Bachelor Thesis

Ioana Semedrea

Automatic Keyword Controlled GUI Screen Layout Generation for RTDX Applications Running on TMS320C6713 DSP

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Ioana Semedrea

Automatic Keyword Controlled GUI Screen Layout Generation for RTDX Applications Running on TMS320C6713 DSP

Bachelor Thesis based on the examination and study regulations for
the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr. Ulrich Sauvagerd
Second examiner: Prof. Dr. rer. nat. Hans-Jürgen Hotop

Day of delivery 26 August 2009

Ioana Semedrea

Title of the Bachelor Thesis

Automatic Keyword Controlled GUI Screen Layout Generation for RTDX Applications Running on TMS320C6713 DSP

Keywords

C6713, DSP, DSK board, RTDX, real-time applications, host-to-target connection, RTDX channel, RTDX block, C-programming

Abstract

The aim of this bachelor project is the implementation of a PC application that assists in the debugging and controlling of digital signal processor applications running on the Texas Instruments TMS320C6713 processor. The developed PC application provides the infrastructure for transferring data between a DSP and a PC in real time, without interrupting the execution of the DSP program. The data transferring procedure is based on the Real-Time Data Exchange technology of Texas Instruments. The distinguishing characteristic of the developed application is its ability to adjust itself to the specific requirements of each DSP application. The user can determine the layout of the PC application screen through a set of keywords inserted in the DSP program code. A graphical user interface offers the necessary tools for visualizing and modifying the DSP data. The exchange of data from PC to DSP is fully realized. From DSP to PC data exchange is available for selected data structures and it can be extended to all supported data structure types.

Ioana Semedrea

Thema der Bachelorarbeit

Automatische Keyword-gesteuerte Generierung von GUI-Layouts für TMS320C6713 DSP RTDX-Anwendungen

Stichworte

C6713, DSP, DSK-Board, RTDX, Echtzeitanwendungen, Host-Target Kommunikation, RTDX-Kanal, RTDX-Block, C Programmierung

Kurzzusammenfassung

Ziel dieser Bachelorarbeit ist die Implementierung einer PC-Anwendung, die eine auf dem TMS320C6713 Prozessor von Texas Instruments laufende Anwendung für digitale Signalverarbeitung beim Debugging und der Steuerung unterstützt. Die entwickelte PC-Anwendung stellt die Infrastruktur für Datenübertragung zwischen PC und DSP in Echtzeit bereit, ohne den Ablauf des DSP-Programms zu unterbrechen. Der Datenaustausch basiert auf der von Texas Instruments bereitgestellten Real-Time Data Exchange Technologie. Der Schwerpunkt der entwickelten Anwendung ist deren Fähigkeit, sich an die spezifischen Anforderungen jeder DSP-Anwendung anzupassen. Der User kann durch im Code des DSP-Programms eingefügte „Keywords“ das Layout des PC-Anwendungsfensters bestimmen. Die Datenübertragung vom PC zum DSP ist vollständig realisiert. Für ausgewählte Datenstrukturen ist auch die Übertragung vom DSP zum PC realisiert, wobei eine Erweiterung für alle unterstützten Datenstrukturen einfach realisiert werden kann.

Contents

1 Introduction

Digital signal processing is concerned with the electronic processing of signals or applications which demand fast numerical computing. It is employed in a wide variety of domains, spanning from audio, speech and image processing to controls and communications.

Digital signal processors (DSPs) perform predominantly real-time signal processing. This implies keeping pace with an external event, usually an analog signal. In analyzing real-time DSP systems, application designers need to obtain accurate, real-world information to ensure proper results. In this context, traditional debugging practices such as stopping the application at designated breakpoints to read data storage locations can only deliver isolated snapshots of the system operation. Since data gathered through this procedure may not correctly reflect the behavior of the running DSP application, Texas Instruments' Real-Time Data Exchange technology (RTDX) comes to respond to the arising need for a more accurate analysis mechanism.

RTDX enables the bidirectional exchanging of data between a host computer, namely a PC and a target processor, the DSP, through data pipelines, in real time and with minimal interference with the operation of the target software. It facilitates the tracking of tasks being executed on a DSP, the gathering of real-time statistics on system execution and the displaying of variables in real-time, hence shortening development time. RTDX is well-suited for a variety of control, servo, audio or embedded applications, as well as some image processing applications.

The bachelor project presented in this thesis resulted in the context of building an RTDX host application that would assist the DSP development carried out within the Digital Signal Processing Laboratory of Hamburg University of Applied Sciences (HAW). The leading feature of the desired PC host application is the flexibility to automatically create an appropriate screen layout and supporting underlying structure for exchanging data in real time with every application running on the TMS320C6713 digital signal processor of Texas Instruments.

The developed application, RTDX Display, employs the RTDX technology in the background of a graphical user interface (GUI), to provide a series of tools for the transfer of data from the host PC to the target DSP, and partially from the target to the host. It is designed to work in combination with a pattern of keywords indicated in the target application, thus allowing the user to easily choose the most convenient format for displaying the data of interest. This design strategy empowers the RTDX Display with the ability to self-adjust according to the individual requirements of each DSP application it serves, relieving the user from the need to create or modify regularly the supporting host application.

The role of the RTDX Display application within an RTDX enabled system is illustrated in Figure 1.1. The original diagram was extracted from reference [6].

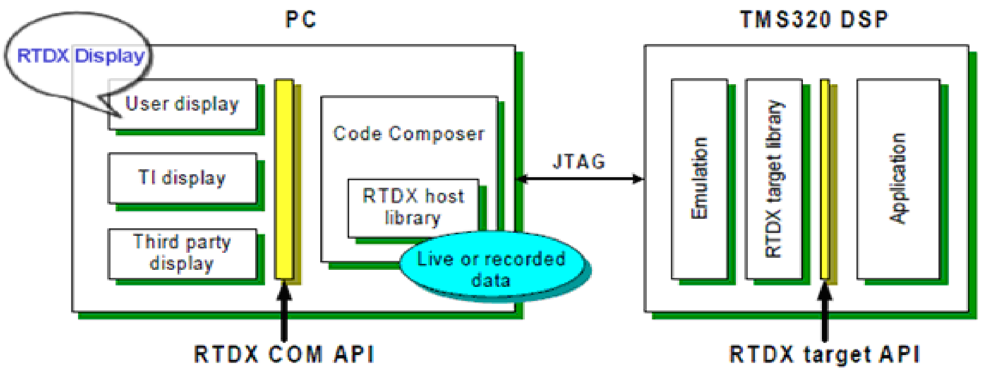


Figure 1.1 An RTDX enabled DSP-PC system employing the RTDX Display application

After an introduction to the RTDX technology, this thesis elaborates on the requirements, design and implementation of RTDX Display, and lays out possible directions for its further development.

2 Hardware and Software Background

For the realization of the RTDX Display application, a DSK (DSP Starter Kit) board containing the TMS320C6713 digital signal processor from Texas Instruments (TI) was used, along with TI's accompanying PC-hosted Code Composer Studio (version 3.1.0), an integrated development environment (IDE) which incorporates the necessary software support tools for DSP application design. The Real-Time Data Exchange is a standard component of the TI DSP and it is available with Code Composer Studio (CCS). An overview of the DSP hardware to which the PC application connects, and a description of the data transfer technology realizing the communication between the two sides are given here.

2.1 The DSK Board with TMS320C6713 DSP

The TMS320C6713-based DSK board manufactured by Spectrum Digital is a standalone platform for the evaluation and development of applications for the TI C67xx DSP family. An image and a block diagram of the board are shown in figures 2.1 and 2.2.

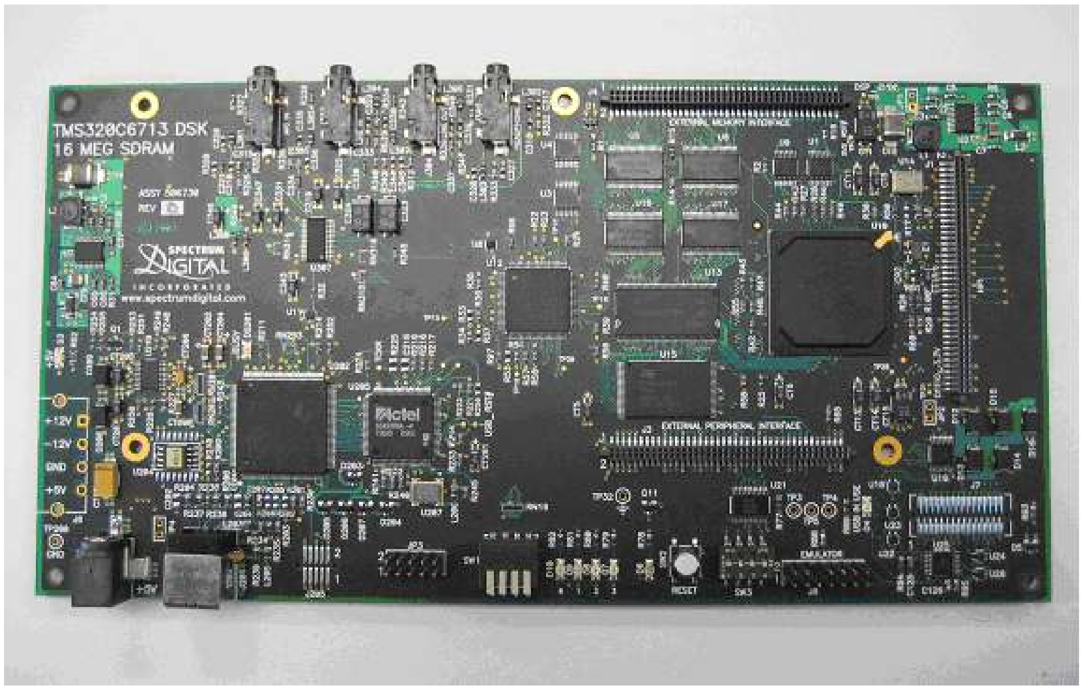


Figure 2.1 TMS320C6713-based DSK board [3]

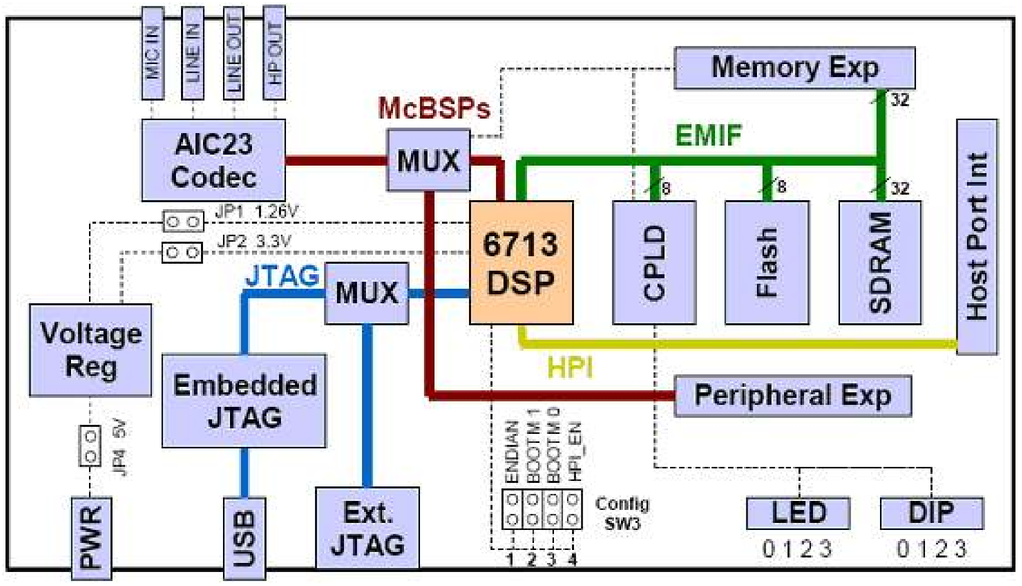


Figure 2.2 Block diagram of the C6713 DSK [2]

A summarized list of the board's key features and devices, as seen in Figure 2.2, is taken from reference [2]:

- a TMS320C6713 DSP (C6713) operating at 225MHz
- a TLV320AIC23 (AIC23) stereo audio codec
- 16 Mbytes of synchronous DRAM
- 512 Kbytes of non-volatile flash memory
- 4 user accessible LEDs and DIP switches
- software board configuration through registers implemented in CPLD
- standard expansion connectors for daughter card use
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator
- single voltage power supply (+5V)

The C6713 onboard the DSK is a floating-point processor, capable of fixed-point arithmetic as well, and based on the very-long-instruction-word (VLIW) architecture. Reference [4] contains a complete technical documentation on the C6713 DSP.

The AIC23 stereo codec provides analog-to-digital conversion and digital-to-analog conversion for the board's input and output of analog audio signals. It communicates with the DSP through two multichannel buffered serial ports (McBSPs), of which McBSP0 serves as control for the codec configuration, and McBSP1 for bi-directional transfer of digital audio samples. Sampling rates between 8 and 96 kHz are supported.

A daughter card expansion permits the board to be used in conjunction with third party add-in boards. By re-routing the McBSP0 and McBSP1 ports to the expansion connectors in software, the C6713 DSP can be joined to external codecs. One such setup is presented in Figure 2.3, where a DSK board is connected to two daughter card codecs, yielding a total of four analog inputs and four analog outputs to the codecs. In the development of the present application a DSK board with this setup is used.

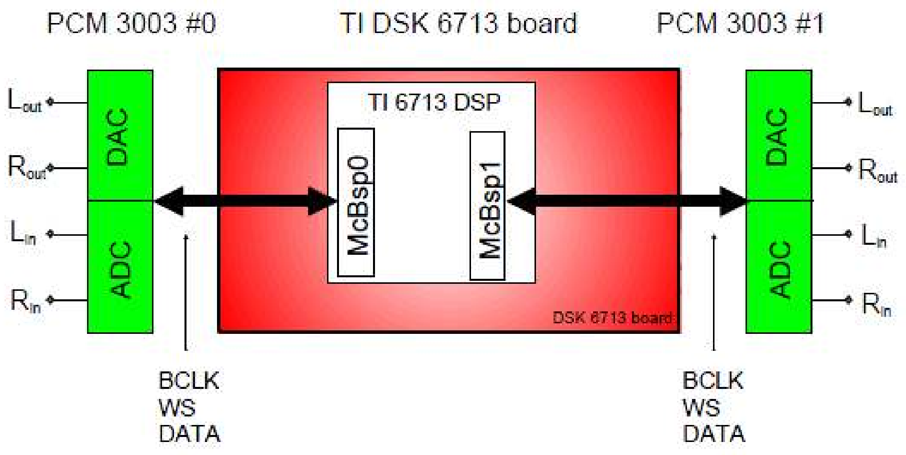


Figure 2.3 C6713 DSK board with two PCM3003 daughter cards [3]

The register based user interface of the programmable logic device (CPLD) allows the user to configure the board by reading and writing to these registers. For interactive feedback, the board includes 4 LEDs and a 4 position DIP switch, also accessed through reading and writing to the CPLD registers. The DSK board communicates with CCS on the host PC through an embedded Joint Test Action Group (JTAG) emulator with a USB host interface. An external JTAG connector enables the DSK to be used with an external emulator as well. Further information on the C6713 DSK board can be found in References [2] and [3].

