Zigouris Th. Evangelos

Kalantzopoulos G. Athanasios

Vassalos E. Evangelos

LabVIEW to CCS Link

Internal Report

Electronics Laboratory
Computer and Electronics Division
Department of Physics
University of Patras

Patras 2007

LABVIEW TO CCS LINK

Table Of Contents

1. INTRODUCTION 1

1.1 Link for Code Composer Studio Development Tools 1
1.2 Test Integration Toolkit for TI DSPs 2
1.3 What is LabVIEW to CCS Link ? 3

2. DESCRIPTION OF SUBVIS IN LABVIEW TO CCS LINK 4

2.1 CCS Setup 4
2.1.1 CCS_Setup_Open.vi 4
2.1.2 CCS_Setup_Close.vi 5
2.1.3 CCS_Setup_Clear.vi 6
2.1.4 CCS_Setup_Add_Board.vi 6
2.1.5 CCS_Setup_Rename_Board.vi 7
2.1.6 CCS_Setup_Remove_Board.vi 8
2.1.7 CCS_Setup_Rename_Processor.vi 9
2.1.8 CCS_Setup_Boards_&_Processors.vi 10
2.1.9 CCS_Setup_Save.vi 10
2.2 CCS Automation 11
2.2.1 CCS_Open.vi 11
2.2.2 CCS_Close.vi 12
2.2.3 CCS_Open_Project.vi 13
2.2.4 CCS_Close_Project.vi 14
2.2.5 CCS_Connect.vi 14
2.2.6 CCS_Disconnect.vi 15
2.2.7 CCS_Build_All.vi 16
2.2.8 CCS_Build_Result.vi 17
2.2.9 CCS_Download.vi 18
2.2.10 CCS_Reset.vi 19
2.2.11 CCS_Run.vi 20
2.2.12 CCS_Restart.vi 21
2.2.13 CCS_Halt.vi 21
2.2.14 CCS_Is_DSP_Running.vi 22
2.2.15 CCS_RTDX_Enable.vi 23
2.2.16 CCS_RTDX_Disable.vi 24
2.2.17 CCS_RTDX_Logfile_Configuration.vi 25
2.3 CCS Communication 26
2.3.1 RTDX_Channel_Disable.vi 26
2.3.2 RTDX_Channel_Enable.vi 27
2.3.3 RTDX_Channel_Status.vi 27
2.3.4 RTDX_Read.vi 28
2.3.5 RTDX_Write.vi 42
2.3.6 MEM_Get_Address.vi 57
2.3.7 MEM_Read.vi 58
2.3.8 MEM_Write.vi 78

2.3.9 Leds_Read_(DSK6713).vi 98
2.3.10 Leds_Write_(DSK6713).vi 99
2.3.11 Switches_Read_(DSK6713).vi 100

3. USING THE LABVIEW TO CCS LINK 102

3.1 CCS Setup 102
3.1.1 CCS Setup for one board 102
3.1.2 CCS Setup for multiple boards 103
3.2 CCS Automation 105
3.2.1 Automate CCS to control one DSP 105
3.2.2 Automate CCS to control more that one DSPs 107
3.3 CCS Communication 109
3.3.1 Direct DSP memory access 109
3.3.2 Using the RTDX technology 111

4. APPLICATIONS 114

4.1 A Three-band Graphical Equalizer 114
4.1.1 Guidelines for graphical equalizers 114
4.1.2 Specifications 117
4.1.3 Design and control of the graphical equalizer using MATLAB 117
4.1.4 Implementation of the graphical equalizer in CCS 123
4.1.5 Implementation of a VI to control the graphical equalizer 127
4.1.6 Results – Conclusions 133
4.2 A Digital Image Processing Application 136
4.2.1 Edge Detection 136
4.2.2 Direct and Inverse Discrete Cosine Transformation 141
4.2.3 JPEG standard encoding and decoding 156
4.2.4 Histogram Equalization 165
4.2.5 Implementing the VI to control the application 176
4.2.6 Results – Conclusions 188

5. BIBLIOGRAPHY 191

1. Introduction

The continuous growth of DSPs (Digital Signal Processors) led many companies such as The Mathworks and National Instruments (NI) to incorporate specific tools for communication with DSP development boards in their main utilities. Texas Instruments (TI) holds the major part of the DSP market today and this is the reason why NI and The Mathworks have emphasized to TI's DSPs along with Code Composer Studio (CCS) development environment.

1.1 Link for Code Composer Studio Development Tools

The Mathworks, trying to follow the enormous growth of DSPs, created the "Link for Code Composer Studio Development Tools", which allows MATLAB and Simulink to connect with the Code Composer Studio (CCS) environment and TI's DSPs.

The "Link for Code Composer Studio Development Tools" is a set of functions, with which a two-way connection between MATLAB and CCS is created. Through this connection, the control of CCS and by consequence the control of DSP is attainable, as shown in figure 1. Using the above set of functions, data transfer from DSP memory or DSP registers, is also attainable. Data transfer from DSP memory to MATLAB and in reverse, is accomplished either by direct DSP memory access or by using TI's RTDX (Real-Time Data Exchange) technology.

The "Link for Code Composer Studio Developments Tools v2.1" requires that CCS v3.1 or later as well as MATLAB R2006b or later, are already installed.

Figure 1. MATLAB connection with CCS

In order to create Graphical User Interfaces (GUIs), MATLAB offers GUIDE (GUI Development Environment) that contains a set of tools for GUI development. These tools can really simplify the GUI design and programming. Using GUIDE in combination with Link for Code Composer Studio Development Tools, the user can create GUIs that will control and communicate with applications on TI's DSPs that are implemented with CCS.

1.2 Test Integration Toolkit for TI DSPs

For LabVIEW to connect with CCS, National Instrument (NI) offers a toolkit named DSP Test Integration Toolkit for TI DSP v2.0. When the above toolkit is installed, LabVIEW acquires a new set of subVIs, with which CCS can be software controlled and automated as well as data can be transferred from CCS to LabVIEW and in reverse, either by direct DSP memory read/write operations, or by using TI's RTDX technology.

