through the RTDX link is performed differently for each platform. IDL_cpuLoad uses an STS object (IDL_busyObj) to calculate the target load. The contents of this object are uploaded to the DSP/BIOS analysis tools through RTA_dispatcher to display the CPU load.

The C55x™ and C6000™ platforms have an interrupt-driven interface for RTDX and therefore has an interrupt configured. These dedicated interrupts are HWI_DLOG for C55x and HWI_RESERVERED1 for C6000. The ISR associated with these interrupts is the RTDX_Poll function. These interrupts are invoked periodically to transfer data from target to host. The advantage here is that this interrupt has a higher priority than SWI, TSK, and IDL functions. The actual interrupt function runs in a very short time. Within the idle loop, the LNK_dataPump function does the more time-consuming work of preparing the RTDX buffers and performing the RTDX calls. Only the actual data transfer is done at high priority. This data transfer may have a small effect on real-time behavior, particularly if a large amount of LOG data must be transferred. In the case C54x, RTDX_Poll has to be called explicitly. Hence DSP/BIOS provides an RTDX_dataPump IDL function. RTDX_dataPump calls RTDX_Poll to transfer data between the target and the host. This occurs only if the DSP has enough free cycles to execute the IDL loop on a regular basis. RTDX_Poll can be called using a timer interrupt if the data to be transmitted is critical.

If communication is critical, it is possible to modify the target application so that a call to RTDX_Poll may be made from within a timer or PRD thread. Be aware however, that calls to RTDX_Poll are not re-entrant. If a re-entrant call to RTDX_Poll is attempted, the second call will merely abort. For a DSP/BIOS application, the user can remove the RTDX_dataPump IDL function and invoke the RTDX_Poll function from a PRD thread or a timer ISR instead of an IDL function. To remove the RTDX_dataPump, set the Host Link Type to None in the Host Channel Manager Properties. Now Disable RTDX from the RTDX Settings properties. This will automatically remove the IDL function RTDX_dataPump. The side effect would be that the DSP/BIOS plug-ins would stop working in real time mode. You can also have RTDX enabled in the configuration tool and along with that a PRD function to call RTDX_Poll.

The user-defined HST channels operations are similar to default HST objects except that the user has to provide handler functions to process that incoming data and for sending the correct information back to the host. DSP/BIOS embeds RTDX channels within the RTA channels.

3.5 Application Using HST for Host – Target Data Transfer

TI provides two examples, hostio1 and hostio2, that performs a data transfer between target and host. These examples are shipped with Code Composer Studio in the tutorials directory. These examples use two HST channels (input_HST and output_HST) to transfer a data file from the host to the target and then back to the host into another file. On examining the code, you can see that the SWI function A2DscaleD2A gets a PIP object from the HST channels and all further operations on the HST is performed using PIP API calls. To run the example, you will need to bind the HST channels to the input and output file from the host as follows. Load the program to the target and from the menu options, open DSP/BIOS–>Host Channel Control. This will open up the host channel window with all the user HST channels listed. Now right click on the input_HST channel object in this window and choose the option *bind*. This will allow you to bind a data file to the input channel. Similarly bind the output_HST channel to an output file. Now run the program. Right click on the output_HST channel in the Host Channel Control and select start. This will make the output channel ready to receive data. Similarly start the input_HST channel. This will make the data from the input file available to the input_HST channel. Once the program has completed execution and falls into the IDL loop, unbind the HST channels. On verifying, you can see that the contents of the input file are copied on to the output file.