2.2 The Real-Time Data Exchange Technology

The Real-Time Data Exchange technology offers the developer continuous visibility into the execution of DSP applications, in real-time, by transferring data between a host computer and the target DSP without interfering with the running target application. In this thesis, the two platforms are referred to in short as target and host. The data can be analyzed in a host client software with the help of the Microsoft Component Object Model (COM) interface available with RTDX, delivering to the user a realistic representation of the DSP application behavior. RTDX consists of target and host components, and operates as a

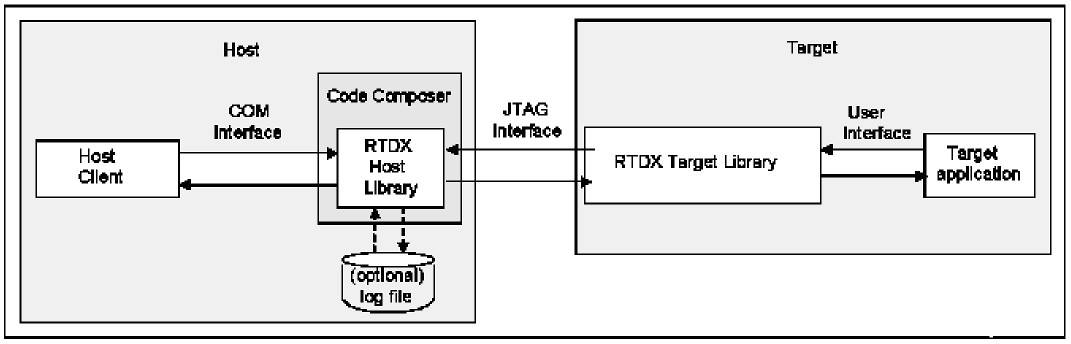


Figure 2.4 RTDX data flow [7]

collection of virtual unidirectional channels between the two platforms, as suggested in Figure 2.4.

To establish **target-to-host** communication, an output channel must be configured on the target side. A small RTDX library located on the target provides functions for writing data to the channel. The data is then automatically recorded into a target buffer defined in the target library. From this buffer, data is sent to the host through the JTAG interface. The host receives the data from the JTAG interface and stores it either into a memory buffer, or into a log file, depending on the RTDX mode of operation. The collected data can be viewed in a GUI based application of choice on the host PC. The process can be visualized as in Figure 2.5.

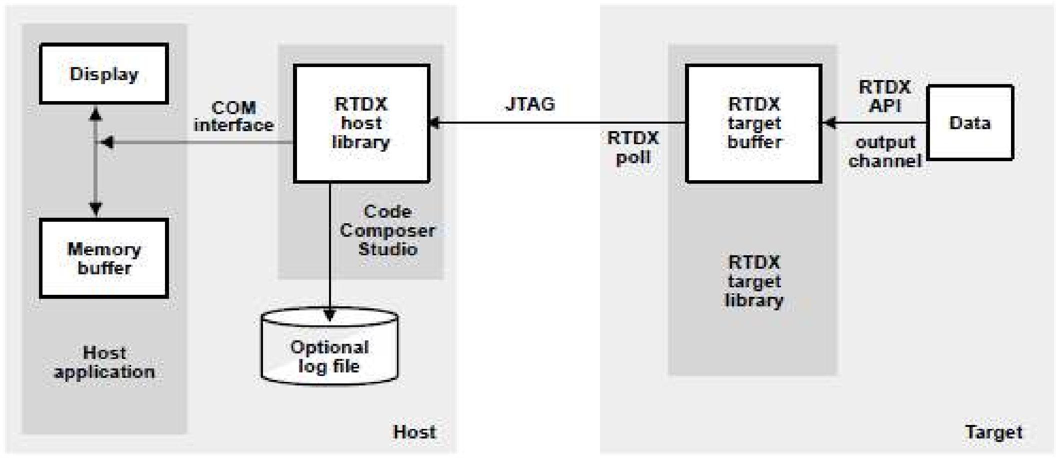


Figure 2.5 Target-to-host communication [5]

Similarly, for **host-to-target** data transfer, an input channel must be created on the target side. The RTDX host library buffers all data sent to the target and waits for a transfer request

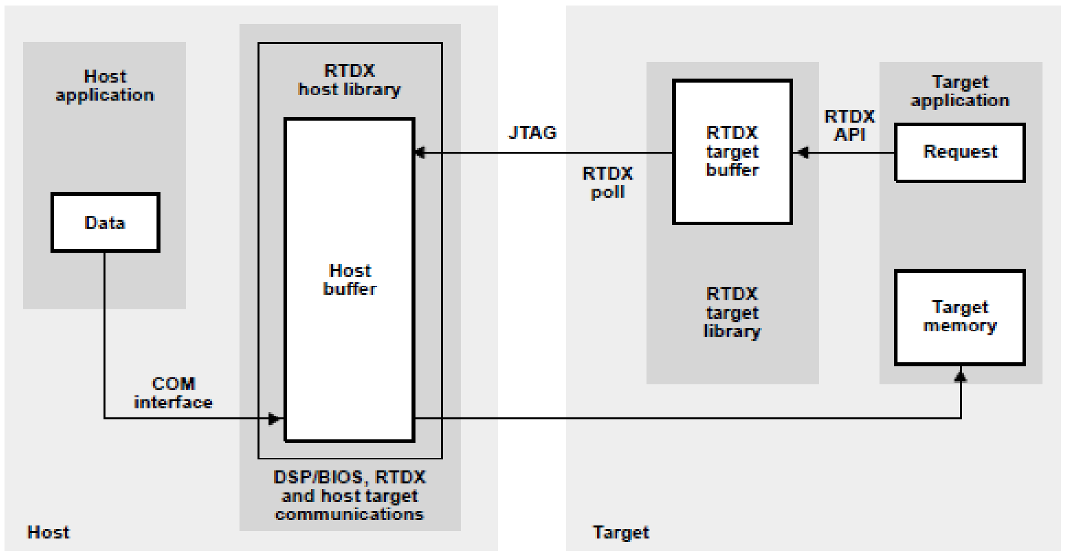


Figure 2.6 Host-to-target communication [5]

from the target. When a request is intercepted and the buffered data is sufficient to satisfy the request, the data is transferred to the specified location and the RTDX target library is notified of the transfer completion. Figure 2.6 shows a diagram of this process.

Following is a description of the CCS project configuration properties that need to be considered when using RTDX, along with key RTDX features and useful control tools available with the CCS IDE.

DSP/BIOS is a kernel organized as a set of modules, packaged as a run-time library and integrated with the Code Composer IDE. It provides run-time services used to build DSP applications and manage application resources. The kernel includes a configuration tool, with the help of which a **configuration file** (.cdb) is generated, along with support files and a linker command file (.cmd). The configuration file can be further managed within a visual editor. The DSP/BIOS Configuration Tool handles automatically a number of RTDX settings discussed next.

A project using the RTDX technology needs to link the corresponding **RTDX target library** (rtdx.lib in the case of C6713), and to specify the **RTDX include directory** (ccsinstall\TARGET\rtdx\include). For compatibility with some previous RTDX releases, the rtdx_evt.h file is provided.

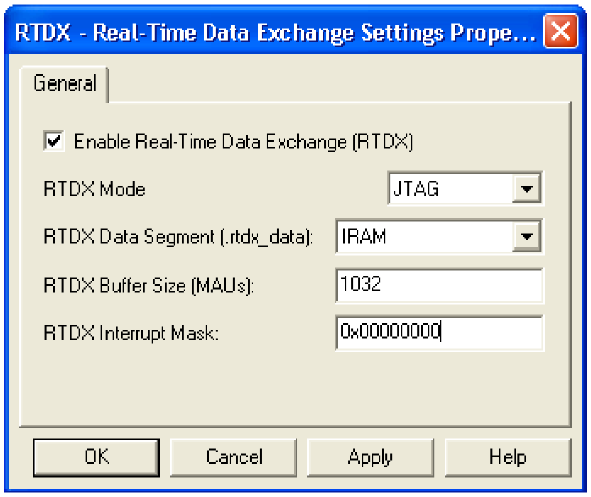
Two implementations of RTDX are available, polling-driven and interrupt-driven, specific to different processor families. The **polling implementation** relies on regular calls to the *RTDX_Poll()* function of the target library in order to capture the data on the channel. The ideal performance of the target application using this implementation is obtained by balancing between low polling rates which lead to low data rates, and high polling rates which lead to high overhead.

In the **interrupt-driven implementation** used by the C6x family of processors (including the C6713), the emulation logic makes calls to the *RTDX_Poll()* function when appropriate. For host-to-target transfers, the host triggers a special emulation interrupt on the target (the so-called message interrupt), whose interrupt service routine (ISR) triggers *RTDX_Poll()* for the transfer to occur. Within the same implementation, the RTDX host library receives data from the target through the emulation driver for CCS, which polls the JTAG interface continuously while RTDX is enabled. The DSP/BIOS configuration sets the two interrupts reserved for RTDX (HWI_INT3, HWI_RESERVED1), maps the appropriate ISRs and sets the interrupt masks. Without the employment of the DSP/BIOS, interrupt related settings must be manually configured. On the C6x processor family, polling-based DSP applications, which continuously poll the McBSP for data ready to be received or transmitted, use the interrupt-driven RTDX implementation as well.

The RTDX protocol uses data buffers on both sides of the communication. The **RTDX target buffer** temporarily stores data that is waiting to be transferred to the host. The buffer size is given in minimal addressable units (MAUs) and it varies per target. In a DSP/BIOS configured project, the RTDX target buffer is handled by DSP/BIOS and can be changed with the DSP/BIOS Configuration Tool (Figure 2.7). By default, the buffer size for the C6713 processor is set to 1032 MAUs. At the same time, memory is automatically allocated for the

RTDX Data Segment (.rtdx_data), used for buffering the target-to-host data transfers, and for the **RTDX Text Segment** (.rtdx_text), used for the RTDX code.

For projects not using the DSP/BIOS, the buffer size can be modified by adding to the project a copy of the target specific rtdx_buf.c file (found under ccs\TARGET\rtdx\lib), which is used by the RTDX buffer manager. The constant BUFRSZ defined in this file can be set to the desired value. Reference [5] provides detailed instructions on configuring RTDX without DSP/BIOS.



☑ Enable Real-Time Data Exchange (RTDX)

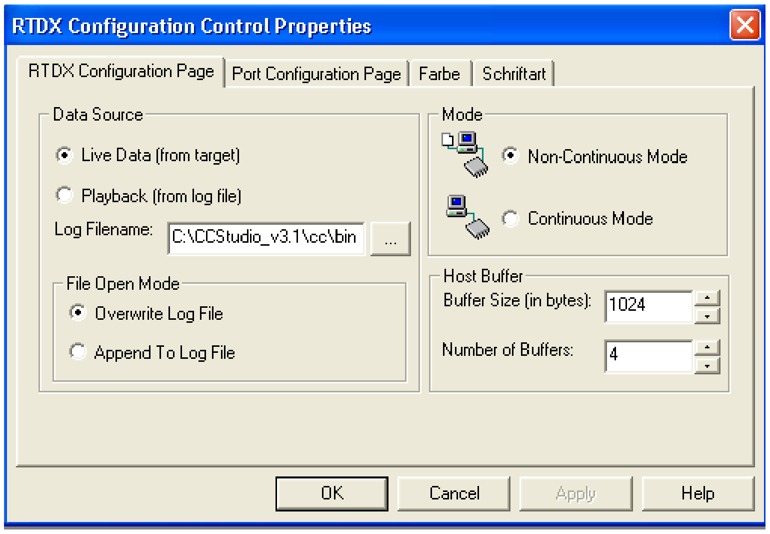
RTDX Mode	JTAG
RTDX Data Segment (.rtdx_data):	IRAM
RTDX Buffer Size (MAUs):	1032
RTDX Interrupt Mask:	0x00000000

Figure 2.7 RTDX Manager Properties

The **RTDX host buffer** stores data transferred from the target when the continuous operation mode is chosen. For 32-bit target architectures, the buffer must be eight bytes larger than the largest message coming from the target, while for 16-bit architectures four extra bytes are required. In a multiprocessor setup, at least one buffer is needed for each processor. The default size of the main buffer is 1024 bytes, and a minimum of four buffers are automatically configured. The buffers can be modified from the RTDX Configuration Page in CCS (Figure 2.8).

RTDX supports two **modes of operation** for receiving data from the target: continuous and non-continuous. In continuous mode, data is recorded in a circular memory buffer by the RTDX host library. The host client must uninterruptedly read from the output channel coming from the target in order to drain the host buffers of data.

In non-continuous mode, data received from the target is written to a log file (*.rtd) on the host. An RTDX host client can read and process data from the log file as it becomes available. In this case, the Data Source must be set to Live Data (from target) in the RTDX Configuration Page (Figure 2.8). Alternatively, the log file can be viewed with the help of the RTDX Dump Utility, for non-real-time analysis. The RTDX Dump Utility, which is provided by TI along with the RTDX technology (ccsinstall\examples\hostapps\rtdx\dumprtd\dumprtd), converts the binary log file into a text file. In this case, the corresponding Data Source is Playback (from log file).



RTDX Configuration Control Properties

RTDX Configuration Page | Port Configuration Page | Farbe | Schriftart

Data Source

[✓ Live Data (from target)]

Playback (from log file)

Log Filename: C:\CCStudio_v3.1\cc\bin

File Open Mode

[✓ Overwrite Log File]

Append To Log File

Mode

[✓ Non-Continuous Mode]

Continuous Mode

Host Buffer

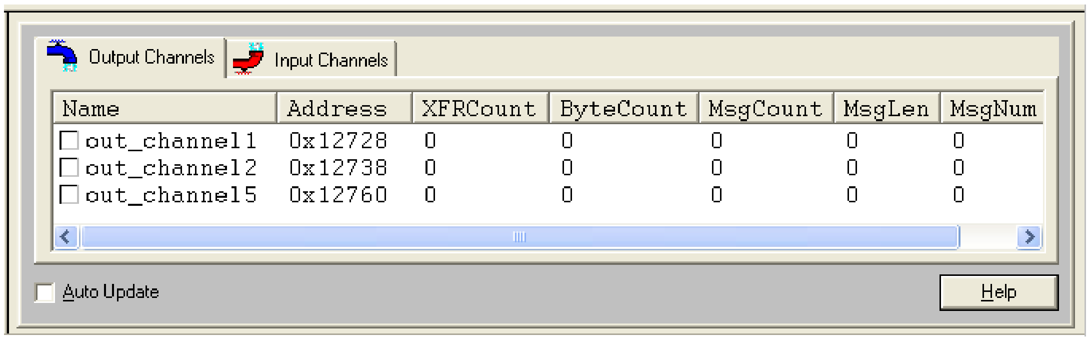
Buffer Size (in bytes): 1024

Number of Buffers: 4

OK | Cancel | Apply | Help

Figure 2.8 RTDX Configuration Page in Code Composer Studio

Data is transferred between the two communicating RTDX libraries across channels. As a matter of convention, channels sending data out of the DSP are called output channels, while those routing data into the DSP are referred to as input channels. For viewing the initialized channels, the CCS IDE provides the **Channel Viewer Control.** As can be seen in Figure 2.9, for output channels the control shows the status of the channels (represented by the check box and disabled in the displayed case), the number of messages transferred on each channel (XFRCount), the number of bytes waiting to be read (ByteCount), the number of messages waiting to be read (MsgCount), the length of the current message in bytes (MsgLen) and the sequence number of the current message (MsgNum). For accessing the data on the channels, the developer must make use of a host display software.