The basic VIs that the toolkit provides, are separated into two main categories :

- Automation of CCS (CCS Automation VIs): VIs that belong to this category are used to control CSS and consequently the DSP.
- Communication with CCS (CCS Communication VIs): The VIs of this category are used for data transfers from DSP to LabVIEW and reversely, either by direct DSP memory access, or by using TI's RTDX technology.

The icon and name of every VI in Test Integration Toolkit for TI DSPs is presented in Table 1, according to the category that it belongs.

Automation of CCS		Communication with CCS	
Icon	**Name**	**Icon**	**Name**
	CCS Open Project.vi		CCS RTDX Read.vi
	CCS Build.vi		CCS RTDX Write.vi
	CCS Download Code.vi		CCS RTDX Enable.vi
	CCS Run.vi		CCS RTDX Enable Channel.vi
	CCS Halt.vi		CCS RTDX Disable.vi
	CCS Close Project.vi		CCS RTDX Disable Channel.vi
	CCS Window Visibility.vi		CCS Memory Read.vi
	CCS Reset.vi		CCS Memory Write.vi
			CCS Symbol to Memory Address.vi

Table 1. NI's Test Integration Toolkit for TI DSPs VIs

NI's toolkit requires that LabVIEW v7.0 or later and Code Composer Studio v2.2 or later is already installed. This toolkit does not support reading and writing floating point numbers and tables by direct DSP memory access. It also does not support reading and writing unsigned integers either by direct DSP memory access, or by using the RTDX technology. Moreover, NI's toolkit does not support the new CCS v3.1 capabilities, such as dynamic board connect and disconnect, which leads to the fact that CCS can not be efficiently controlled through LabVIEW.

1.3 What is LabVIEW to CCS Link ?

Following the design philosophy of NI's Test Integration Toolkit for TI DSPs and in order to overcome its weak points, a new toolkit is created from scratch "LabVIEW to CCS Link". This new toolkit has the advantage of fully controlling the Code Composer Studio v3.1 and communicating with TI's DSPs. Additionally it provides the opportunity to control CCStudio Setup v3.1, so that the hardware with which the CCS will communicate, could be software define through LabVIEW. This toolkit supports reading and writing numbers and tables of all kind (floating-point single or double precision, signed and unsigned 1-, 2-, or 4-bytes integers) as well as strings by direct DSP memory access. Moreover it supports number and table (floating-point single or double precision, signed and unsigned 1-, 2-, or 4-bytes integers) transfers from/to DSP, using the RTDX technology. The LabVIEW to CCS Link requires that LabVIEW v7.1 or later and Code Composer Studio v3.1 are already installed. In figure 2 the connection between LabVIEW and CCS is presented, using the LabVIEW to CCS Link.

Figure 2. LabVIEW connection with CCS

Using LabVIEW to CCS Link one can create in a fast and easy way VIs that will actually work as Graphical User Interfaces (GUIs) for controlling and managing DSPs' applications. The LabVIEW to CCS Link is a very useful tool not only for educational purposes but for DSP system designers as well, because of the little time, needed to develop GUIs. This is ought to LabVIEW graphical programming language and to the advantages of LabVIEW to CCS Link. Combining the capabilities of this toolkit and LabVIEW, the user can create GUIs for DSPs' applications, with huge potential for both controlling the application and further result processing or data preprocessing.

2. Description of subVIs in LabVIEW to CCS Link

The LabVIEW to CCS Link subVIs are separated into three categories, according to their usage :

- CCS Setup
- CCS Automation
- CCS Communication

2.1 CCS Setup

The subVIs that belong to CCS Setup category control the CCStudio Setup v3.1 or later so that the board(s) that will communicate with CCS could be software defined. The term "board(s)" contains both development platforms and simulators. CCS does not support the usage of multiple simulators at the same time. However it does support, the presence of one simulator and one or more boards, simultaneously. The CCS setup category SubVIs are presented in Table 2.

Icon	Name	Icon	Name
	CCS_Setup_Open.vi		CCS_Setup_Remove_ Board.vi
	CCS_Setup_Close.vi		CCS_Setup_Rename_ Processor.vi
	CCS_Setup_Clear.vi		CCS_Setup_Boards_&_ Processors.vi
	CCS_Setup_ Add_Board.vi		CCS_Setup_Save.vi
	CCS_Setup_Rename_ Board.vi		

Table 2. The VIs of the CCS Setup category

2.1.1 CCS_Setup_Open.vi

The subVI CCS_Setup_Open.vi, shown in Figure 3, loads the CCStudio Setup and creates a reference to CCStudio Setup.

Figure 3. The CCS_Setup_Open.vi

Input "Visible" is Boolean and controls if the CCStudio Setup window will become visible to the user. Its default value is True.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. If any error occurs before the execution of this specific VI, no action will be made by this VI and the content of input “error in” will pass to output “error out”. The VI will work normally only if no error has occurred in earlier stages. The default value of this input is the “no error” situation. The elements that constitute input “error in” are:

- Input “status” is Boolean. Its value is False (✓) if no error has occurred before this VI execution, or True (✗) if any error has occurred earlier. Its default value is False.
- Input “code” is a 32-bit integer showing the error code. If input “status” has a True value then the input “code” will have a non-zero value representing the specific error, otherwise it will have a zero value. Its default value is zero.
- Input “source” is a String describing an error. Its value is an empty string, if no error has occurred. Its default value is an empty String.

Output “CCSetup Out” is a reference to CCStudio Setup.

Output “error out” is a cluster of outputs containing errors information. If the “error in” input, shows that an error has occurred before the execution if the VI, then the “error out” output’s content will be the same as the “error in” input’s, otherwise it will describe the error that might has occurred during the execution of this specific VI. The elements that constitute output “error out” are:

- Output “status” is Boolean. Its value is False (✓) if no error has occurred, otherwise its True (✗).
- Output “code” is a 32-bit integer showing the error code. If the “status” output has a True value, then the “code” output will have a non-zero value representing the specific error, otherwise it will have a zero value.
- Output “source” is a String describing an error. Its value is an empty string, if no error has occurred.

2.1.2 CCS_Setup_Close.vi

The subVI CCS_Setup_Close.vi shown in Figure 4, closes the CCStudio Setup and the reference that the CCS_Setup_Open.vi has created.

Figure 4. The CCS_Setup_Close.vi

Input “CCSetup In” is a reference to CCStudio Setup.