4 Stream Support for HST

4.1 HST and Host Link Driver

DSP/BIOS provides support for streaming data between the host and the target for use in applications using the task model. This is provided by the Host Link Driver module, which is also referred to as the DHL module. The DHL device has an underlying HST object. DHL allows data transfers between target and host through the HST channels using SIO streaming API calls rather the using pipes. DHL module provides an interface between the SIO and HST modules. DHL devices copy the data between the frames in the HST channel's pipe and the stream buffers. The DHL devices can be opened in input mode or output mode. In input mode, it is the size of the frames in the HST that drives the data transfer. Only when all the data in a frame has been transferred to stream buffers, the DHL device will make the current buffer available to the application. Even if the stream buffers can hold more data than the HST channel frames, the stream buffers will be returned partially full. In the output mode, it is exactly the opposite. It is the size of the stream buffers that drive the data transfer. Only when all the data in the stream buffer has been transferred to the HST channel frames, the DHL device returns the current frame to the channel's pipe. The frames return to the HST pipe partially full even if the HST channel's frames can hold more data than the stream buffers. The maximum performance in a DHL device is obtained when frame size of the HST channel matches the buffer size of the stream that uses the DHL device. Another alternative is to have the stream buffer size to be larger than and a multiple of HST frame size. However the DHL does not impose any restrictions on the size of the HST frames or the stream buffers. Figure 8 shows the interface between the SIO and the HST using a host link driver.

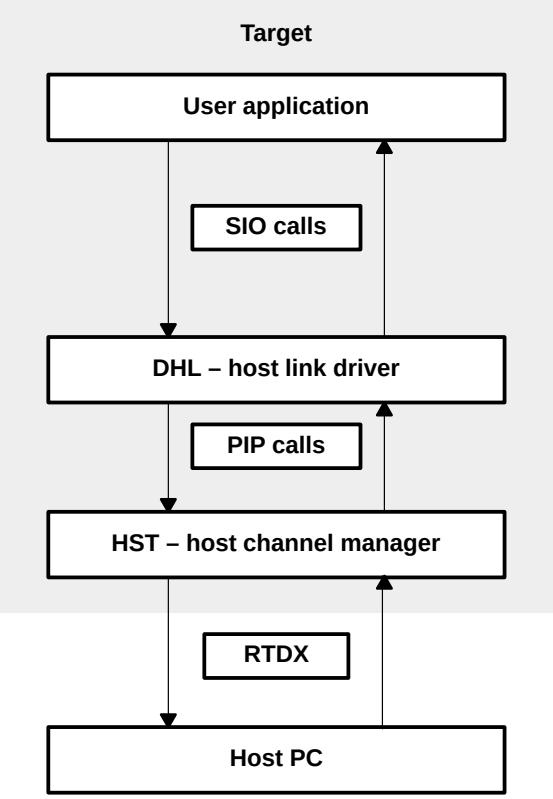

Figure 8. DSP/BIOS Stream Support for HST

4.2 Developing an Application With the DHL Interface for HST

This section will explain how to develop a target application with a HST interface between the host and target using the host link driver. This application will perform a data transfer between target and host. All data transfers will be done through SIO calls to a DHL device. The data from the input file on the host is fetched by the target and sent back to the host to be written to an output file. The file transfer sequence is the same as it is for the hostio2 example except that the target application uses a DHL device to interface with the HST channels. The steps involved in developing this application will be explained in two steps. The first section will cover the configuration of the HST, DHL and SIO objects in the configuration file and the second section will cover the application code. The example application for TMS320C6711 is provided with this application note. However, porting this sample application to any other target will be easy since only the configuration file will need to be modified.

Start Code Composer Studio for a DSK 6711 and open the configuration file dsk6711.cdb. In the configuration file, expand the Input/Output section. Under this section you can find the HST-Host Channel Manager. Right click on the host channel manager and insert two HST objects. Rename these newly created HST channels as ‘input_HST’ and ‘output_HST’. Right click on these HST objects and choose properties to configure the HST channel as shown in Figure 9.