Output Channels | Input Channels

Name	Address	XFRCount	ByteCount	MsgCount	MsgLen	MsgNum
☐ out_channel1	0x12728	0	0	0	0	0
☐ out_channel2	0x12738	0	0	0	0	0
☐ out_channel5	0x12760	0	0	0	0	0

☐ Auto Update | Help

Figure 2.9 RTDX Channel Viewer Control

In terms of **RTDX performance**, two measures are considered: the communication rate and the target rate. The **communication rate** refers to the target-to-host data exchange rate. The so-called "standard RTDX" technology described in this chapter which uses the XDS510-class (eXtended Development System) scan-based emulator is capable of data rates of 10 Kbytes/sec. and higher [12]. The achieved communication rate from host to target is more unpredictable, due to the fact that two data transmissions occur for this type of transfer, namely a request from the target followed by data being supplied by the host. The rate of retrieving target data from the host library by the host client varies also, as the underlying host operating system may not conform to real-time constraints. Some of the recommendations given in reference [11] for improving the communication rate include:

- Running on a host PC with a faster CPU and more memory, which allows the emulation driver faster and longer lasting polling of the JTAG interface for data.
- Reducing the number of tasks competing for CPU execution cycles, allowing the emulation driver more uninterrupted polling of the JTAG interface.
- Saving the log file always on a local hard drive, as saving over a network connection may violate real-time constraints of RTDX and may cause data loss.

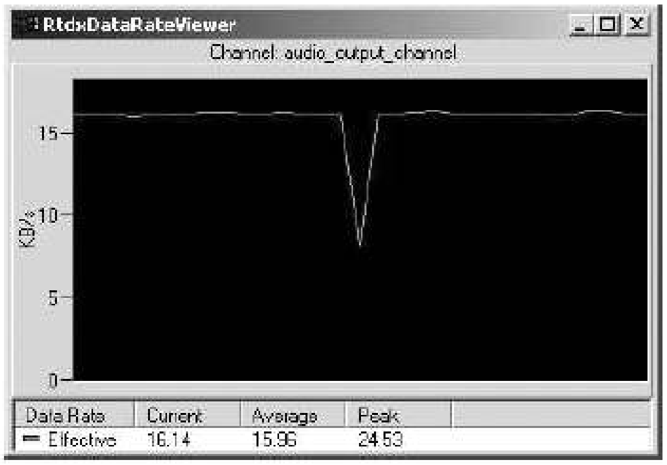
To reduce data transfer overheads on the host, larger data blocks can be passed to and from the RTDX interface, as opposed to many small blocks.

The **target rate** measures the amount of target resources consumed by RTDX. As recommended in reference [11], the percentage of target processor capacity used by RTDX can be reduced by:

- Increasing the clock rate of the DSP, for a faster execution of the RTDX communication mechanism.
- Sending data in fewer and larger blocks, which decreases the impact of the overhead associated with each transfer. The required RTDX control information is two words per message, which amounts to 64 bits (2*32 bits) per message in the case of the C6713 DSP. According to reference [5], the total effect of the RTDX activity on the real-time behavior of the DSP application is given by the actual data transfer, specifically by the size of the message including the overhead, which is the single RTDX task executed at high-priority. The more time-consuming work of preparing the RTDX buffers and performing the RTDX calls is done within the idle loop.
- Linking RTDX code and data to on-chip memory, which is generally faster than external memory, or reducing the wait states of the target system, to allow for quicker data access and code execution in external memory.
- Reducing the polling frequency, in polling implementations of RTDX.

To assist in diagnosing target-to-host transfer issues, TI provides the RTDX **Data Rate Viewer Control** as an ActiveX control available with CCS. The Data Rate Viewer (Figure 2.10) monitors and displays the effective target-to-host throughput of the target application. The current, average and peak data rates for a given output channel and the amount of bandwidth consumed by the RTDX transfer overhead are displayed. The control does not measure host-to-target throughput.

The CCS IDE supports C language and assembly language DSP development. For **assembly language** target applications, a corresponding RTDX interface (rtdx.i) exists in the form of a macro interface include file. Data communication is possible in both directions between an assembly level application and a host client.



Data Rate	Current	Average	Peak
━ Effective	16.14	15.96	24.53

Figure 2.10 Data Rate Viewer Control [9]

RTDX can be used in a **multiprocessor environment** as well. The RTDX host library is equipped to handle such a setup from the host application, with the restriction that all connected processors must share the same configuration for the operation mode, the host buffers and the log file, as these are global attributes of RTDX.

Texas Instruments produces **DSP simulators** for most of its device configurations. Multiple simulators are available, as they offer a variety of debug and tuning features. Specific RTDX target simulator libraries exist to support RTDX when running the DSP application inside simulators.

On the host platform, **display and analysis tools** communicate with the TI CCS debugger through Microsoft's Automation, an inter-process communication mechanism based on the COM model. The host application, an Automation client, extracts an RTDX wrapper class from the RTDX type library rtdxint.dll exposed by the RTDX component and included with Code Composer Studio. In turn, the CCS debugger acts as an Automation server. Designers can develop COM enabled customized applications using standard software display packages, such as National Instruments' LabVIEW, Quinn-Curtis' Real-Time Graphics Tools and Microsoft Excel, or build their own GUI applications in Visual C++, Visual Basic or MATLAB. The Code Composer Studio installation includes a few basic host applications, such as the RTDX General-Purpose Display and the RTDX Dump Utiliy. Chapter 3 gives an overview of the existing RTDX enabled host applications, and of the possible environments for developing new ones.

The host side COM API is provided by version 1.0 of standard RTDX. A later extension using the same type of emulator comes with version 2.0, which adds an RTDX host-side Java API and is supported by Code Composer Studio versions starting with 3.3.

Another variation of the technology is **High-speed RTDX** (HSRTDX), which operates in conjunction with an additional silicon module on the DSP, the so-called HSRTDX unit, and requires an emulation controller of the XDS560 class. The diagram of the basic HSRTDX target architecture in Figure 2.10 shows the additional components present between the two connected systems. Compared to the standard RTDX bandwidth of about 10 Kbytes/second, HSRTDX provides much higher data transfer rates (2 Mbytes/second), and an increased responsiveness of RTDX enabled host controlled applications [10]. The XDS560 emulator also supports standard RTDX, raising the transfer rates as high as 130 Kbytes/second, even when the HSRTDX unit is not available on the target processor [12]. When converting a standard RTDX-based DSP application to HSRTDX, one must take into account that HSRTDX is used with the interrupt-driven RTDX implementation, for which the HSRTDX unit reserves three interrupts, namely INT3, INT11 and INT12. Reference [10] guides the user through applying High-speed RTDX to a DSP application.

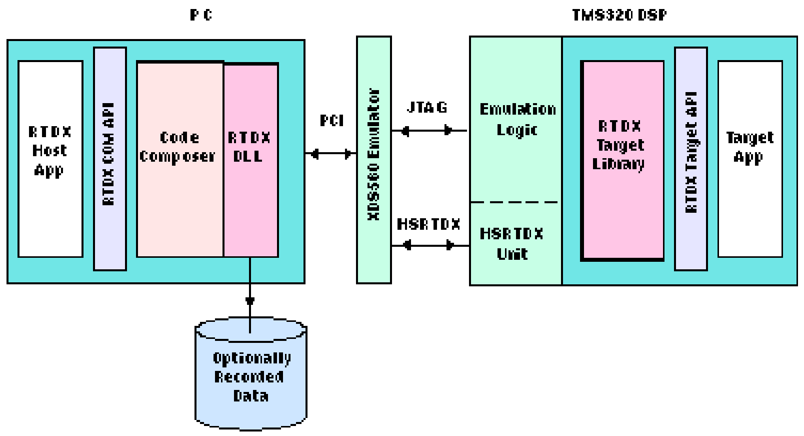


Figure 2.11 Basic components of the HSRTDX target architecture [11]

On both sides of the communication, the **RTDX API** exposes a series of functions for configuring RTDX, creating, enabling and opening channels, for sending and receiving data, as well as query functions to monitor the status of the transfers. A variety of data types can be transferred as single values or arrays in both directions and also as structures from the target to the host, including 8-bit, 16-bit or 32-bit integers, and 32-bit or 64-bit floating-point values. The target interface defines one read-function and one write-function for transfers of all data types. The two functions which require a named channel, a handle to the buffer to read from or write to and the size of this buffer are declared as follows:

```
int RTDX_read(RTDX_InputChannel *ichan, void *buffer, int bsize)
int RTDX_write(RTDX_outputChannel *ochan, void *buffer, int bsize)
```

On the host side, the COM interface defines specialized functions for each type and size of data to be read from and written to the RTDX library buffer. Functions sending and receiving single values take such forms as

```
long ReadI4( long * pData )
long WriteI4( long Data, long * numBytes )
```

where *pData indicates where a 4-byte integer is to be received and Data is the 4-byte integer being sent. To transfer data efficiently, arrays of all data types are placed into single dimensional SAFEARRAYs and bundled in VARIANT structures to be passed to the channels of both directions, VARIANT and SAFEARRAY being types defined by the

Automation library. Read-functions handling arrays on the host side resemble the following declaration applying to 2-byte integer arrays:

```
long ReadSAI2( VARIANT * pArr )
```

Transferring data on multiple channels simultaneously from the target to the host is also possible. Data is received in the host buffer in the order in which it was sent from the target, while the reading from the log file is done in sequence. This transferring scenario is not supported in the opposite direction, and write-functions on the host do not specify a channel name. Therefore transfers from host to target are done with one opened channel at a time. A complete documentation of the host and target RTDX APIs is available in reference [11].

The discussion of the software tools involved in the development of the RTDX Display application is carried out further, in relation to the first sketch of functionality requirements.

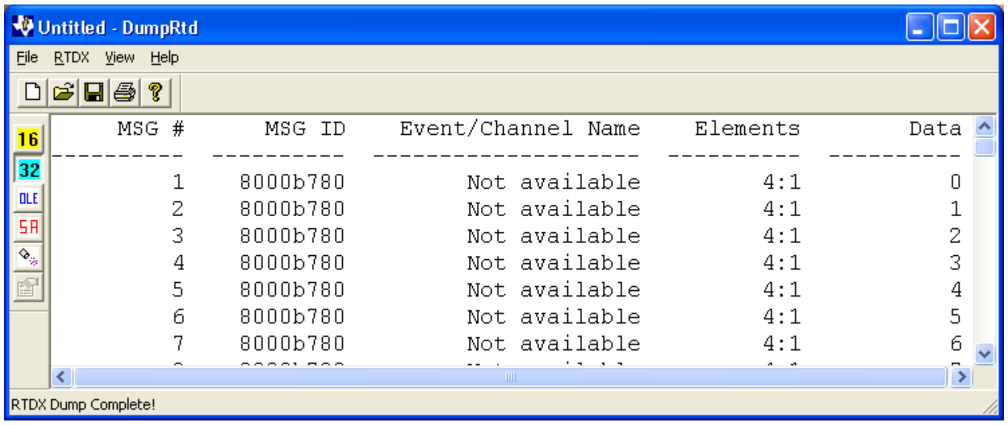
3 Analysis and Requirements of the Host Application

This chapter presents the tool options that DSP designers have at their disposal for use on the PC host side to display and control data from a target software. It introduces available host applications for RTDX, as well as graphical display tools appropriate for RTDX development, and it concludes with an analysis of the tool alternatives and a layout of requirements for the developed RTDX host application.

3.1 Texas Instruments Host Applications for Real-Time Data Exchange

Two basic host applications provided by Texas Instruments as part of the Code Composer Studio package are discussed here, to present the features they offer in support of Real-Time Data Exchange. The RTDX Dump Utility and the RTDX General-Purpose Display can be accessed from the directory ccsinstall\examples\hostapps\rtdx.

The RTDX Dump Utility (dumprtd.exe) serves both as an RTDX COM client for live data, and as a log file dumper for non-real-time data viewing. It was created using Microsoft Visual C++, it supports 16-bit and 32-bit TI processors, and it offers a display for target data only. In raw log file dump mode, it does not require an active connection to CCS. The screenshot in Figure 3.1 shows the Dump Utility being used to read converted data from a pre-recorded binary log file (.rtd). The display lists the data in column format. Even in live mode, the tool is not equipped for host-to-target transfer.

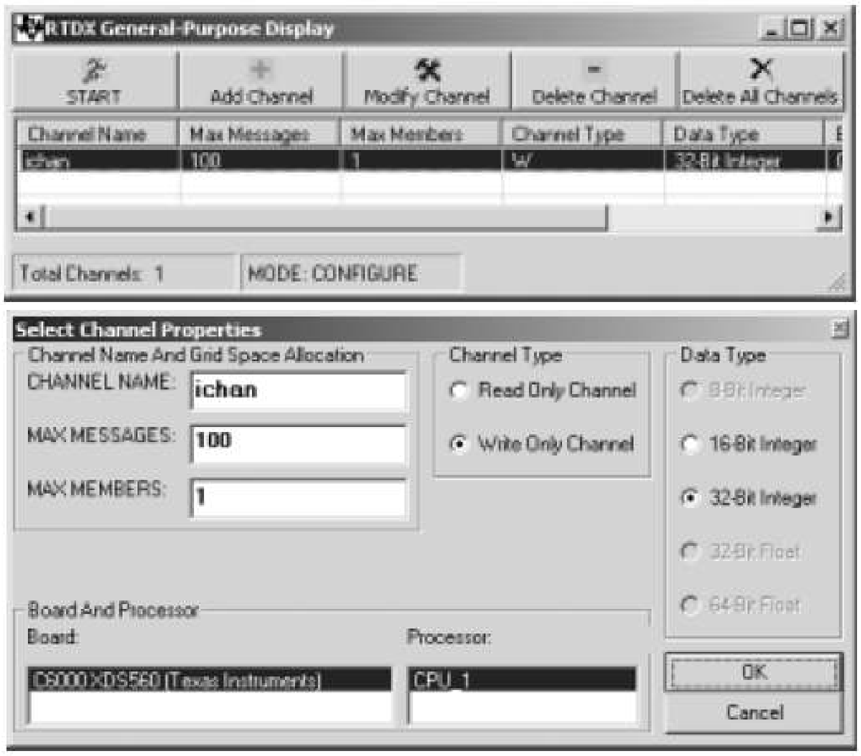


```
     MSG #      MSG ID     Event/Channel Name   Elements       Data
----------  ----------  --------------------  ----------  ----------
         1    8000b780         Not available         4:1           0
         2    8000b780         Not available         4:1           1
         3    8000b780         Not available         4:1           2
         4    8000b780         Not available         4:1           3
         5    8000b780         Not available         4:1           4
         6    8000b780         Not available         4:1           5
         7    8000b780         Not available         4:1           6
```

RTDX Dump Complete!

Figure 3.1 Reading an RTDX log file with the Dump Utility

The RTDX General-Purpose Display (gpdprog.exe) enables the transfer of integer and float type values, mainly from target to host, on any number of channels. It was built using Microsoft Visual Basic, and is able to support High-speed RTDX as well. In order to set up a transfer, the user must specify the channels as declared in the target application, and several parameters of the DSP board. Data in displayed in a spreadsheet-like window, in which the number of rows is determined by the total number of messages written to the selected channel, while the columns correspond to the number of members of each message, as in the case of transferring arrays or C structs. Channels are read-only or write-only, and each channel is visualized in its individual window.