Input“error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.1.3 CCS_Setup_Clear.vi

The subVI CCS_Setup_Clear.vi shown in Figure 5, clears all the setting that have been made at CCStudio Setup.

Figure 5. The CCS_Setup_Clear.vi

Input "CCSetup In" is a reference to CCStudio Setup.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCSetup Out" is a reference to CCStudio Setup.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.1.4 CCS_Setup_Add_Board.vi

The subVI CCS_Setup_Add_Board.vi, shown in Figure 6, adds a board to CCStudio Setup. The drivers (files with .ccs extension) of this board are declared to input "Driver Path".

Figure 6. The CCS_Setup_Add_Board.vi

Input "CCSetup In" is a reference to CCStudio Setup.

Input "Driver Path" is the full path for the drivers of the board that is to be added. The drivers files for all the boards that are supported by Code Composer Studio v3.1, as

long as CCS is installed to its default location, are in C:\CCStudio_v3.1\drivers\import*.ccs (file name).

Input "Options" is a 32-bit integer with specified states that are presented in Table 3.

Value	State	Description
0	CLEAR_ORIGINAL	Detects if the board that is about to be added, already exists even with another name. If it is, then it is replaced with its default name.
1	NO_DUPLICATES	Detects if the board that is about to be added, already exists according to its name. If it is, then it is not being added.
2	REPLACE_DUPLICATES	Detects if the board that is about to be added, already exists according to its name. If it is, then then it is replaced with its default name.
3	RENAME_DUPLICATES	Detects if the board that is about to be added, already exists according to its name. If it is, then it is replaced with another name.

Table 3. The states of "Options" input.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCSetup Out" is a reference to CCStudio Setup.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.1.5 CCS_Setup_Rename_Board.vi

The subVI CCS_Setup_Rename_Board.vi, shown in Figure 7, gives to the board indicated by the input "Board" the name that is included in input "New Name". In order to the CCS_Setup_Rename.vi to be used, at least one board has to be set to the CCStudio Setup.

Figure 7. The CCS_Setup_Rename_Board.vi

Input "CCSetup In" is a reference to CCStudio Setup.

Input "Board" is a 32-bit integer showing which board, from those that are set to CCStudio Setup, will be renamed. If N boards are set then the input "Board" can take

values from zero to N-1, otherwise an error will occur. The default value of input "Board" is zero.

Input "New Name" is a String and defines the board's new name.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCSetup Out" is a reference to CCStudio Setup.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.1.6 CCS_Setup_Remove_Board.vi

The subVI CCS_Setup_Remove_Board.vi, shown in Figure 8, removes the boardindicated by the input "Board" from the CCStudio Setup current setting. In order for the CCS_Setup_Remove_Board.vi to be used at least one board has to be set to the CCStudio Setup.

Figure 8. The CCS_Setup_Remove_Board.vi

Input "CCSetup In" is a reference to CCStudio Setup.

Input "Board" is a 32-bit integer showing which board, from those that are set to CCStudio Setup, will be removed. If N boards are set then the input "Board" can take values from zero to N-1, otherwise an error will occur. The default value of input "Board" is zero.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCSetup Out" is a reference to CCStudio Setup.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.1.7 CCS_Setup_Rename_Processor.vi

The subVI CCS_Setup_Rename_Processor.vi shown in Figure 9, renames the processor declared by the inputs “Processor” and “Board” according to the name indicated by the input “ProcName”. The input “Board” defines the board that contains the specific processor, while the input “Processor” defines the processor. In order for the CCS_Setup_Rename_ Processor.vi to be used, at least one board has to be set to the CCStudio Setup.

Figure 9. The CCS_Setup_Rename_Processor.vi

Input“ProcName” is a String containing the name that is to be given to the processor.

Input “CCSetup In” is a reference to CCStudio Setup.

Input “Board” is a 32-bit integer showing the board’s processor that is to be renamed. If N boards are set then the input “Board” can take values from zero to N-1, otherwise an error will occur.The default value of input “Board” is zero.

Input “Processor” is a 32-bit integer showing the processor that is to be renamed. If the board, that includes this processor (declared by input “Board”), has M processors then the input “Processor” can take values from 0 to M-1, otherwise an error will occur. The default value of input “Processor” is zero.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCSetup Out” is a reference to CCStudio Setup.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.1.8 CCS_Setup_Boards_&_Processors.vi

The subVI CCS_Setup_Boards_&_Processors.vi shown in Figure 10, returns an array to the output “Boards & Processors” showing the current settings of CCStudio Setup.

Figure 10. The CCS_Setup_Rename_Processor.vi

Input “CCSetup In” is a reference to CCStudio Setup.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCSetup Out” is a reference to CCStudio Setup.

Output “Boards & Processors” is a two-dimensional array with String elements showing the current settings of CCStudio Setup. The first element of each row has the name of the board, while the next elements of each row have the name of the processor that the specific board contains. For a board named my_board that contains three processors named cpu1, cpu2 and cpu3, the first element of the array’s row will be “my_board”, the second element will be “cpu1”, the third element will be “cpu2” and the fourth one will be “cpu3”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.1.9 CCS_Setup_Save.vi

The subVI CCS_Setup_Save.vi, shown in Figure 11, saves the settings to CCStudio Setup.

Figure 11. The CCS_Setup_Save.vi

Input “CCSetup In” is a reference to CCStudio Setup.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCSetup Out” is a reference to CCStudio Setup.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.2 CCS Automation

The subVIs that belong to the CCS Automation category can fully control the Code Composer Studio v3.1 or later and by consequence the DSP. In Table 4 the subVIs of this category are presented.

Icon	Name	Icon	Name
	CCS_Open.vi		CCS_Reset.vi
	CCS_Close.vi		CCS_Run.vi
	CCS_Open_ Project.vi		CCS_Restart.vi
	CCS_Close_ Project.vi		CCS_Halt.vi
	CCS_Connect.vi		CCS_Is_DSP_ Running.vi
	CCS_Disconnect.vi		CCS_RTDX_Enable.vi
	CCS_Build_All.vi		CCS_RTDX_Disable.vi
	CCS_Build_ Result.vi		CCS_RTDX_Logfile_ Configuration.vi
	CCS_Download.vi		

Table 4. The VIs of the CCS Automation category

2.2.1 CCS_Open.vi

The subVI CCS_Open.vi shown in Figure12 opens the CCS.