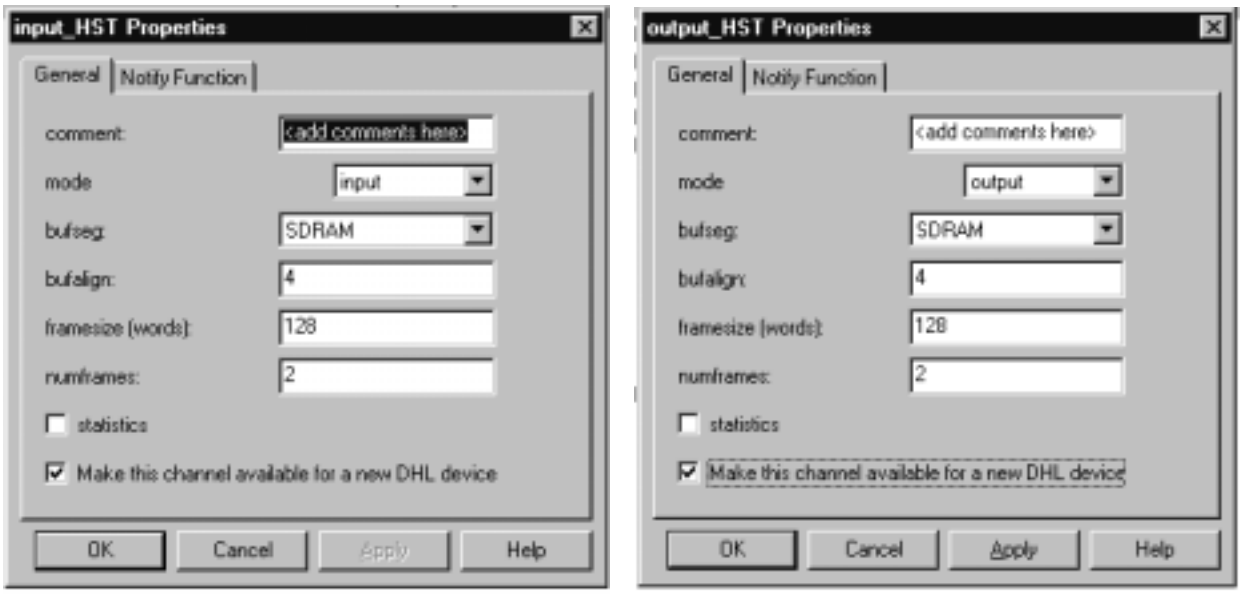

Figure 9. HST Channel Properties

In Figure 9, you can see that both the input and output HST channels are configured with two frames of size of 128 with an alignment of 4. The comment field can be ignored. The statistics check box is used if we want to monitor this channel with an STS object. You can display the STS object for this channel to see a count of the number of frames transferred with the Statistics View Analysis Tool. The second check box “Make this channel available for a new DHL device” is only used when we want to use SIO calls to perform data transfer. The configuration tool will allow you to insert a DHL object only if this check box is enabled. For this example we will need this option to be enabled.

The notify function tab provides three fields: a notify function and the two arguments to it. This tab can be ignored, since the notify function and the arguments are set automatically by the configuration tool if the DHL option is selected. The notify function is set to DHL_notify if the DHL option is set in the HST properties (refer to Figure 10). At this point we have finished configuring the HST channels.