Channel Name	Max Messages	Max Members	Channel Type	Data Type	E
ichan	100	1	W	32-Bit Integer	(

Total Channels: 1

MODE: CONFIGURE

Channel Name And Grid Space Allocation

CHANNEL NAME: ichan

MAX MESSAGES: 100

MAX MEMBERS: 1

Channel Type

Read Only Channel [✓ Write Only Channel]

Data Type

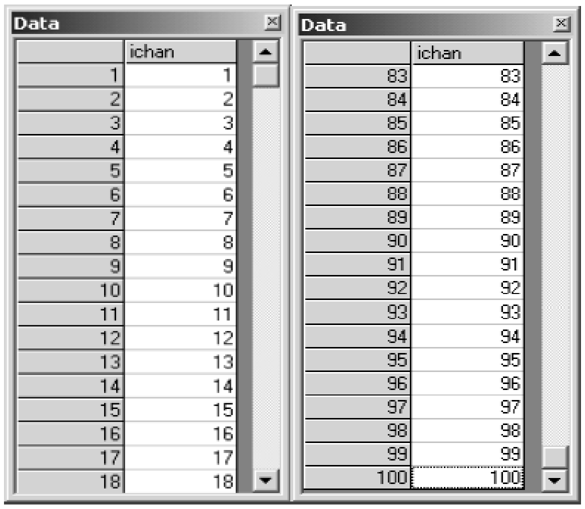
8-Bit Integer
16-Bit Integer
[✓ 32-Bit Integer]
32-Bit Float
64-Bit Float

Board And Processor

Board: C6000 XDS560 (Texas Instruments)

Processor: CPU_1

Figure 3.2 General-Purpose Display setup [10]



	ichan
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	16
17	17
18	18

	ichan
83	83
84	84
85	85
86	86
87	87
88	88
89	89
90	90
91	91
92	92
93	93
94	94
95	95
96	96
97	97
98	98
99	99
100	100

Figure 3.3 General-Purpose Display channel data [10]

For host-to-target transfer, the spreadsheet offers the single option of sending automatically generated values, starting with 0 and incremented by 1, up to any specified number of values, limited only by the size of the RTDX buffers. The user cannot otherwise edit the data on a host-to-target channel. Figures 3.2 and 3.3 show the setup of the General-

Purpose Display with a host-to-target channel, and the automatically generated data on that channel.

3.2 Tools for Developing RTDX Host Applications

The DSP developer who wishes to have more flexibility in the host application than provided by the two TI basic hosts, can stream RTDX data into a number of COM compliant visualization packages. Microsoft Visual C++ and Visual Basic applications can be built to use the RTDX host interface according to the requirements defined by the user. A Microsoft Excel RTDX COM client can be built as well, using Excel and Visual Basic for Applications (VBA, also available within Excel), to produce a spreadsheet style host application. Alternatively, MATLAB from The MathWorks can be used to connect through RTDX to a DSP application. MATLAB is a high-performance language and environment for technical computing. It provides a function plotting mechanism which can be a useful component to an RTDX host application, and GUI building features that can aid the developer in displaying RTDX data. When CCS v.3.3 is used, the supported standard RTDX v.2.0 offers a Java host interface, which extends further the developer’s choice of IDEs.

In addition to programmatically developing the host application, the user can also choose from several RTDX compatible graphical tool packages. Some of these packages can enhance the host application user interface, and others can even eliminate the need for programming it, by providing a complete graphical development environment. The most popular of them are briefly introduced here.

Quinn-Curtis’ Real-Time Graphics Tools contain a collection of real-time graphics and user interface routines, oriented toward real-time applications requiring fast display and updating of data. This product was designed for development of Windows applications using Visual C++. The main part of this software is provided as a dynamic link library (DLL), and a small part comes as C or C++ source code, which must be compiled and linked with a user application. By incorporating this toolkit in a Visual C++ RTDX host client, the DSP developer can build instrument interfaces for handling data on RTDX channels, or build editable graphs, real-time plots, and exchange images between the host and other applications.

Another popular software for testing DSP performance with RTDX is National Instruments’ LabVIEW, a graphical development environment with configuration utilities designed specifically for test and measurement applications. A LabVIEW program consists of two parts: the front panel, a user interface containing control inputs and graphs, and the block diagram, where the graphical code is developed. LabVIEW has the capacity to establish communication with the DSP through an RTDX toolkit. Channels must be created in the DSP application inside CCS, and the user must specify the names of the configured channels in the host LabVIEW application to be able to view and modify them in the front panel. Figure 3.4 shows a block diagram of an RTDX enabled LabVIEW program.

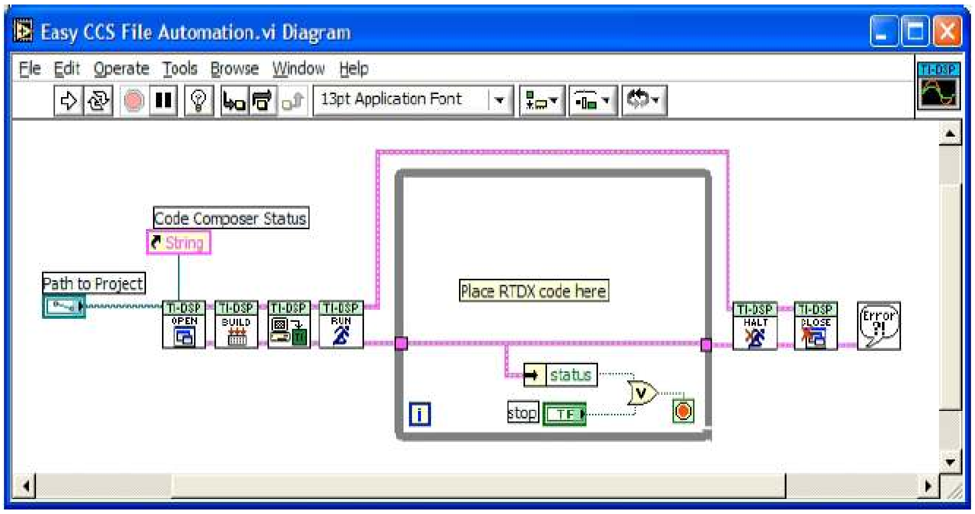


Figure 3.4 Basic DSP test system architecture using LabVIEW [13]

3.3 Requirements of the Developed RTDX Host Application

The preliminary considerations regarding the planned RTDX host application resulted in the following list of functionality requirements:

- The host application must apply to all DSP programs using the C language and executed on the given C6713 processor, so that the user does not need to modify it and recompile it, regardless of the number of RTDX channels involved and of the DSP program design.
- The user must be able to decide which target program variables are to be used for data transfer.
- The user should choose the form of graphical display for each target program variable that was marked for data transfer; particularly, a display possibility for large size buffers of data is required.
- Data transfers for 2-byte and 4-byte integers need to be supported, while no float value support is necessary.
- The design should allow the host client to read the initial values of the selected variables from the target application, display them, enable the user to modify them on the spot, and finally to send them from the host back to the target.
- For buffer type variables, data transfers from the target to the host should also be possible.
- The user must have the option to control from the host client when the data transfers for each variable occur.

The laboratory resources available for the development and future maintenance of the host application were summarized as follows:

- Version 3.1 of Code Composer Studio is mainly used for DSP application development, while currently migrating to v.3.3; RTDX v.1.0 accompanies CCS v.3.1.

- The DSK boards which the RTDX host client is meant to assist use the JTAG emulation for standard RTDX communication; a possible future employment of HSRTDX would allow for an easy conversion of the host client from standard RTDX.
- The Visual Studio 6 IDE is used for C and C++, PC based application development, and a future migration to a later version is presumed.
- The MATLAB IDE is the tool of choice for data analysis, visualization and simulation.

Based on the formulated requirements and listed tools at hand, an evaluation of the options for the desired RTDX host application is now possible. The readily available TI host applications offer very limited features, which do not meet the expected level of flexibility from the RTDX host client. The previously discussed graphical environments and toolkits would only enhance the graphical interface of the RTDX host client. An application built in LabVIEW has the benefit of requiring a much shorter development time, but it would maintain still a level of dependence on the target application, since channel names would have to be specified and the host client would have to be readapted for each individual DSP application.

A study of the laid out functionality requirements, and an analysis of the commercially available development tools and supporting laboratory resources, led to the following technical requirements for the constructed RTDX host application:

- A customized application is needed to meet the requirements of the RTDX host client.
- Visual Studio 6 and C++ are to be used for the development, enhanced by the Microsoft Foundation Class Library (MFC) v. 7.0.
- The C language target application needs to specify within its code which variables are to take part in RTDX transfers, the direction of the transfers and the format in which to have data displayed.
- At startup, the host client should obtain the instructions regarding the marked variables from the target program and display their initial values in the requested graphical format.
- The target application needs to declare the required RTDX channels, and the host application must match the same channel structure after reading at startup the instructions from the target.
- The host client GUI must be able to build itself according to the information read from the target program, namely to adjust its content and size.
- GUI controls that the host should be able to provide, include: slider, push button, radio button, check box and edit box; for arrays beyond a certain size, a separate window should display an appropriate large size edit box.
- For variables requiring transfer channels in both directions, data must be displayed in the same control on the GUI, to allow for easy data modification and resending.
- Support for transfers of 2-byte and 4-byte integer data must be provided.

4 Design of the RTDX Display Application

Described herein is the design strategy for the RTDX Display application, starting with a view of the entire system as determined by the traced out requirements, and followed by a detailed account of the application structure.

4.1 System Design

Figure 4.1 illustrates the two-platform system involved in DSP development, in which the TI Code Composer Studio IDE is used on the PC side to produce the executable file running on the DSP. The RTDX technology is employed to provide real-time communication between the two processors, according to the concept described in chapter 2.2. The host client for visualizing the exchanged data is represented by the RTDX Display application. As marked on the diagram, RTDX Display acquires its defining information at startup through a simple file reading procedure. Within the DSP program, the global declaration section of the .c source file supplies the self-adjusting host application with all the details of the developer's needs for monitoring the target execution. The further exchange of live data between RTDX Display and the target application occurs as described by the scheme of the RTDX data flow presented earlier. The implementation makes use of an RTDX log file for reading the transferred data.

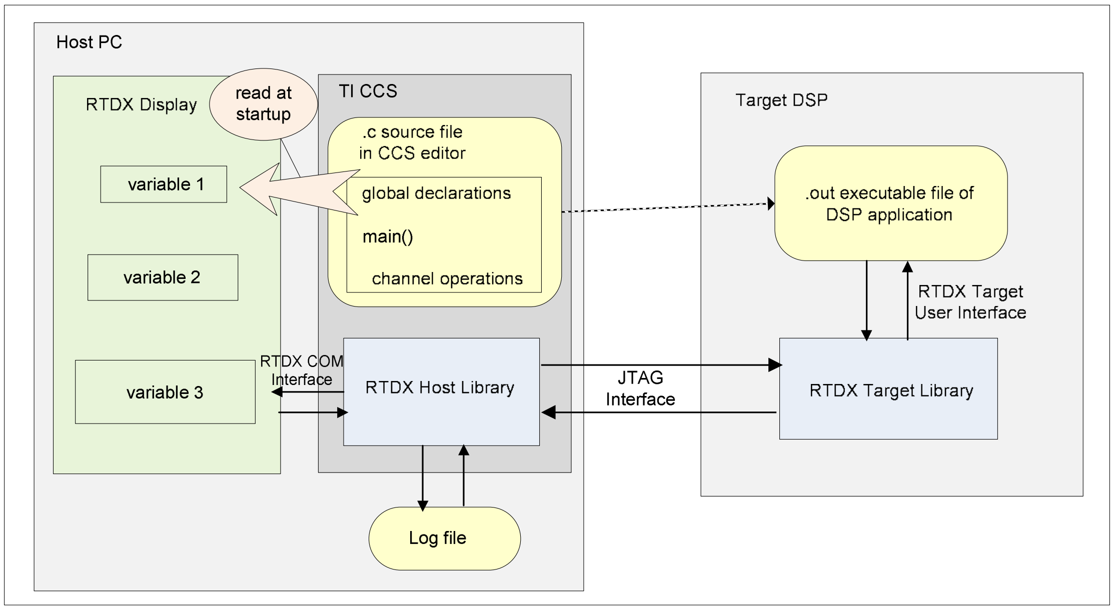


Figure 4.1 System Design including the RTDX Display

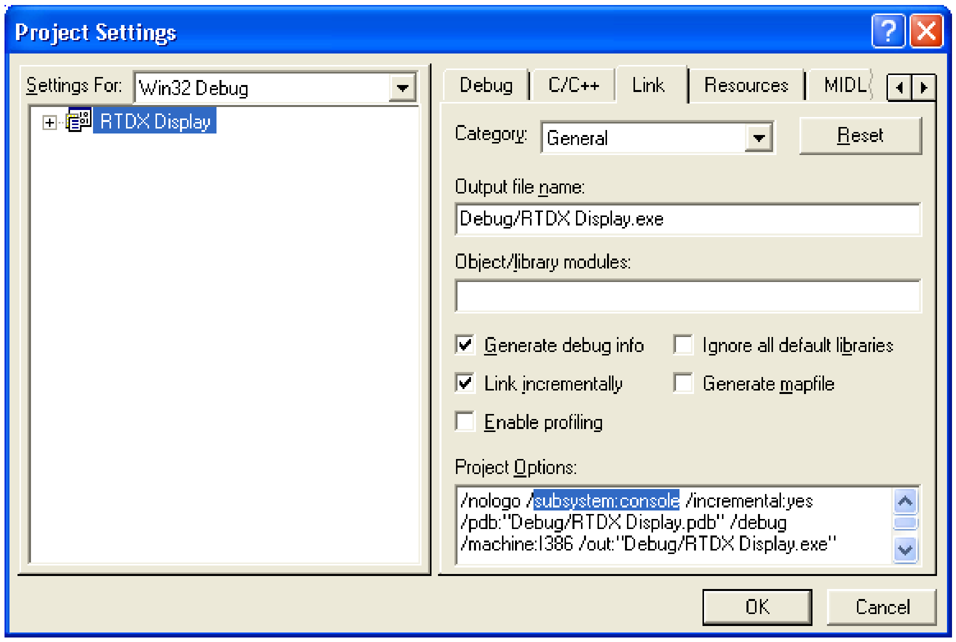
4.2 Design of the Host Application

RTDX Display was developed as a dialog-based MFC application in the Visual Studio 6 IDE. The Microsoft Foundation Classes library is a collection of pre-written C++ classes which assist the developer in building Windows applications efficiently. The MFC term "dialog-based" refers to applications containing at least one screen window, named dialog box and referred to as dialog. Two files automatically generated by the AppWizard constitute the core of the present application:

- RTDX Display.cpp, the main application source file that defines the application class CRTDXDisplayApp, which handles application startup and termination. A global CRTDXDisplayApp instance is created here.
- RTDX DisplayDlg.cpp, with the CRTDXDisplayDlg class, which defines the behavior of the application's main dialog

Other standard files of this dialog based project include: RTDX Display.rc, a resource script storing the dialog's template, Resource.h, a header file defining new resource IDs and StdAfx.h, a header file for standard system include files that provide the MFC support structure.

In order to ease the debugging process during development and to provide a convenient method of controlling the RTDX data transfer when using a debug configuration of RTDX Display, a console otherwise not present in an MFC application is added to the project. The "subsystem" linker option is changed from "windows" to "console", as shown in Figure 4.2.



Settings For: Win32 Debug

RTDX Display

Category: General

Output file name: Debug/RTDX Display.exe

Object/library modules:

[✓] Generate debug info
[ ] Ignore all default libraries
[✓] Link incrementally
[ ] Generate mapfile
[ ] Enable profiling

Project Options:

```
/nologo /subsystem:console /incremental:yes
/pdb:"Debug/RTDX Display.pdb" /debug
/machine:I386 /out:"Debug/RTDX Display.exe"
```

Figure 4.2 Introducing a console into an MFC application

This option indicates to the operating system how the project executable is to be run and it determines the starting address for the program. The project adds the file MFCConsole.cpp which provides the main() function required by the chosen linker option. To restore the regular entry point of the Windows application, the control flow is passed to WinMain(), a Windows API function called in the startup path of an MFC program to achieve the duty of a C main() function. Debugging code that directs its output to the console can be inserted with the preprocessor #ifdef DEBUG statement. A release configuration will exclude these code sections and the console can be deactivated by resetting the “subsystem” linker option to “windows”. This will cause the linker to ignore the main() function in MFCConsole.cpp.