Figure 12. The CCS_Open.vi

Input "CCS Visible" is Boolean and controls if the Parallel Debug Manager (PDM) is going to be visible, when the CCS has been set to support more than one board. However, when CCS is set to support onle one board then the input "CCS Visible" defines if the Debug Window of CCS will be visible. Its default value is True.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" its a group containing information about CCS.

Output "Is CCS Visible" is Boolean and shows if the Parallel Debug Manager is visible (as long as the multiple board support setting has been chosen) or if the Debug Window of CCS is visible (as long as the single board support setting has been chosen).

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.2.2 CCS_Close.vi

The subVI CCS_Close.vi, shown in Figure 13, closes the CCS and the relative references.

Figure 13. The CCS_Close.vi

Input "CCS In" is a cluster containing information about CCS.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.2.3 CCS_Open_Project.vi

The subVI CCS_Open_Project.vi, shown in Figure 14, opens the project indicated by the input “Project Path” in the Debug Window of CCS for the board’s processor indicated by the input “Boards & Processor”.

Figure 14. The CCS_Open_Project.vi

Input “Project Path In” is a String indicating the full path of the project (file with .pjt extension) which is about to be loaded.

Input “CCS In” is a cluster containing information about CCS.

Input “Board & Processors” is a cluster of inputs showing the board and the processor which will be used for the project’s implementation. Input “Board & Processor” is constituted by the below elements:

- Input “Processor” is a 8-bit, unsigned integer showing the processor to be used. Its default value is zero.
- Input “Board” is a 8-bit, unsigned integer showing the board to be used. Its default value is zero.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Debug Window Visible” is Boolean and controls if the Debug Window of CCS will be visible for the specific board and processor. Its default value is True.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

2.2.4 CCS_Close_Project.vi

The subVI CCS_Close_Project.vi, shown in Figure 15, close the project and clears all information related to that.

Figure 15. The CCS_Close_Project.vi

- Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

- Input “CCS In” is a cluster containing information about CCS.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

2.2.5 CCS_Connect.vi

The subVI CCS_Connect.vi, shown in Figure 16, gives CCS the command to connect to the board. In fact it takes advantage of the capability of CCS v3.1 for dynamic connection to the board. The CCS_Connect.vi has no meaning to be used when CCS has already been set to use a simulator, because in this case the dynami connection is not supported by the CCS.

Figure 16. The CCS_Connect.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the connection with the card has been successfully completed. If this maximum time period has passed and no connection has been made, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Connection Status” is a 32-bit integer with specific states that are presented in Table 5.

Value	State	Description
0	CONNECTED	Board has been connected to CCS
1	CONNECTING	Board connect process is running
2	DISCONNECTED	Board has been disconnected to CCS
3	DISCONNECTING	Board disconnect process is running

Table 5. States of input “Connection Status”

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.2.6 CCS_Disconnect.vi

The subVI CCS_Disconnect.vi shown in Figure17 gives CCS the command to disconnect to the board. In fact it takes advantage of the capability of CCS v3.1 for dynamic disconnection to the board. The CCS_Disconnect.vi has no meaning to be used when CCS has already been set to use a simulator, because in this case the dynami connection is not supported by the CCS.

Figure 17. The CCS_Disconnect.vi

Input “CCS In” is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the disconnection with the card has been successfully completed. If this maximum time period has passed and no disconnection has been made, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "Connection Status" is a 32-bit integer with specific states that are presented in Table 6.

Value	State	Description
0	CONNECTED	Board has been connected to CCS
1	CONNECTING	Board connect process is running
2	DISCONNECTED	Board has been disconnected to CCS
3	DISCONNECTING	Board disconnect process is running

Table 5. States of input "Connection Status"

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.2.7 CCS_Build_All.vi

The subVI CCS_Build_All.vi, shown in Figure 18, gives the command to build the project and to create the executable file (with .out extension) for the project and the board that is described by the information from input "CCS In".

Figure 18. The CCS_Close_Project.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the project build has

been successfully completed. If this maximum time period has passed and the project has not been successfully built, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.2.8 CCS_Build_Result.vi

The subVI CCS_Build_Result.vi, shown in Figure 19, controls the file cc_build_Debug.log for errors, warnings or remarks that may have occurred after the project's build. The file cc_build_Debug.log is created automatically by CCS when the build process is completed. If CCS_Build_Result.vi locates an error, warning or remark and the input "Ignore Errors From Build" is False, it passes to output "error out" an error message informing the subVIs that follow.

Figure 19. The CCS_Build_Result.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Ignore Errors From Build" is Boolean. Its default value is False. If input "Ignore Errors From Build" is False and an error, warning or remark has been noticed during the build process, then an error will be passed to "error out" output informing the subVIs that follow.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.

Input “source” is a String describing an error.

Output “Error” is a String showing the number of errors that occurred during build process, regardless of the value of input “Ignore Errors From Build”

Output “Warnings” is a String showing the number of warnings that occurred during build process, regardless of the value of input “Ignore Errors From Build”

Output “Remarks” is a String showing the number of remarks that occurred during build process, regardless of the value of input “Ignore Errors From Build”

Output “CCS Out” is a cluster containing information about CCS.

Output “Build Result” is a String showing showing the results from the building process of a project, regardless of the value of input “Ignore Errors From Build”

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.2.9 CCS_Download.vi

The subVI CCS_Download.vi, shown in Figure 20, gives the command to CCS to download the executable file (with .out extension) to DSP. During the building process of the projects, the name of the executable must have been declared exactly as the name of the project and its path must be set to the Debug directory (default CCS settings) otherwise the CCS_Download.vi could not locate the specified file and an error will occur.

Figure 20. The CCS_Download.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the executable file has been successfully downloaded to the DSP. If this maximum time period has passed and the executable file has not been downloaded to the DSP, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

- Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

2.2.10 CCS_Reset.vi

The subVI CCS_Reset.vi, shown in Figure 21 commands the DSP, through CCS, to reset to its initial state.

Figure 21. The CCS_Reset.vi

- Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

- Input “CCS In” is a cluster containing information about CCS.