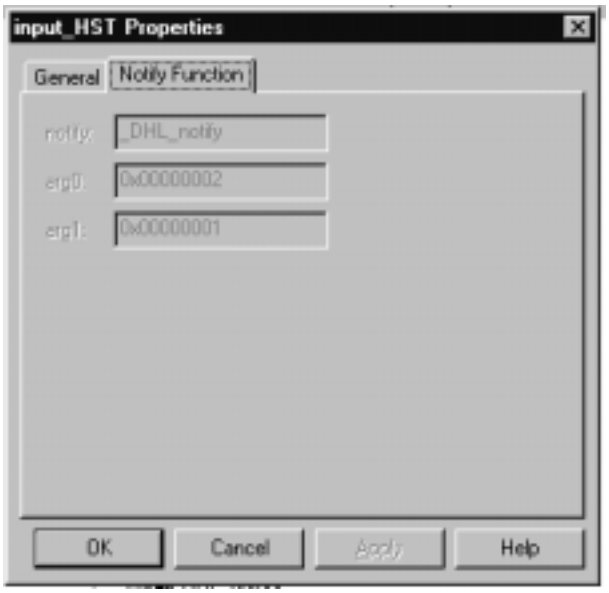

Figure 10. HST Notify Function

To configure the DHL devices, expand the SIO Drivers section. Right click on the DHL- Host Link Driver section and insert two DHL objects. DHL devices can be inserted only if an HST channel is defined and it is made available to the DHL driver by enabling the option “make this channel available for a new DHL device”. Rename these DHL objects as DHL_input and DHL_output. The DHL objects should have an underlying HST channel. Right click on the DHL objects and open the DHL properties box. Select the underlying HST channel for these DHL devices as shown in Figure 11. Once an HST channel is tied to a DHL device, the channel is owned by the DHL device and no longer available to other DHL channels.

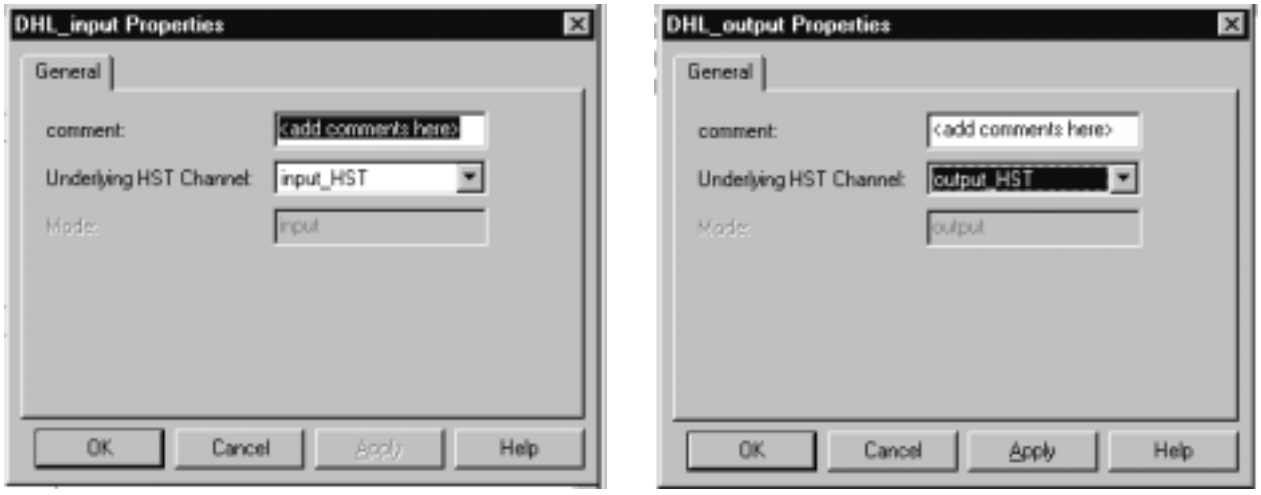

Figure 11. DHL Object Properties

The configuration tool will set the mode of the DHL device depending on the mode of the underlying HST channel. At this point we have finished configuring a DHL device with an underlying HST channel.

DHL devices can be opened for input or output data streaming. A DHL device used by a stream created in output mode must be associated with an output HST channel. A DHL device used by a stream created in input mode must be associated with an input HST channel.

To configure the SIO object, right click on the SIO-Stream Input and Output Manager and insert two SIO objects. Rename the SIO objects as SIO_input and SIO_output. To use a DHL device in a stream created with the Configuration Tool, select the device from the drop-down list in the Device box of its Properties dialog. In the properties dialog box, set the correct device and mode for the input and output streams. Note that the device selected for the input stream should have an underlying HST channel that is configured for input. Also enable the option 'Allocate Static Buffers'. Figure 12 explains configuration of a SIO object. For more details on SIO configuration, refer to Code Composer Studio Help.