Before laying out an appropriate design for the self-adjusting host application, the initial defining information about the target data to be transferred later through RTDX is described in detail. To allow for an automatic interpretation of the requests specified in the target program code, the user is instructed to note the following standardized format when setting up the target C source file:

- All variables intended for RTDX transfer must be declared globally and before the preprocessor directives.
- The list of variables must be framed by the opening and closing comment lines //START RTDX and //END RTDX. This section of the target code is referred to as the “RTDX block”.
- Only the five control types announced in the requirements can be recognized by RTDX Display: edit box, slider push button, radio button and check box.
- Edit box controls can deal only with array variables, i.e. single values must request other control types.
- The RTDX channels are labeled in accordance with the convention used by the RTDX API, namely from the perspective of the target application: in-channels for PC to DSP transfer and out-channels for DSP to PC transfer.
- All variable types can attach an in-type channel, while out-type channels are only supported for array variables.
- Array declarations must always specify the array size, and preprocessor constants are not recognized for this purpose.
- Only one variable should be declared per line.
- Uninitialized variables are allowed.
- Fully initialized arrays of large sizes and may split the declaration on several lines.
- Every variable declaration must be followed by a comment on the same line (for an array, on the last line of its declaration), indicating the direction of the channels connecting it with the host, and the associated control type to be used in the host application.

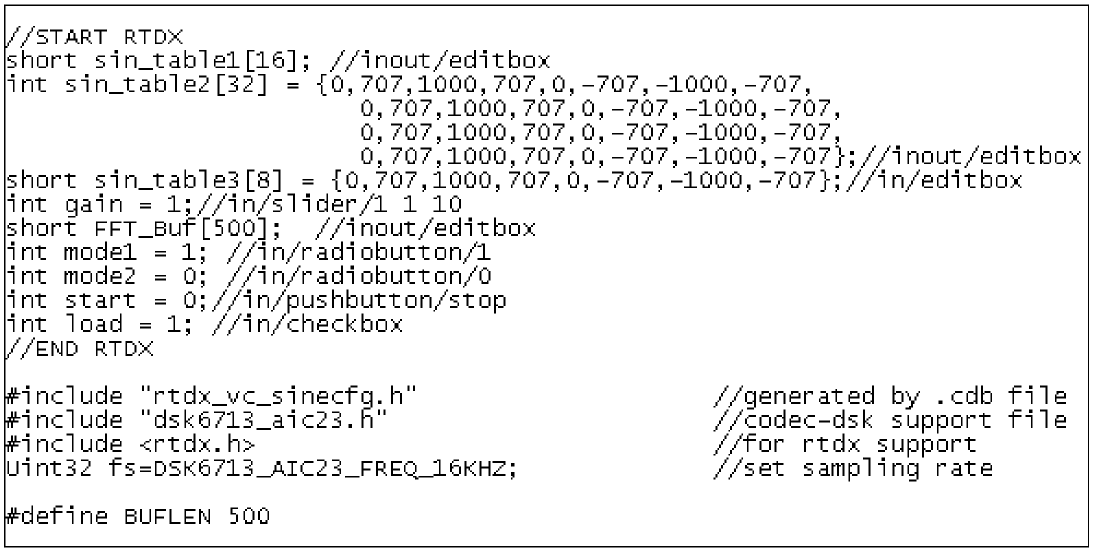
The top section of a C source file containing an RTDX block is extracted in Figure 4.3. The comments accompanying variable declarations in the RTDX block are expected to have the following format:

- For an edit box: //in/editbox i.e. channel/control. The channel type may also be “out”, or “inout” when the target declares both an in-channel and an out-channel.
- For a slider: //in/slider/1 1 10 i.e. channel/control/range. The range is expressed as: start of range, increment, end of range. Here, the indicated range is between 1 and 10, with increments of 1.

- For a push button: //in/pushbutton/stop i.e. channel/control/2nd name. The second name is a label for the opposite state of the push button and is optional.
- For a radio button: //in/radiobutton/1 i.e. channel/control/group_flag. As MFC radio buttons function only in groups, at least two radio button declarations should exist in one group, and they should be declared on adjacent lines. The group flag sets the first radio button of a group to one and the others to 0.
- For a check box: //in/checkbox i.e. channel/control.

The startup state of each control results from the value to which the mapping variable is initialized. Additional information regarding the RTDX block is given in the Appendix RTDX Display User's Guide.

Since RTDX channels must be named the same way in both host and target applications, RTDX Display expects that the channels on the target are named "in_channelx" and "out_channelx", where x is the sequence number of the corresponding variable in the RTDX block (e.g. "out_channel3" for the variable declared third).



```
//START RTDX
short sin_table1[16]; //inout/editbox
int sin_table2[32] = {0,707,1000,707,0,-707,-1000,-707,
                      0,707,1000,707,0,-707,-1000,-707,
                      0,707,1000,707,0,-707,-1000,-707,
                      0,707,1000,707,0,-707,-1000,-707};//inout/editbox
short sin_table3[8] = {0,707,1000,707,0,-707,-1000,-707};//in/editbox
int gain = 1;//in/slider/1 1 10
short FFT_Buf[500];  //inout/editbox
int mode1 = 1; //in/radiobutton/1
int mode2 = 0; //in/radiobutton/0
int start = 0;//in/pushbutton/stop
int load = 1; //in/checkbox
//END RTDX

#include "rtdx_vc_sinecfg.h"                      //generated by .cdb file
#include "dsk6713_aic23.h"                        //codec-dsk support file
#include <rtdx.h>                                 //for rtdx support
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ;               //set sampling rate

#define BUFLEN 500
```

Figure 4.3 Top section of a C source file showing an RTDX block

Another decisive factor for the application design is the strategy used in the transferring of data through RTDX. In this respect, two aspects are considered: the assignment of data either to individual or to collective channels, and the timing of channel enabling for transfers.

Sending target data from all variables in one single package to the host avoids transporting overhead bytes on the target, as previously discussed in chapter 2.2. However, this is possible only when all transferred variables are of the same data type, packaged either as an array or as a structure, since the host interface provides only read-functions for data of one type on each channel. At the same time, since the target-to-host transfers apply to arrays only in the current implementation and their number and size are not known in advance, handling all array transfers collectively would assume that possibly very large buffers of values are unnecessarily transferred at every modification of one single array. Therefore it is more advantageous to perform target-to-host data transfers on individual channels. For host-to-target transfers, RTDX does not support the packaging of data in structures. Variables of the same type could be sent to the target as a collective array and unpacked correspondingly in the target application. RTDX Display transfers variables individually from host to target as well.

The choice of multiple host-to-target channels of different data types in the application has an impact on the enabling strategy of in-channels (going to the DSP).The write-functions of the RTDX host interface do not apply to specific channels, instead they send data to the host library, where data waits for a read-request from the target (see host-to-target data transfers described in chapter 2.2). In order to keep host-leaving data mapped to its channel, only one enabled channel can exist in the application at one time.

Proceeding from the presented requirements, the class structure illustrated by the class diagram in Figure 4.4 was chosen. Class and sequence diagrams included in this thesis are generated using the Borland Together Unified Modeling Language (UML) tool (v. 6.1). The class diagram employs the UML graphical notation to show the data members and class composition relationships. The general design of classes follows the data encapsulation concept of object oriented programming (OOP). Private data members are accessed through standard read and write public functions. The defining features of the implemented classes are described next.

Class CRTDXDisplayApp is derived from CWinApp, the MFC base class for Windows application objects, whose InitInstance() member function of CWinApp is overridden to create the CRTDXDisplayDlg dialog object and load it.

The CRTDXDisplayDlg class inherits CDialog, the parent class of dialog boxes. CDialog provides the framework for handling Windows notification messages received by the dialog from its controls, as a result of the user's actions. MFC provides a message map facility consisting of a set of macros which connect Windows messages to specific message handler functions. The message map entries and message handler member functions corresponding to the selected Windows messages are generated by the ClassWizard. The CRTDXDisplayDlg class overrides the OnInitDialog() function of its base class to handle all dialog initialization, including obtaining the defining information from the target application and building the appropriate GUI screen. The class declares a message map for the handlers which apply to the supported control types.

To link the RTDX technology into the application, class IRtdxExp, which makes available the RTDX client API, is generated from the rtdxint.dll type library with the help of the ClassWizard. As an IDispatch wrapper class derived from the COleDispatchDriver class, IRtdxExp automates the host application as described in chapter 2.2, providing the two-way communication between RTDX Display and the CCS debugger. An IRtdxExp instance is used by the dialog to establish the connection with the DSP and to perform all RTDX related operations.

For the identification of the C source file to be opened, the user is requested to enter the file path in an input box generated by RTDX Display before initializing the main application

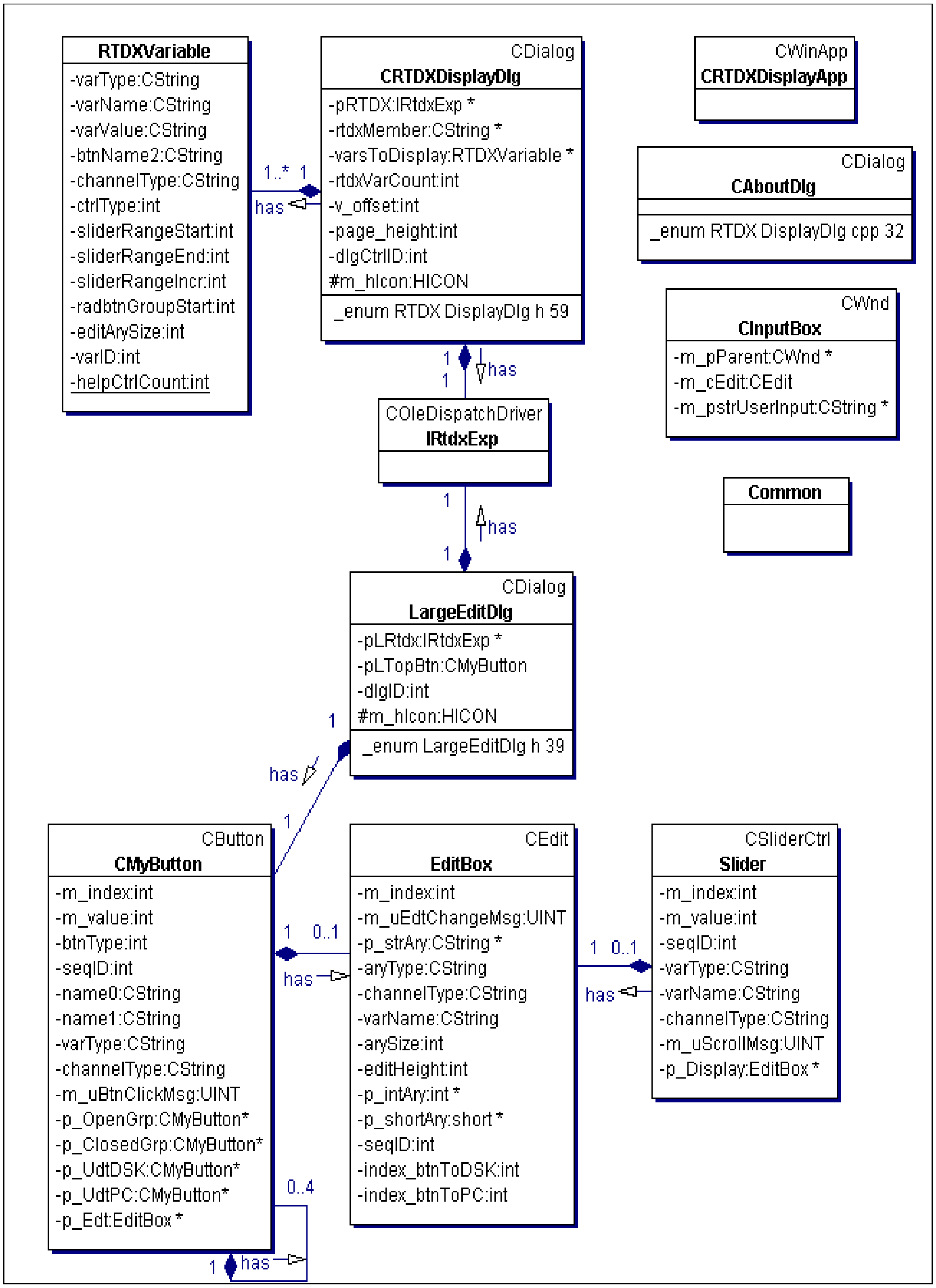


Figure 4.4 RTDX Display class diagram showing data members and class relationships

dialog. This design is advantageous, as it does not enforce a precise relative location of the two communicating applications. The process can be shortened by storing the last successfully opened file path and displaying it as the default option in the input box. For this purpose, class CInputBox derived from the MFC class CWnd is constructed to take user input in an edit box and to pass it to the main dialog.

Class RTDXVariable groups the information collected at startup from the RTDX block, including variable definition, channel type and control type. The dialog keeps an array of RTDXVariable objects to use in building the GUI.

For the representation of the possible controls to be handled by the dialog, three classes are created by inheriting corresponding MFC control type classes: class Slider, derived from CSliderCtrl, class EditBox, derived from CEdit and class CMyButton, derived from CButton. Push buttons, radio buttons and check boxes are constructed as objects of type CButton specifying different styles. According to the requirements, all classes representing controls handle storage and data transfers for variables of types short and int.
Class Slider holds defining information about a slider object. To provide a way to visualize the current value of a Slider object (particularly useful for sliders with a large number of increments), class Slider references a read-only edit box which is updated along with the movement of the slider.

Class EditBox supports only array variables, for which it keeps two parallel arrays, in string format for display purposes, and in the specified numeric (short or int) for transferring on the RTDX channels. To trigger the transfers of edit box data, an EditBox object requires one attached button for each direction of data transfer. For uninitialized arrays declared in the target application, an empty edit box is created. Depending on the DSP program concept, the user can either manually enter values for the respective array and send them to the target, or acquire live data from the DSP program, with the help of the corresponding transfer button. The attached transfer buttons are enabled or disabled to match the transfer directions allowed by the declared channels, and by the availability of data in the edit box. For space management reasons, arrays above a set maximum size are displayed in a separate dialog handled by class LargeEditDlg.

Class LargeEditDlg builds a new dialog for the display of a large size edit box. To realize this, the large edit box and its accompanying objects are created and initialized in the main dialog, where they remain invisible. The LargeEditDlg instance takes two pointers from the main dialog at construction time, pointers which supply the IRtdxExp object and the invisible edit box structure. The large dialog creates local objects for the edit box and its buttons which are initialized to the state of the original objects they mirror. When closing, the large dialog stores all changes made in the local objects into their invisible ones in the main dialog. This design allows the same data to be available for later instances of LargeEditDlg which deal with this edit box.

Each edit box and its attached transfer buttons are placed on a frame (of type CMyButton with group box style). The frame of a large edit box is displayed in closed form, and attempting to open it triggers the creation of the new dialog.

Instances of class CMyButton are implemented to support controls of type push button, radio button, check box or group box. For push buttons associated with edit boxes, the class holds four references of its own type to provide for the control structure related to an edit box. Within the same structure, a button instance uses a reference to an edit box.