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the reset of the DSP has been completed successfully. If this maximum time period has passed and the reset of the DSP has not been made, an error occurs. Because f the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

- Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

2.2.11 CCS_Run.vi

The subVI CCS_Run.vi, shown in Figure 22, commands the DSP, through CCS, to initiate the execution of the progrtam.

Figure 22. The CCS_Run.vi

- Input “CCS In” is a cluster containing information about CCS.

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment that execution of the program will be initiated by the DSP. If this maximum time period has passed and the execution of the main program has not been started, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

2.2.12 CCS_Restart.vi

The subVI CCS_Restart.vi , shown in Figure 23, restarts the DSP's Program Counter from its initial value.

Figure 23. The CCS_Restart.vi

- Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.
- Input "CCS In" is a cluster containing information about CCS.
- Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the restart of the Program Counter to its initial value has benn completed successfully. If this maximum time period has passed and the restart of th PC has not been made, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).
- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.
- Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.
- Output "CCS Out" is a cluster containing information about CCS.
- Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:
  - Output "status" is Boolean and shows if any error occurred.
  - Output "code" is a 32-bit integer showing the error code.
  - Output "source" is a String describing an error.

2.2.13 CCS_Halt.vi

The subVI CCS_Halt.vi, shown in Figure 24, commands the DSP, through CCS, to stop the execution of the main program. In case the CCS_Halt.vi is executed when the DSP is halted, the command is simply ignored and no error occurs.

Figure 24. The CCS_Halt.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the execution of the main program is halted. If this maximum time period has passed and the execution of the program has not been halted by the DSP, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.2.14 CCS_Is_DSP_Running.vi

The subVI CCS_Is_DSP_Running.vi, shown in Figure 25, controls if the DSP is executing any program.

Figure 25. The CCS_Is_DSP_Running.vi

Input "CCS In" is a cluster containing information about CCS.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “DSP_State” is a 32-bit integer showing the DSP status. The value -1 means that the DSP is in an undefined state or the connection between the board and the CCS has been lost. The value 0 means that the DSP has stopped the execution of a program and the value 1 means that the DSP is running a program.

Output “Is CCS Visible” is Boolean showing if the DSP is running a program (True).

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.2.15 CCS_RTDX_Enable.vi

The subVI CCS_RTDX_Enable , shown in Figure 26, enables and controls the RTDX technology.

Figure 26. The CCS_RTDX_Enable.vi

Input “CCS In” is a cluster containing information about CCS.

Input “RTDX Settings” is a cluster of inputs containing information for setting the RTDX technology. Input “RTDX Settings” is constituted by the below elements:

- Input “Mode” is a 32-bit integer with specific states, indicating the method that the RTDX technology will follow. The value 0 means that the Non Continuous method will be used, while the value1 means that the Continuous method will be used. Its default value is zero (Non Continuous).
- Input “Buffer Size” is a 32-bit integer, indicating the size in bytes of each buffer that the RTDX technology uses. Its default value is 1024.
- Input “Num of Buffers” is a 32-bit integer, indicating the number of buffers that the RTDX technology uses. Its default value is 4.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the RTDX technology is successfully enabled. If this maximum time period has passed and the RTDX technology has not been enabled, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.2.16 CCS_RTDX_Disable.vi

The subVI CCS_RTDX_Disable.vi, shown in Figure 27, disables the RTDX technology.

Figure 27. The CCS_RTDX_Enable.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the RTDX technology is successfully disabled If this maximum time period has passed and the RTDX technology has not been disabled, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.2.17 CCS_RTDX_Logfile_Configuration.vi

The subVI CCS_RTDX_Logfile_Configuration.vi, shown in Figure 28, controls the properties of the Logfile (with .rtd extension) that the RTDX technology uses when the Non Continuous method has been chosen.

Figure 28. The CCS_RTDX_Enable.vi

- Input "CCS In" is a cluster containing information about CCS.

- Input "Logfile Parameters" is a cluster of inputs that contain information for setting the parameters of the Logfile that will be used from the RTDX technology, if the Non Continuous method will be chosen. Input "Logfile Parameters" is constituted by the below elements:
  - Input "FileSize" is a 32-bit integer, defining the size of the Logfile in bytes. Its default value is 32768 bytes.
  - Input "FileFullMode" is a 32-bit integer with specific states, indicating the way that the Logfile will be created. The value 0 corresponds to the Discard method which is the default. The value 1 corresponds to the Circular method.
  - Input "FileOpenMode" is a 32-bit integer with specific states, indicating the way that the Logfile will be opend. The value 0 corresponds to the RTDX _Read_Only method. The value 1 corresponds to the RTDX _Append method and the value 2 corresponds to the RTDX _Over_ Write method, which is the default value .

- Input "Logfile Path" is a string, defining the path that the Logfile will be craeted. Its default value is C:\CCStudio_v3.1\cc\bin\logfile.rtd (default CCS setting).

- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.

- Output "CCS Out" is a cluster containing information about CCS.

- Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:
  - Output "status" is Boolean and shows if any error occurred.
  - Output "code" is a 32-bit integer showing the error code.
  - Output "source" is a String describing an error.

2.3 CCS Communication

The subVIs that belong to the CCS Communication are used for data exchange between LabVIEW and CCS. With these subVIs, the user can develop VIs that will send to and receive from the DSP, data, either by direct DSP memory access, or by the RTDX technology. The subVIs of this category are presented in Table 7.

Icon	Name	Icon	Name
	RTDX_Channel_ Disable.vi		MEM_Read.vi
	RTDX_Channel_ Enable.vi		MEM_Write.vi
	RTDX_Channel_ Status.vi		Leds_Read_ (DSK6713).vi
	RTDX_Read.vi		Leds_Write_ (DSK6713).vi
	RTDX_Write.vi		Switches_Read_ (DSK6713).vi
	MEM_Get_ Address.vi		

Table 7. The subVIs of the CCS Communication category

2.3.1 RTDX_Channel_Disable.vi

The subVI RTDX_Channel_Disable.vi, shown in Figure 29, disables the RTDX channel indicated by the input "Channel".

Figure 29. The RTDX_Channel_Disable.vi

- Input "CCS In" is a cluster containing information about CCS.
- Input "Channel" is a String, defining the name of the RTDX channel that will be disabled.
- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.
- Output "CCS Out" is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

2.3.2 RTDX_Channel_Enable.vi

The subVI RTDX_Channel_Enable.vi, shown in Figure 30, enables the RTDX channel indicated by the input “Channel”.