To use a DHL device in a stream created dynamically with SIO_create, use the DHL device name (as it appears in the Configuration Tool) preceded by "/" (forward slash) as the first parameter of SIO_create:

```
SIO_input = SIO_create("/DHL_input", SIO_INPUT, 128, NULL);
```

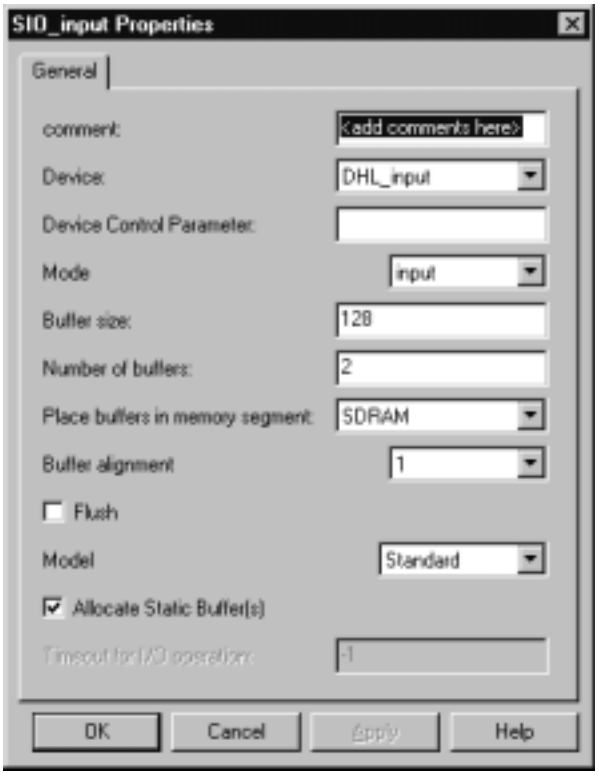

Figure 12. SIO Object Properties

Now we need a task to be created that will perform the host-target data transfer. Insert a task object and rename it as transfer_task. In the properties dialog box for this task, define the function (transfer_function) associated with the task and the arguments to this task function. The SIO_input and SIO-output streams are passed as inputs to the task. Refer to Figure 13. The first argument is set to 1 here. Argument 0 is used in the task function to determine the number of transfers to be done. This value can be set to any value depending on the amount of data to be transferred and program logic involved. All other fields in the task properties dialog can be ignored, as they are not relevant here. Save the configuration file.

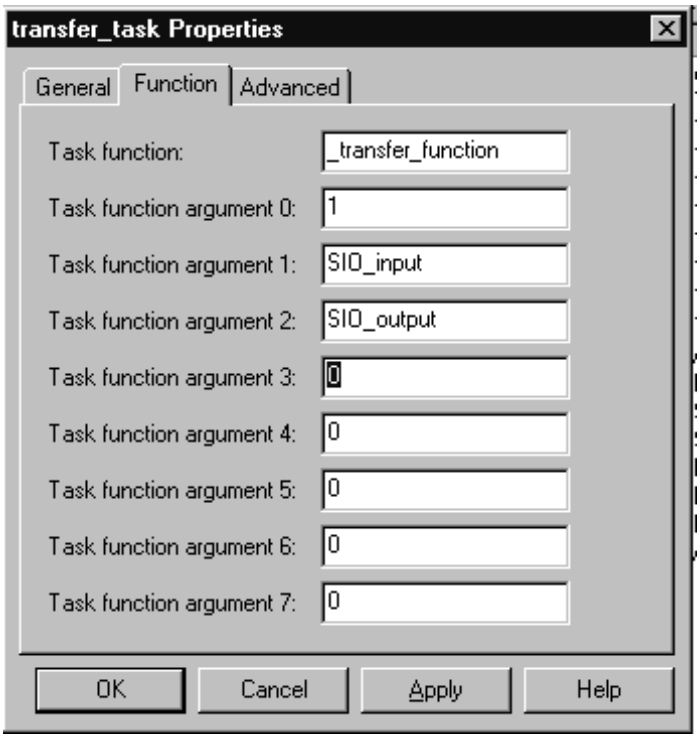

Figure 13. Task Properties

The code for this example is given in Appendix A. In the code you can see that in the function 'transfer_function', we use only SIO calls and the implementation is very simple compared to a non-DHL version of this. Create a new project and add the above configured cdb file, the generated cmd file and the 'c' source file from appendix A. Build the project and load the program. Before you run the program, you need to open the Menu→DSP/BIOS→Host Channel Control and bind the input (sine.dat) and output files to the HST channels. Start the input and output HST channels and run the program, you can see that the data from the input file gets copied to the output file. The task function performs the data transfer through the associated DHL device that has an underlying HST channel. The status of the data transfer can be seen in the Host Channel Control plug-in as shown in Figure 14.