The constructed pointer structure dealing with the displaying and transferring of data to and from an edit box can be seen in the diagram in Figure 4.5. The button controlling the expansion of the group box is placed at the structure top and it contains references to all dependent objects. The LargeEditDlg dialog holds a reference to this button to have access to the entire structure it must build for a large edit box.

A model of the resulting screen layout of RTDX Display is sketched in Figure 4.6.

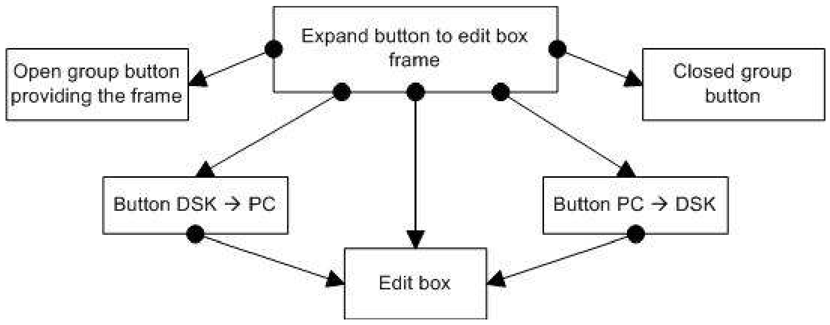

Expand button to edit box frame
Open group button providing the frame
Closed group button
Button DSK → PC
Button PC → DSK
Edit box

Figure 4.5 Pointer structure handling edit box operations

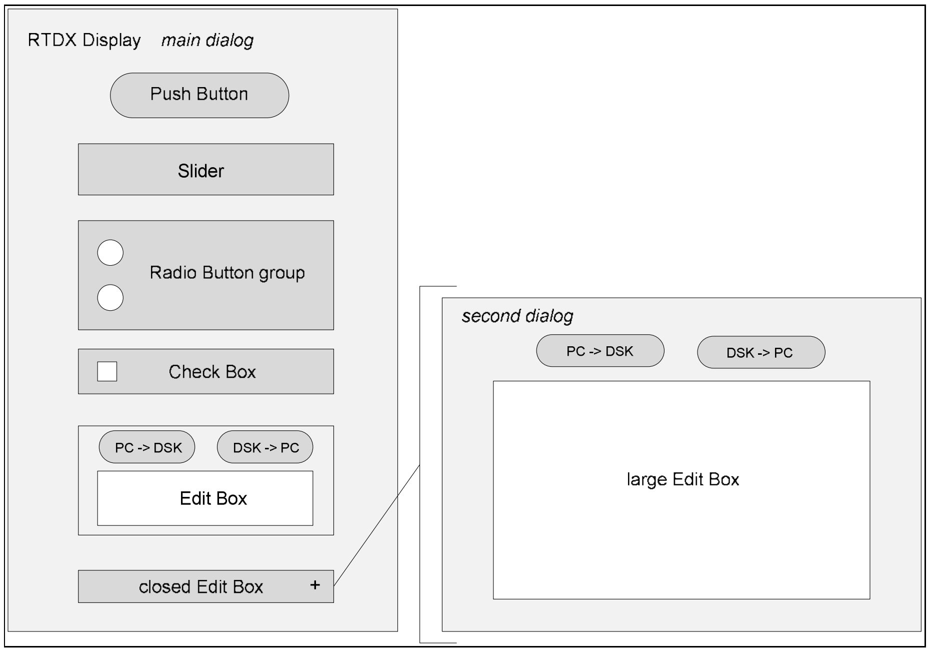


Figure 4.6 RTDX Display GUI layout model

Class Common serves as a utility class, grouping static functions for handling string operations and type conversion of array members between numeric and string representation. These functions are employed in reading the RTDX block, as well as in manipulating the values displayed by an edit box.

Class CAboutDlg is generated as part of RTDX DisplayDlg.cpp and it displays an “About” message dialog typical of window-based applications.

The current design of RTDX Display does not support reading from the RTDX log file in Playback mode, as the initial data displayed by the host application is extracted from the RTDX block of the DSP program, and the loading of processed data from the log file to the edit boxes through the DSK->PC buttons (seen in Figure 4.6) requires an active connection to the DSP.

The following chapter presents the implementation of the application according to the outlined design concept.

5 Implementation of the RTDX Display Application

This chapter describes the implementation of the host application for Real-Time Data Exchange. The realization of its automatic generation based on the DSP application requirements is presented, followed by a discussion of the procedures achieving the transfer of data between the two communicating platforms. Guidelines for handling RTDX in the DSP program and an analysis of the performance of RTDX Display conclude the chapter.

5.1 Automatic GUI Screen Generation

The diagram in Figure 5.1 shows the sequence of processes involved in the automatic generation of the RTDX Display screen. The flow of tasks behind the individual illustrated blocks is detailed here in sequence.

1. Typically of an MFC dialog application, an instance of class CRTDXDisplayApp is created at the execution start.
2. As dictated by the “subsystem” linker option, a console is generated and the main() function defined in file MFCConsole.cpp is detected as the entry point to the project. From here the flow of control is directed to the WinMain() function of the Windows API to start building a Windows application.
3. The InitInstance() function of the application object instantiates the main dialog CRTDXDisplayDlg.
4. The dialog’s initialization, which takes place in the OnInitDialog() function, begins with creating an IRtdxExp object of type IDispatch to define the application as an RTDX Automation client. A connection to the DSP is established and RTDX is enabled for this connection.
5. A CInputBox instance is created, which loads the input box for entering the path of the DSP program source file.
6. If the file is successfully opened, its path is stored in a text file located in the RTDX Display project folder, to be loaded in the input box as the default path for the next application execution. A message box informs the user if the indicated file cannot be opened.
7. The RTDX block is read from the C source file and if unexpected content is detected, the user is informed to review the file setup. The individual variables are stored in the dialog as a two-dimensional array of the MFC CString type.
8. Objects of RTDXVariable type are created in an array from the acquired variable declarations, to provide throughout the application execution easy access to the user defined RTDX block information.
9. The last task of the dialog’s initialization is the function BuildGUI(), which constructs the required controls on the screen. A sequence diagram of this process follows in Figure 5.2. The types of controls to be created are obtained in a loop through the RTDXVariable array. To place the controls on the dialog, the vertical offset from the dialog top is recorded after each control is created. The screen size calculation is based on the final value of the vertical offset.

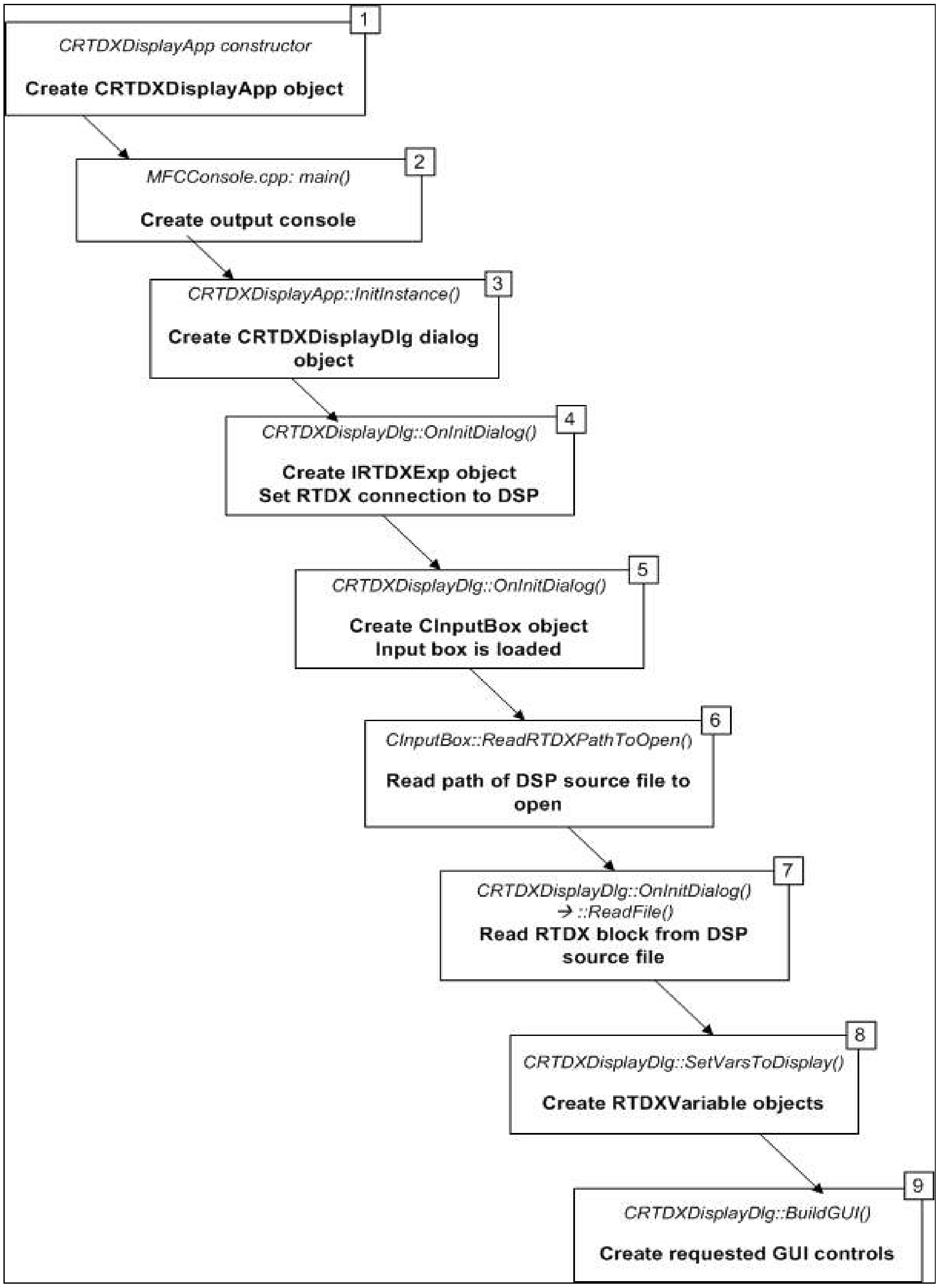

1
CRTDXDisplayApp constructor
Create CRTDXDisplayApp object
2
MFCConsole.cpp: main()
Create output console
3
CRTDXDisplayApp::InitInstance()
Create CRTDXDisplayDlg dialog object
4
CRTDXDisplayDlg::OnInitDialog()
Create IRTDXExp object
Set RTDX connection to DSP
5
CRTDXDisplayDlg::OnInitDialog()
Create CInputBox object
Input box is loaded
6
CInputBox::ReadRTDXPathToOpen()
Read path of DSP source file to open
7
CRTDXDisplayDlg::OnInitDialog()
→ ::ReadFile()
Read RTDX block from DSP source file
8
CRTDXDisplayDlg::SetVarsToDisplay()
Create RTDXVariable objects
9
CRTDXDisplayDlg::BuildGUI()
Create requested GUI controls

Figure 5.1 Flow diagram of the automatic GUI screen generation of RTDX Display

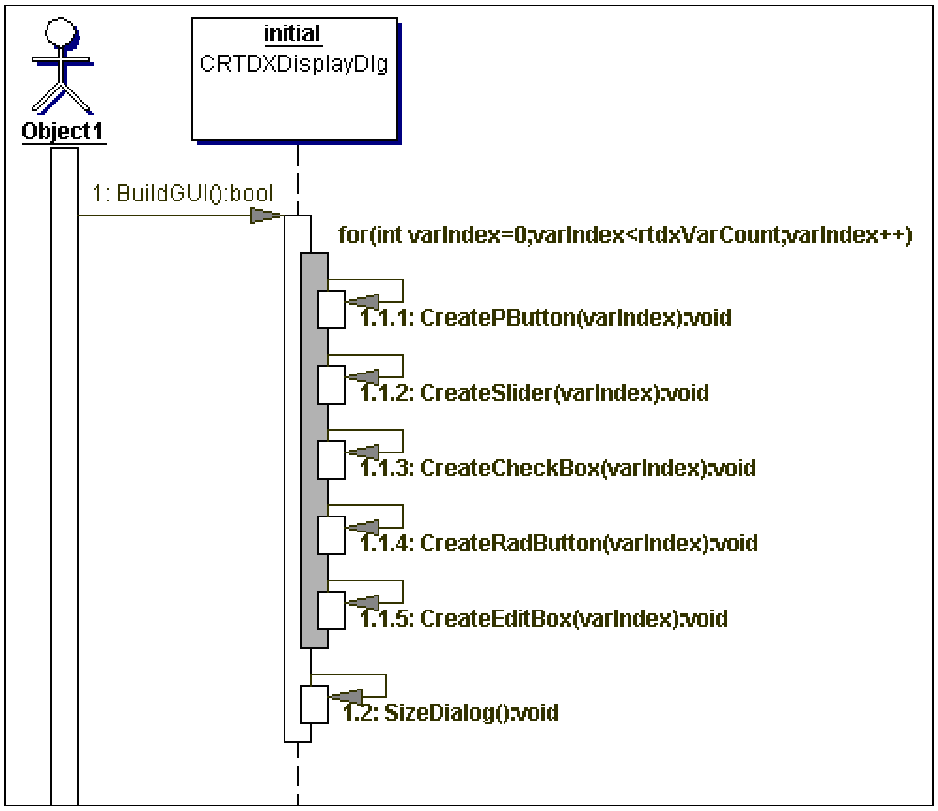

initial
CRTDXDisplayDlg
Object1
1: BuildGUI():bool
for(int varIndex=0;varIndex<rtdxVarCount;varIndex++)
1.1.1: CreatePButton(varIndex):void
1.1.2: CreateSlider(varIndex):void
1.1.3: CreateCheckBox(varIndex):void
1.1.4: CreateRadButton(varIndex):void
1.1.5: CreateEditBox(varIndex):void
1.2: SizeDialog():void

Figure 5.2 Sequence diagram of the BuildGUI() function

5.2 RTDX Realization in the Host Application

The host application controls the exchanging of data between the two platforms by initiating the transfers every time the user requests them. In the case of sliders, user action triggers the OnHScroll() message handler of a horizontal scroll bar or slider, where the transfer of slider values to the DSP application occurs. The active slider object is identified, the appropriate channel for host-to-target transfer is opened and enabled, data is sent on the channel and finally the channel is disabled, according to the chosen design. This message handler is executed once for every increment crossed by the slider moving action, therefore the transfer of slider values to the DSP consists of one data package for each increment change.

Handling of data transfers from edit box and button-type controls takes place in the dialog's OnMsgBtnClick () message handler which detects button-click actions. A sequence diagram of this implemented handler function is illustrated in Figure 5.3, in which the non-essential steps were hidden. The function starts by determining the type of button that Windows received the message from. For check boxes, radio buttons and push buttons other than the ones serving edit boxes, the transfer is implemented only from host to target, therefore a single variable is sent to the DSP after its channel name is built and the channel is enabled. In the case of edit box helper buttons, the referenced edit box object is found and the data package for the requested transfer is built. At this point, arrays are sent to the target or read from the incoming channel as a wrapping VARIANT type structure (as described in chapter 2.2). The channels are disabled before the message handler quits.