Figure 30. The RTDX_Channel_Enable.vi

- Input “CCS In” is a cluster containing information about CCS.
- Input “Channel” is a String, defining the name of the RTDX channel that will be enabled.
- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.
- Output “CCS Out” is a cluster containing information about CCS.
- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

2.3.3 RTDX_Channel_Status.vi

The subVI RTDX_Channel_Enable.vi, shown in Figure 31, designates the status of the RTDX channel indicated by the input “Channel”.

Figure 31. The RTDX_Channel_Status.vi

- Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel the status of which, will be checked.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “ChannelStatus” is a 32-bit integer with specific states designating the status of the RTDX channel. The value 0 corresponds to the RTDX _Channel_Enabled status, indicates that the channel is enabled. The value 1 corresponds to the status RTDX_Channel_Disable, and it indicates that the channel is enabled and the value 2 corresponds to the Unknown status indicates that the status of channel can not be defined.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.3.4 RTDX_Read.vi

The subVI RTDX_Read.vi is a polymorphic VI that reads arrays and single or double precision floating-point numbers, 8-, 16-, 32-bit signed or unsigned integers from a RTDX channel. The RTDX_Read.vi consists of a group of subVIs that are presented in Table 8 and will be decribed in detail shortly.

Icon	Name	Icon	Name
F4	RTDX_Read_F4.vi	SA F4	RTDX_Read_SA_F4.vi
F8	RTDX_Read_F8.vi	SA F8	RTDX_Read_SA_F8.vi
I1	RTDX_Read_I1.vi	SA I1	RTDX_Read_SA_I1.vi
I2	RTDX_Read_I2.vi	SA I2	RTDX_Read_SA_I2.vi
I4	RTDX_Read_I4.vi	SA I4	RTDX_Read_SA_I4.vi
UI1	RTDX_Read_UI1.vi	SA UI1	RTDX_Read_SA_UI1.vi
UI2	RTDX_Read_UI2.vi	SA UI2	RTDX_Read_SA_UI2.vi

Icon	Name	Icon	Name
	RTDX_Read_UI4.vi		RTDX_Read_SA_UI4.vi

Table 8. The subVIs of the RTDX_Read.vi

RTDX_Read_F4.vi

The subVI RTDX_Read_F4.vi, shown in Figure 32, reads 32-bit (4 bytes), single precision floating-point numbers from the RTDX channel indicating by the input "Channel".

Figure 32. The RTDX_Read_F4.vi

- Input "CCS In" is a cluster containing information about CCS.
- Input "Channel" is a String, defining the name of the RTDX channel from which the data will be read.
- Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).
- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.
- Output "CCS Out" is a cluster containing information about CCS.
- Output "Data" is a 32-bit, single precision floating-point number showing the content of the specific RTDX channel.
- Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:
  - Output "status" is Boolean and shows if any error occurred.
  - Output "code" is a 32-bit integer showing the error code.
  - Output "source" is a String describing an error.

RTDX_Read_F8.vi

The subVI RTDX_Read_F8.vi, shown in Figure 33, reads 64-bit (8 bytes) double precision floating-point numbers from the RTDX channel indicating by the input "Channel".

Figure 33. The RTDX_Read_F8.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Channel" is a String, defining the name of the RTDX channel from which the data will be read.

Input "Timeout" is a 32-bit integer and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is a 64-bit, double precision, floating point number showing the content of the specific RTDX channel.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

RTDX_Read_I1.vi

The subVI RTDX_Read_I1.vi, shown in Figure 34, reads 8-bit (1 byte) , signed integers from the RTDX channel indicating by the input "Channel".

Figure 34. The RTDX_Read_I1.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel”is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an 8-bit, signed integer showing the content of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Read_I2.vi

The subVI RTDX_Read_I2.vi, shown in Figure 35, reads 16-bit (2 bytes), signed integers from the RTDX channel indicating by the input “Channel”.

Figure 35 The RTDX_Read_I2.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has

been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is a 16-bit, signed integer showing the content of the specific RTDX channel.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

RTDX_Read_I4.vi

The subVI RTDX_Read_I4.vi, shown in Figure 36 reads 32-bit, signed integers (4 bytes) from the RTDX channel indicating by the input "Channel".

Figure 36. The RTDX_Read_I4.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Channel" is a String, defining the name of the RTDX channel from which the data will be read.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is a 32-bit, signed integer showing the content of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Read_UI1.vi

The subVI RTDX_Read_UI1.vi, shown in Figure 37, reads 8-bit (1 byte), unsigned integer from the RTDX channel indicating by the input “Channel”.

Figure 37. The RTDX_Read_UI1.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an 8-bit, unsigned integer showing the content of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Read_UI2.vi

The subVI RTDX_Read_UI2.vi, shown in Figure 38, reads 16-bit, unsigned integers (2 bytes) from the RTDX channel indicating by the input “Channel”.

Figure 38. The RTDX_Read_UI2.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is a 16-bit, unsigned integer showing the content of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.

Output “source” is a String describing an error.

RTDX_Read_UI4.vi

The subVI RTDX_Read_UI4.vi, shown in Figure 39, reads 32-bit, unsigned integers (4 bytes) from the RTDX channel indicating by the input “Channel”.

Figure 39. The RTDX_Read_UI4.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is a 32-bit, unsigned integer showing the content of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Read_SA_F4.vi

The subVI RTDX_Read_SA_F4.vi, shown in Figure 40, reads arrays consisting of 32-bit (4 bytes), single precision, floating-point numbers from the RTDX channel indicating by the input “Channel”.

Figure 40. The RTDX_Read_SA_F4.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Channel" is a String, defining the name of the RTDX channel from which the data will be read.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is an array consisting of 32-bit, single precision floating-point numbers and comprises the contents of the specific RTDX channel.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

RTDX_Read_SA_F8.vi

The subVI RTDX_Read_SA_F8.vi, shown in Figure 41, reads arrays consisting of 64-bit (8 bytes), double precision floating-point numbers from the RTDX channel indicating by the input "Channel".