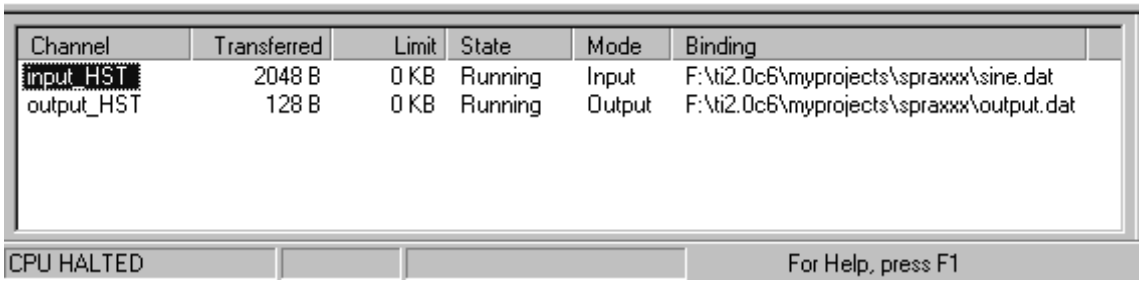

Figure 14. HST Channel Control Plug-In

For more information on DSP/BIOS and its use, see the *DSP/BIOS User's Guide* (SPRU423) in the online help of Code Composer Studio.

5 References

1. *RTDX Online Help* (SPRH059)
2. *DSP/BIOS User's Guide* (SPRU423)
3. *How to Use High-Speed RTDX Effectively* (SPRA821)

Appendix A hst2dhl.c Listing

```
/*
 *  Copyright 2001 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *
 */
/*************************************************************************/
/*                                                                       */
/*     hst2dhl.c                                                         */
/*                                                                       */
/*     An example that performs a streaming data transfer between target   */
/*     and host using the DSP/BIOS DHL driver and the SIO interface.       */
/*                                                                       */
/*     Author: Harish Thampi S                                           */
/*     Date:   09/16/2002                                                */
/*                                                                       */
/*************************************************************************/
#include <std.h>
#include ”hst2dhlcfg.h”  /* All DSP/BIOS related header files are included */
                         /* in the generated header file                  */
/* Task function tied to ”transfer_task” */
Void transfer_function(Uns nloops, SIO_Obj *input, SIO_Obj *output);
/*
 * ======== main ========
 */
Void main()
{
}
/*
 *  ======== transfer_function ========
 *
 * FUNCTION: Called from transfer_task TSK to get data from a host file
 *           through an SIO input stream and send output data back to
 *           the host through an SIO output stream.
 *
 * PARAMETERS: address of input and output streams
 *
 * RETURN VALUE: None.
```

```
 */
Void transfer_function(Uns nloops, SIO_Obj *input, SIO_Obj *output)
{

              Ptr           buf;
              Inti,         nbytes;

              if( SIO_staticbuf(input, &buf) == 0 ) {
                            SYS_abort( ”%s”, TSK_getname(TSK_self()) );
              }

              for( i = 0; i < nloops; i++ ) {
                            if( (nbytes = SIO_get(input, &buf)) < 0 ) {
                                 SYS_abort( ”Error %s”, TSK_getname(TSK_self()) );
                            }

                            /* Apply algorithm to buffer(s) of data */

                            if( SIO_put(output, &buf, nbytes) < 0 ) {
                                 SYS_abort( ”Error %s”, TSK_getname(TSK_self()) );
                            }
              }

}
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265