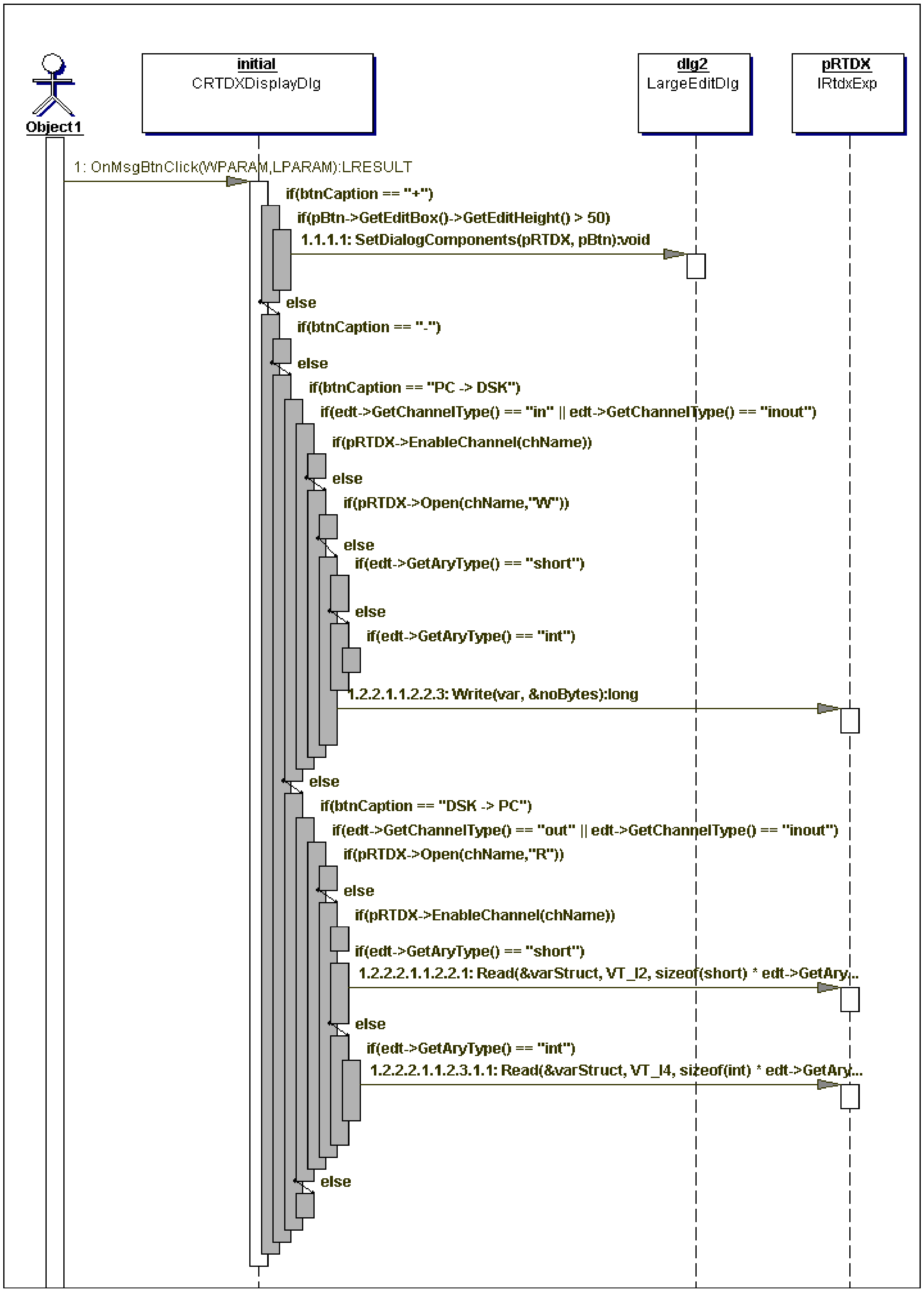

initial
CRTDXDisplayDlg
dlg2
LargeEditDlg
pRTDX
IRtdxExp
Object1
1: OnMsgBtnClick(WPARAM,LPARAM):LRESULT
if(btnCaption == "+")
if(pBtn->GetEditBox()->GetEditHeight() > 50)
1.1.1.1: SetDialogComponents(pRTDX, pBtn):void
else
if(btnCaption == "-")
else
if(btnCaption == "PC -> DSK")
if(edt->GetChannelType() == "in" || edt->GetChannelType() == "inout")
if(pRTDX->EnableChannel(chName))
else
if(pRTDX->Open(chName,"W"))
else
if(edt->GetAryType() == "short")
else
if(edt->GetAryType() == "int")
1.2.2.1.1.2.2.3: Write(var, &noBytes):long
else
if(btnCaption == "DSK -> PC")
if(edt->GetChannelType() == "out" || edt->GetChannelType() == "inout")
if(pRTDX->Open(chName,"R"))
else
if(pRTDX->EnableChannel(chName))
if(edt->GetAryType() == "short")
1.2.2.2.1.1.2.2.1: Read(&varStruct, VT_I2, sizeof(short) * edt->GetAry...
else
if(edt->GetAryType() == "int")
1.2.2.2.1.1.2.3.1.1: Read(&varStruct, VT_I4, sizeof(int) * edt->GetAry...
else

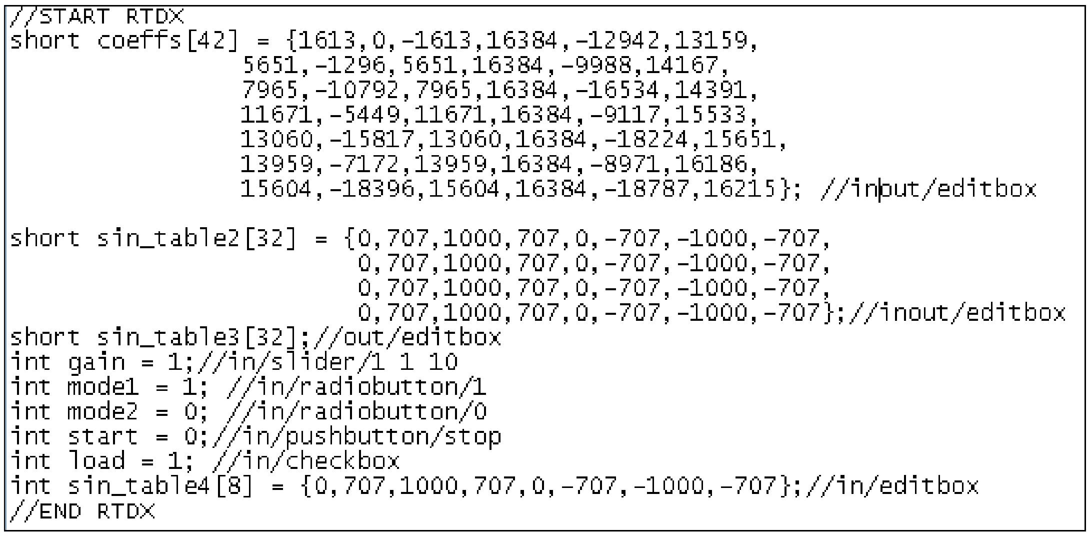
Figure 5.3 Partial sequence diagram of the message handler OnMsgBtnClick()

When the button message handler is generated from expand buttons of edit box frames, the creation of a LargeEditDlg dialog is initiated, whose SetDialogComponents() function is called to pass two pointers from the main dialog. The data transfers of large arrays follow the same procedure as in the case of small arrays. Changes made to the data in a large edit box are stored in the original edit box array from the main dialog, to be available for subsequent instances of the LargeEditDlg object.

In response to changes occurring in an edit box, whether through loading from the DSP, typing or pasting, the OnMsgEditChange() handler of the dialog is executed. The entered characters are verified and only numeric entries are allowed as positive or negative values, separated by blank spaces or tabs. For an array with an in-channel attached, the PC->DSK button is enabled if valid text is detected in the edit box and a data transfer to the target is possible.

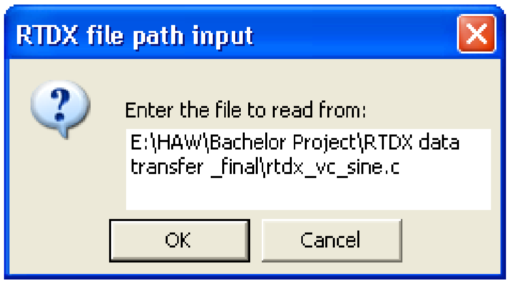
5.3 RTDX Display Performance

To test the functionality of the realized host application, a test DSP program is built with the RTDX block shown in Figure 5.4, which requests all supported controls. The path of the file is entered at startup by an input box, as seen in Figure 5.5. When executed, RTDX Display generates the screen captured in Figure 5.6. The variables listed in the given RTDX block have been assigned the appropriate controls and are laid out in the same sequence on the screen, showing variable names and initial values. Three large size edit boxes are represented by closed frames. Figure 5.7 shows an image of the opened *sin_table2* edit box, allowing data transfers in both directions (both transfer buttons are enabled), as indicated by the requested channel type. From the host application screen, the user can monitor and adjust the activity in the DSP program.



```
//START RTDX
short coeffs[42] = {1613,0,-1613,16384,-12942,13159,
                 5651,-1296,5651,16384,-9988,14167,
                 7965,-10792,7965,16384,-16534,14391,
                 11671,-5449,11671,16384,-9117,15533,
                 13060,-15817,13060,16384,-18224,15651,
                 13959,-7172,13959,16384,-8971,16186,
                 15604,-18396,15604,16384,-18787,16215}; //input/editbox

short sin_table2[32] = {0,707,1000,707,0,-707,-1000,-707,
                        0,707,1000,707,0,-707,-1000,-707,
                        0,707,1000,707,0,-707,-1000,-707,
                        0,707,1000,707,0,-707,-1000,-707};//inout/editbox
short sin_table3[32];//out/editbox
int gain = 1;//in/slider/1 1 10
int mode1 = 1; //in/radiobutton/1
int mode2 = 0; //in/radiobutton/0
int start = 0;//in/pushbutton/stop
int load = 1; //in/checkbox
int sin_table4[8] = {0,707,1000,707,0,-707,-1000,-707};//in/editbox
//END RTDX
```

Figure 5.4. RTDX block in DSP program

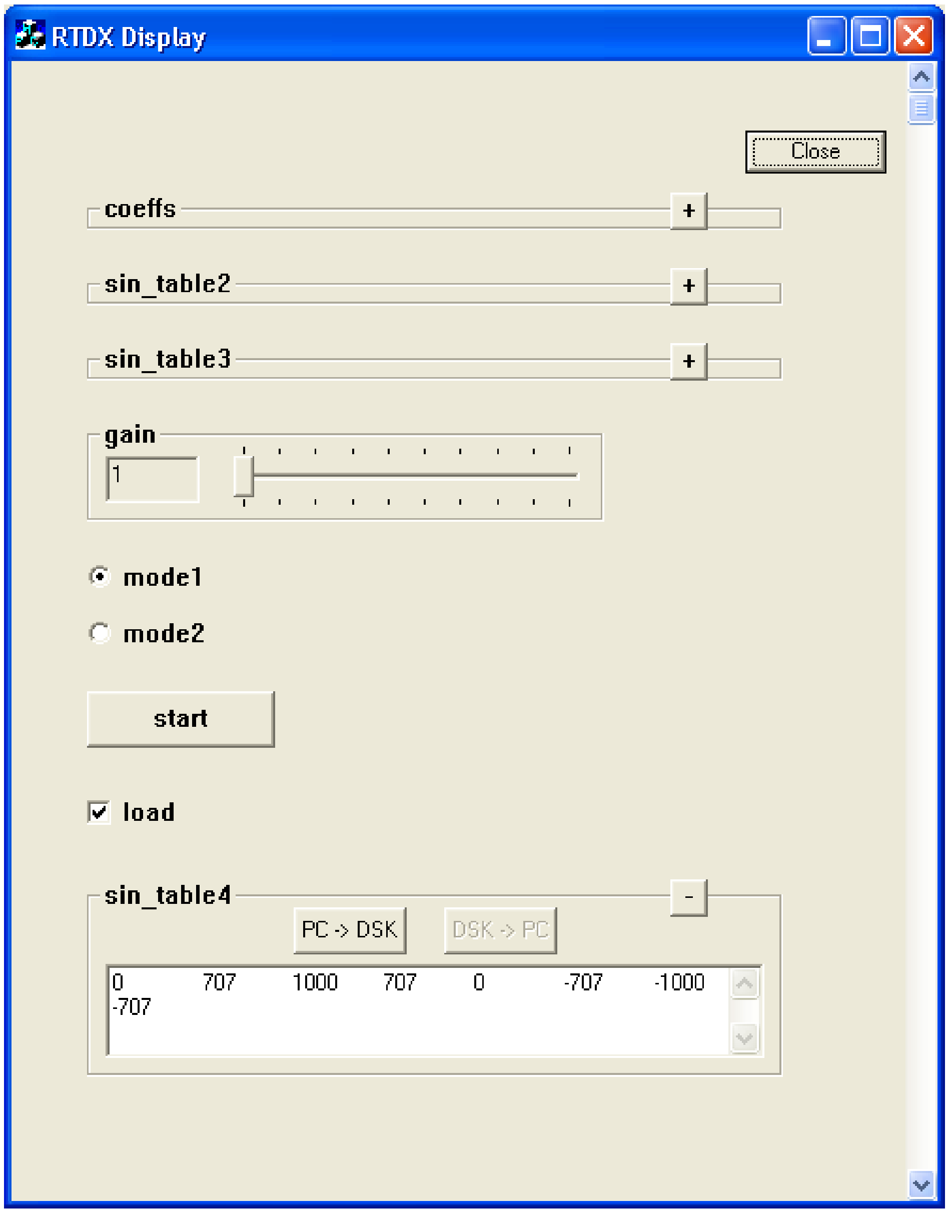


RTDX file path input

Enter the file to read from:

E:\HAW\Bachelor Project\RTDX data transfer _final\rtdx_vc_sine.c

Figure 5.5 Input box requesting the C source file path at startup



RTDX Display

gain: 1

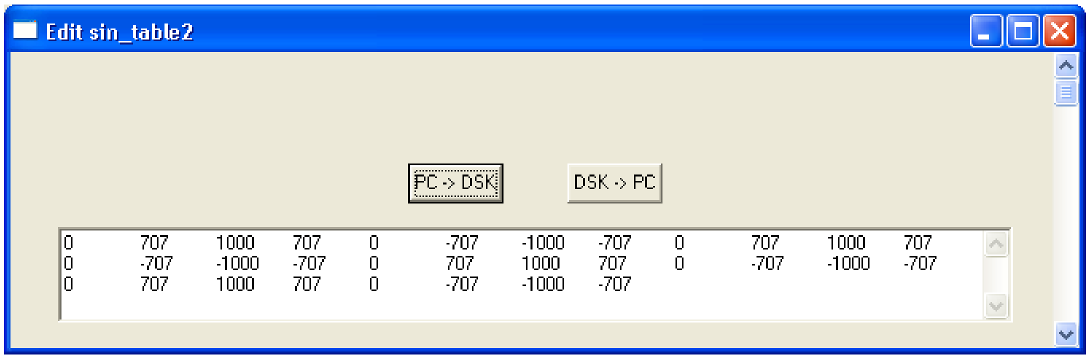
[✓ mode1] mode2

load ✓

sin_table4

```
0        707      1000     707      0        -707     -1000
-707
```

Figure 5.6 RTDX Display screen built from the RTDX block in Figure 5.4



```
0   707   1000   707   0   -707   -1000   -707   0   707   1000   707
0   -707  -1000  -707  0   707    1000    707    0   -707  -1000  -707
0   707   1000   707   0   -707   -1000   -707
```

Figure 5.7 Large edit box dialog with in-channel and out-channel

5.4 Guidelines for DSP Applications Using RTDX Display

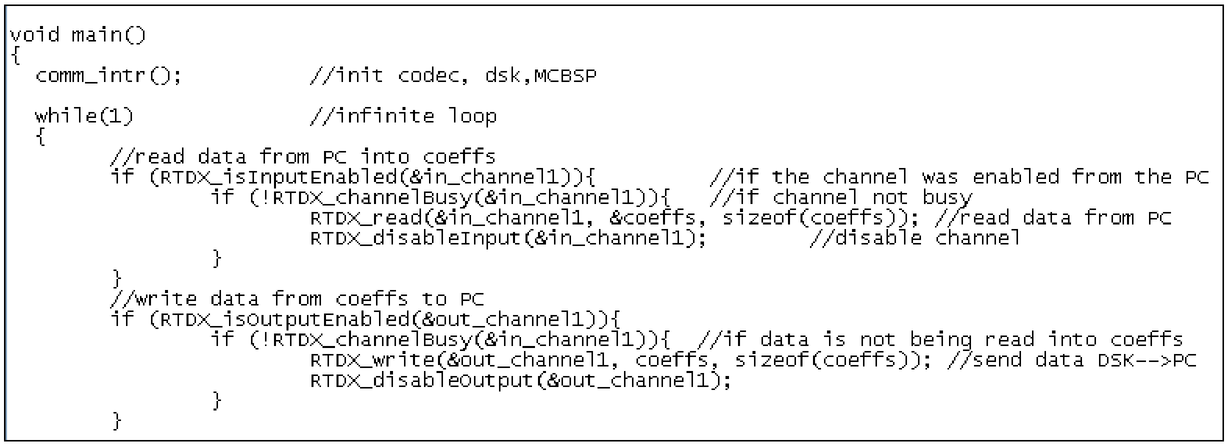
When introducing RTDX operations into a DSP application intended to communicate with RTDX Display, the user must adhere to a number of guidelines. As previously mentioned, RTDX channels are declared globally on the target as either input or output channels, and their names should follow the established naming conventions. The channel declarations required for the test case presented in this chapter are extracted in Figure 5.7. As requested by the design of the host application, channels should not be enabled on the target, as this task is handled on the host.

```
// create RTDX channels
RTDX_CreateInputChannel(in_channel1);   // input channel for coeffs
RTDX_CreateOutputChannel(out_channel1); // output channel for coeffs

RTDX_CreateInputChannel(in_channel2);   // input channel for sin_table2
RTDX_CreateOutputChannel(out_channel2); // output channel for sin_table2

RTDX_CreateOutputChannel(out_channel3); // output channel for sin_table3

RTDX_CreateInputChannel(in_channel4);   // input channel for gain
RTDX_CreateInputChannel(in_channel5);   // input channel for mode1
RTDX_CreateInputChannel(in_channel6);   // input channel for mode2
RTDX_CreateInputChannel(in_channel7);   // input channel for start
RTDX_CreateInputChannel(in_channel8);   // input channel for load

RTDX_CreateInputChannel(in_channel9);   // input channel for sin_table4
```

Figure 5.8 Channel declaration section in the target program

The test program used is interrupt-driven using INT11. In the main() function, an infinite loop executes data transfer statements on all declared channels. Upon interrupt, execution proceeds to the mapping ISR, where simple tasks are performed to serve the testing concept. The beginning of this infinite loop is presented in Figure 5.9 to illustrate the channel handling strategy. All reading and writing actions occur when the respective channels are enabled from the host. To read input from the host, the channel's busy-status is checked and the read-function is called. Writing data to a channel for a variable having a read channel as well requires confirmation that there is no interference with a parallel reading action to the same variable.

Based on the size of data transferred on the out-channels, the size of the RTDX target buffer must be considered, to insure the largest message including its overhead of 8 bytes (for the C6713 processor) is accommodated. Similarly, the RTDX host buffer must be large enough for the data transferred on the in-channels to avoid buffer overruns (see chapter 2.2 for a discussion of RTDX buffers).



```
void main()
{
  comm_intr();          //init codec, dsk,MCBSP

  while(1)              //infinite loop
  {
        //read data from PC into coeffs
        if (RTDX_isInputEnabled(&in_channel1)){        //if the channel was enabled from the PC
                if (!RTDX_channelBusy(&in_channel1)){   //if channel not busy
                        RTDX_read(&in_channel1, &coeffs, sizeof(coeffs)); //read data from PC
                        RTDX_disableInput(&in_channel1);        //disable channel
                }
        }
        //write data from coeffs to PC
        if (RTDX_isOutputEnabled(&out_channel1)){
                if (!RTDX_channelBusy(&in_channel1)){  //if data is not being read into coeffs
                        RTDX_write(&out_channel1, coeffs, sizeof(coeffs)); //send data DSK-->PC
                        RTDX_disableOutput(&out_channel1);
                }
        }
```

Figure 5.9 Sending and receiving data on RTDX channels in the DSP application

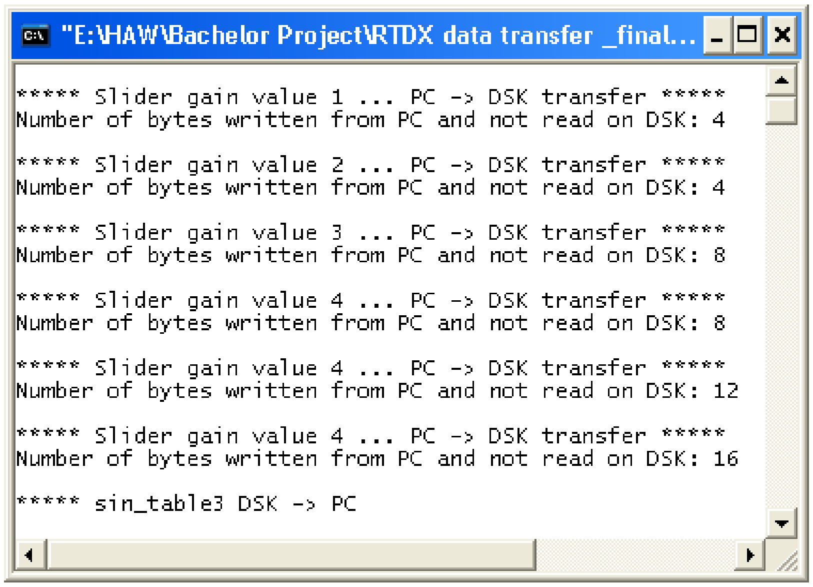
5.5 Testing Outcome for RTDX Display

In testing the RTDX Display application, a performance issue related to data transfers triggered by slider controls is noticed. Host-to-target packages sent from the OnHScroll() message handler of a slider do not reach in some cases the DSP application as expected. This behavior is investigated with the help of a query function provided by the RTDX host interface and declared as follows:

long StatusOfWrite(long * numBytes)

The numBytes parameter returns the state of the RTDX Host library internal buffer, where data sent from the host application is stored until a read request is received from the target. After the write-function in the OnHScroll() handler executes, StatusOfWrite () is called and the value numBytes is directed to the console. An attempt to increase the slider value from 1 to 4 produces the messages shown in Figure 5.8. A message is recorded for every increment passed, and additional handler executions may occur when dropping the slider in the final position. Analyzing the host-to-target transfer procedure, an inference can be made that the slider channel is at times disabled when the target read request reaches the host buffer, which leads to an occasional build-up of data packages sent from the slider and not picked up by the target in-channel.

The coming chapter presents considerations for directions of improvements and functionality extensions for the RTDX application.



"E:\HAW\Bachelor Project\RTDX data transfer _final...

```
***** Slider gain value 1 ... PC -> DSK transfer *****
Number of bytes written from PC and not read on DSK: 4

***** Slider gain value 2 ... PC -> DSK transfer *****
Number of bytes written from PC and not read on DSK: 4

***** Slider gain value 3 ... PC -> DSK transfer *****
Number of bytes written from PC and not read on DSK: 8

***** Slider gain value 4 ... PC -> DSK transfer *****
Number of bytes written from PC and not read on DSK: 8

***** Slider gain value 4 ... PC -> DSK transfer *****
Number of bytes written from PC and not read on DSK: 12

***** Slider gain value 4 ... PC -> DSK transfer *****
Number of bytes written from PC and not read on DSK: 16

***** sin_table3 DSK -> PC
```

Figure 5.10 Console displaying debugging messages from the OnHScroll() message handler function

6 Further Development of RTDX Display

In response to the drawbacks detected in the performance of RTDX Display, suitable modifications to the realization of the application are suggested here. At the same time, proposal for functionality extensions are given.

To avoid the limitations imposed by the requirement that only one outgoing RTDX channel can transfer data from the host to the target application at a given time, the design can introduce the option to group some of the host data on one single channel, when the needs of the DSP program allow it. The user can indicate to the self-adjusting host application which variables of the same type are to be grouped in arrays, and the target code must unpack them correspondingly upon receiving. This transfer alternative would allow slider channels to be kept enabled for the entire duration of the application execution, which would eliminate the occasional data losses discussed in chapter 5.5.

A helpful feature when using large edit boxes can be the possibility to have more such dialog windows available in parallel. This would permit the comparison of data from different vectors, for instance between input and output values to implemented DSP algorithms. To achieve this, all created MFC dialogs must be declared non-modal.

Additional flexibility can be easily added to the structure of the RTDX block. Initialized arrays could be read without a specified size, and preprocessor constant definitions representing the length of arrays can be included in the recognized declaration styles as well.

In the implementation of the main dialog, problems related to calculating the scrolling parameters of the window's vertical scroll bar occurred. Improvements to the vertical scrolling possibilities are necessary when dealing with dialogs containing more controls than the screen length can accommodate.

The target-to-host data transfers can be implemented for all control types supported by RTDX Display.

For DSP applications requiring frequent target-to-host transfers of data from several variables, the user can be allowed to choose if sending this data as members of one structure is beneficial. Measurements performed with the Data Rate Viewer Control can aid in estimating the impact of the target-to-host transfers on the performance of the DSP application. This transfer strategy is more suitable for single variables than for large vector variables. The RTDX block can take information regarding the method of choice from the user.

7 Conclusion

This bachelor project set out to build a tool that accompanies the development of digital signal processing applications on the TMS320C6713 digital signal processor of Texas Instruments. The tool was required to provide visual and interactive means to monitor and control the execution of DSP applications from a PC environment, by employing TI’s Real-Time Data Exchange technology. RTDX supplies a framework for communicating in a non-invasive manner with real-time DSP applications, allowing in this way the acquisition of accurate debugging results. In utilizing this technology, the principal demand of the PC-hosted application was to have the capability to adjust itself according to the specific DSP program needs.

RTDX Display was developed as an MFC-based application which collects its defining information from the DSP program it supports. By laying down a keyword-based format for the user requests, the PC application achieves the required ability to automatically generate a complete structure enabled for data exchange with the DSP.

A detailed requirement analysis was carried out and a variety of RTDX capabilities were investigated to create a design that responds to the common DSP application debugging needs presented by the intended users.

The developed application realizes PC-to-DSP data transfer for all data structures supported and it provides DSP-to-PC transfer for the most frequent scenarios specific to the described development setup. The evaluation of the application performance yielded a series of recommendations for further extensions and improvements. RTDX Display enhances the accuracy of DSP application debugging by providing user-determined tools for real-time analysis.

8 References

[1] Chassaing, R.: *Digital Signal Processing and Applications with the C6713 and C6416 DSK*
Wiley, New Jersey, 2005

[2] Spectrum Digital Inc., *TMS320C6713 DSK Technical Reference*
November 2003. Document Number: 506735-0001, Revision B

[3] Sauvagerd, U: *Dual PCM Evaluation Texas Instruments Test Board*, DV_DUETT_board_description.pdf
Hamburg University of Applied Sciences, May 2006

[4] Texas Instruments Inc., *TMS320C6713 Floating-Point Digital Signal Processor*
November 2005. Literature Number: SPRS186L

[5] Texas Instruments Inc., *DSP/BIOS, RTDX and Host-Target Communications*
February 2003. Literature Number: SPRA895

[6] Texas Instruments Inc., *TMS320C6700 Code Composer Studio Tutorial*
February 2000. Literature Number: SPRU301C

[7] Texas Instruments Inc., *TMS320 DSP/BIOS User's Guide*
November 2004. Literature Number: SPRU423F

[8] Texas Instruments Inc., *DSP/BIOS Kernel Technical Overview*
August 2001. Literature Number: SPRA780

[9] Texas Instruments Inc., *How to Optimize Your Target Application for RTDX Throughput*
January 2003. Literature Number: SPRA872A

[10] Texas Instruments Inc., *How to Use High-Speed RTDX Effectively*
May 2002. Literature Number: SPRA821

[11] Texas Instruments Inc., Code Composer Studio Help, *RTDX.hlp*
May 2005. Literature Number: SPRH197A

[12] Texas Instruments Inc., Tools & Software Overview, Emulators/Analyzers, *RTDX and High-Speed RTDX*
http://focus.ti.com/dsp/docs/dspfindtoolswbytooltype.tsp?sectionId=3&tabId=2093&toolTypeId=12&familyId=44
Last accessed on 10 August 2009

[13] Trimborn, M.: *Developing DSP Test Systems for TI DSPs*
National Instruments, November 2006
http://www.en-genius.net/site/zones/dspZONE/technical_notes/dsp_technote_110606#
Last accessed on 10 August 2009

[14] Sauvagerd, U.: *Software Construction in C++*
Hamburg University of Applied Sciences, March 2004

[15] Lippman, S. B., J. Lajoye und B. E. Moo: *C++ Primer, Fourth Edition*
Addison Wesley Professional, 2005

[16] Chien, C. C.: *Professional Software Development with Visual C++ 6.0 & MFC*
Charles River Media, Inc., Hingham, Massachusetts, 2002

[17] Microsoft Developer Network, MSDN Library, *MFC Reference*
http://msdn.microsoft.com/en-us/library/bk77x1wx(VS.71).aspx
Last accessed on 10 August 2009

Glossary and Abbreviations

API	Application Programming Interface
C6713	The TMS320C6713 digital signal processor of Texas Instruments
CCS	Code Composer Studio, a TI Integrated Development Environment
COM	Microsoft Component Object Model
Control	A visual object used in window-based applications to perform specific tasks
CPLD	Complex Programmable Logic Device
Dialog	Dialog box, a screen window built on the MFC framework
DSK	DSP Starter Kit
DSP	Digital signal processor
DSP/BIOS	A real-time kernel integrated with the Code Composer Studio
Emulator	A special purpose software or hardware imitating the behavior of another system
GUI	Graphical User Interface
Host	A PC which hosts software monitoring the operation of a connected DSP
HSRTDX	High-speed RTDX
In-channel	An RTDX channel for PC to DSP data transfer
JTAG	Joint Test Action Group
Library	A collection of subroutines or classes used to develop software
MFC	Microsoft Foundation Class Library
Out-channel	An RTDX channel for DSP to PC data transfer
Real-time	Describing a system that is subject to operational deadlines from event to system response
RTDX	The Real-time Data Exchange technology of Texas Instruments
RTDX block	A code section contained in a target C source file and specifying user requests for the automatic building of the RTDX Display host application
Target	A DSP on which applications being monitored from a connected PC are executed
TI	Texas Instruments

List of Figures

Appendix

This Bachelor Thesis contains an appendix including a program source code listing (MS Visual C++ project) and a supporting Code Composer Studio project on a CD deposited with Prof. Dr. Ulrich Sauvagerd.

Declaration

I declare within the meaning of section 25(4) of the Examination and Study Regulations of the International Degree Course Information Engineering that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sourced and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, 26 August 2009

Ioana Semedrea