Figure 41. The RTDX_Read_SA_F8.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 64-bit (8 bytes), double precision floating-point numbers and comprises the contents of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Read_SA_I1.vi

The subVI RTDX_Read_SA_I1.vi, shown in Figure 42, reads arrays consisting of 8-bit, signed integers (1 byte) from the RTDX channel indicating by the input “Channel”.

Figure 42. The RTDX_Read_SA_I1.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read

process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is an array consisting of 8-bit, signed integers and comprises the contents of the specific RTDX channel.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

RTDX_Read_SA_I2.vi

The subVI RTDX_Read_SA_I2.vi, shown in Figure 43, reads arrays consisting of 16-bit, signed integers (2 bytes) from the RTDX channel indicating by the input "Channel".

Figure 43. The RTDX_Read_SA_I2.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Channel" is a String, defining the name of the RTDX channel from which the data will be read.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

Input “status” is Boolean and shows if any error occurred.
Input “code” is a 32-bit integer showing the error code.
Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 16-bit, signed integers and comprises the contents of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
Output “status” is Boolean and shows if any error occurred.
Output “code” is a 32-bit integer showing the error code.
Output “source” is a String describing an error.

RTDX_Read_SA_I4.vi

The subVI RTDX_Read_SA_I4.vi, shown in Figure 44, reads arrays consisting of 32-bit, signed integers (4 bytes) from the RTDX channel indicating by the input “Channel”.

Figure 44. The RTDX_Read_SA_I4.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
Input “status” is Boolean and shows if any error occurred.
Input “code” is a 32-bit integer showing the error code.
Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 32-bit, signed integers and comprises the contents of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Read_SA_UI1.vi

The subVI RTDX_Read_SA_UI1.vi, shown in Figure 45, reads arrays consisting of 8-bit (1 byte), unsigned integers from the RTDX channel indicating by the input “Channel”.

Figure 45. The RTDX_Read_SA_UI1.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel from which the data will be read.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 8-bit, unsigned integers and comprises the contents of the specific RTDX channel.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Read_SA_UI2.vi

The subVI RTDX_Read_SA_UI2.vi, shown in Figure 46, reads arrays consisting of 16-bit, unsigned integers (2 bytes) from the RTDX channel indicating by the input "Channel".

Figure 46. The RTDX_Read_SA_UI2.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Channel" is a String, defining the name of the RTDX channel from which the data will be read.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is an array consisting of 16-bit, unsigned integers and comprises the contents of the specific RTDX channel.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

RTDX_Read_SA_UI4.vi

The subVI RTDX_Read_SA_UI4.vi, shown in Figure 47, reads arrays consisting of 32-bit, unsigned integers (4 bytes) from the RTDX channel indicating by the input "Channel".

Figure 47. The RTDX_Read_SA_UI4.vi

Input "CCS In" is a cluster containing information about CCS.

Input "Channel" is a String, defining the name of the RTDX channel from which the data will be read.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is an array consisting of 32-bit, unsigned integers and comprises the contents of the specific RTDX channel.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

2.3.5 RTDX_Write.vi

The subVI RTDX_Write.vi is a polymorphic VI that writes arraysa nd single or double precision floating-point numbers, 8-, 16-, 32-bit signed or unsigned integers to a RTDX channel. The RTDX_Write.vi consists of a group of subVIs that are presented in Table 9 and will be decribed in detail shortly.

Icon	Name	Icon	Name
RT DX F4	RTDX_Write_F4.vi	RT DX SA F4	RTDX_Write_SA_F4.vi
RT DX F8	RTDX_Write_F8.vi	RT DX SA F8	RTDX_Write_SA_F8.vi

Icon	Name	Icon	Name
RT DX I1	RTDX_Write_I1.vi	RT DX SA I1	RTDX_Write_SA_I1.vi
RT DX I2	RTDX_Write_I2.vi	RT DX SA I2	RTDX_Write_SA_I2.vi
RT DX I4	RTDX_Write_I4.vi	RT DX SA I4	RTDX_Write_SA_I4.vi
RT DX UI1	RTDX_Write_UI1.vi	RT DX SA UI1	RTDX_Write_SA_UI1.vi
RT DX UI2	RTDX_Write_UI2.vi	RT DX SA UI2	RTDX_Write_SA_UI2.vi
RT DX UI4	RTDX_Write_UI4.vi	RT DX SA UI4	RTDX_Write_SA_UI4.vi

Table 9. The subVIs of theRTDX_Write.vi

RTDX_Write_F4.vi

The subVI RTDX_Write_F4.vi, shown in Figure 48, writes arrays that contain 32-bit (4 bytes), single precision floating-point numbers to the RTDX channel indicating by the input "Channel".

Figure 48. The RTDX_Write_F4.vi

- Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

- Input "CCS In" is a cluster containing information about CCS.

- Input "Channel" is a String, defining the name of the RTDX channel in which data will be written.

- Input "Data" is a 32-bit, single precision, floating point number, containing the data that will be written to the RTDX channel.

- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Write_F8.vi

The subVI RTDX_Write_F8.vi, shown in Figure 49 writes 64-bit (8 bytes), double precision floating-point numbers to the RTDX channel indicating by the input “Channel”.

Figure 49. The RTDX_Write_F8.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is 64-bit, single precision floating-point number, containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.

Output “code” is a 32-bit integer showing the error code.
Output “source” is a String describing an error.

RTDX_Write_I1.vi

The subVI RTDX_Write_I1.vi, shown in Figure 50, writes 8-bit, signed integers (1 byte) to the RTDX channel indicating by the input “Channel”.

Figure 50. The RTDX_Write_I1.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is an 8-bit, signed integer, containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

Input “status” is Boolean and shows if any error occurred.
Input “code” is a 32-bit integer showing the error code.
Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

Output “status” is Boolean and shows if any error occurred.
Output “code” is a 32-bit integer showing the error code.
Output “source” is a String describing an error.

RTDX_Write_I2.vi

The subVI RTDX_Write_I2.vi, shown in Figure 51, writes 16-bit, signed integers (2 bytes) to the RTDX channel indicating by the input “Channel”.

Figure 51. The RTDX_Write_I2.vi

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “CCS In” is a cluster containing information about CCS.

- Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

- Input “Data” is a 16-bit, signed integer, containing the data that will be written to the RTDX channel.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

RTDX_Write_I4.vi

The subVI RTDX_Write_I4.vi, shown in Figure 52, writes 32-bit, signed integers (4 bytes) to the RTDX channel indicating by the input “Channel”.

Figure 52. The RTDX_Write_I4.vi

- Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

- Input "CCS In" is a cluster containing information about CCS.

- Input "Channel" is a String, defining the name of the RTDX channel in which data will be written.

- Input "Data" is a 32-bit, signed integer, containing the data that will be written to the RTDX channel.

- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.

- Output "CCS Out" is a cluster containing information about CCS.

- Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:
  - Output "status" is Boolean and shows if any error occurred.
  - Output "code" is a 32-bit integer showing the error code.
  - Output "source" is a String describing an error.

RTDX_Write_UI1.vi

The subVI RTDX_Write_UI1.vi, shown in Figure 53, writes 8-bit (1 byte), unsigned integers to the RTDX channel indicating by the input "Channel".

Figure 53. The RTDX_Write_UI1.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is an 8-bit, unsigned integer, containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Write_UI2.vi

The subVI RTDX_Write_UI2.vi, shown in Figure 54, writes 16-bit, unsigned integers (2 bytes) to the RTDX channel indicating by the input “Channel”.

Figure 54. The RTDX_Write_UI2.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the

values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is a 16-bit, unsigned integer, containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Write_UI4.vi

The subVI RTDX_Write_UI4.vi, shown in Figure 55, writes 32-bit, unsigned integers (4 bytes) to the RTDX channel indicating by the input “Channel”.

Figure 55. The RTDX_Write_UI4.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is a 32-bit, unsigned integer, containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Write_SA_F4.vi

The subVI RTDX_Write_SA_F4.vi, shown in Figure 56, writes arrays that consist of 32-bit (4 bytes), single precision floating-point numbers to the RTDX channel indicating by the input “Channel”.

Figure 56. The RTDX_Write_SA_F4.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is an array that consists of 32-bit, single precision floating-point numbers containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Write_SA_F8.vi

The subVI RTDX_Write_SA_F8.vi, shown in Figure 57, writes arrays that consist of 64-bit (8 bytes), double precision floating-point numbers to the RTDX channel indicating by the input “Channel”.

Figure 57. The RTDX_Write_SA_F8.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is an array that consists of 64-bit (8 bytes), double precision floating-point numbers containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.

Input “code” is a 32-bit integer showing the error code.
Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
Output “status” is Boolean and shows if any error occurred.
Output “code” is a 32-bit integer showing the error code.
Output “source” is a String describing an error.

RTDX_Write_SA_I1.vi

The subVI RTDX_Write_SA_I1.vi, shown in Figure 58, writes arrays that consist of 8-bit, signed integers (1 byte) to the RTDX channel indicating by the input “Channel”.

Figure 58. The RTDX_Write_SA_I1.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is an array that consists of 8-bit, signed integers containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
Input “status” is Boolean and shows if any error occurred.
Input “code” is a 32-bit integer showing the error code.
Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

RTDX_Write_SA_I2.vi

The subVI RTDX_Write_SA_I2.vi, shown in Figure 59, writes arrays that consist of 16-bit, signed integers (2 bytes) to the RTDX channel indicating by the input “Channel”.

Figure 59. The RTDX_Write_SA_I2.vi

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “CCS In” is a cluster containing information about CCS.

- Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

- Input “Data” is an array that consists of 16-bit, signed integers containing the data that will be written to the RTDX channel.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

RTDX_Write_SA_I4.vi

The subVI RTDX_Write_SA_I4.vi, shown in Figure 60, writes arrays that consist of 32-bit, signed integers (4 bytes) to the RTDX channel indicating by the input "Channel".

Figure 60. The RTDX_Write_SA_I4.vi

- Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

- Input "CCS In" is a cluster containing information about CCS.

- Input "Channel" is a String, defining the name of the RTDX channel in which data will be written.

- Input "Data" is an array that consists of 32-bit, signed integers containing the data that will be written to the RTDX channel.

- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.

- Output "CCS Out" is a cluster containing information about CCS.

- Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:
  - Output "status" is Boolean and shows if any error occurred.
  - Output "code" is a 32-bit integer showing the error code.
  - Output "source" is a String describing an error.

RTDX_Write_SA_UI1.vi

The subVI RTDX_Write_SA_UI1.vi, shown in Figure 61, writes arrays that consist of 8-bit (1 byte), unsigned integers to the RTDX channel indicating by the input "Channel".

Figure 61. The RTDX_Write_SA_UI1.vi

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "CCS In" is a cluster containing information about CCS.

Input "Channel" is a String, defining the name of the RTDX channel in which data will be written.

Input "Data" is an array that consists of 8-bit, unsigned integers containing the data that will be written to the RTDX channel.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

RTDX_Write_SA_UI2.vi

The subVI RTDX_Write_SA_UI2.vi, shown in Figure 62, writes arrays that consist of 16-bit, unsigned integers (2 bytes) to the RTDX channel indicating by the input "Channel".

Figure 62. The RTDX_Write_SA_UI2.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is an array that consists of 16-bit, unsigned integers containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

RTDX_Write_SA_UI4.vi

The subVI RTDX_Write_SA_UI4.vi, shown in Figure 63, writes arrays that consist of 32-bit, unsigned integers (4 bytes) to the RTDX channel indicating by the input “Channel”.

Figure 63. The RTDX_Write_SA_UI4.vi

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the

values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “CCS In” is a cluster containing information about CCS.

Input “Channel” is a String, defining the name of the RTDX channel in which data will be written.

Input “Data” is an array that consists of 32-bit, unsigned integers containing the data that will be written to the RTDX channel.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.3.6 MEM_Get_Address.vi

The subVI MEM_Get_Address.vi, shown in Figure 64, locates the page and the address on DSP memory where the symbol indicated by the input “Symbol Name” is saved. The term symbol means an array or variable of any type that has already been set in the CCS project. If it is for an array the MEM_Get_Address.vi will return the page and address on DSP memorywhere the first element of the array is saved. The symbol is suggested to be global.

Figure 64. The MEM_Get_Address.vi

Input “CCS In” is a cluster containing information about CCS.

Input “Symbol Name” is a String defining the name of the symbol. The symbol must have already been set to the project otherwise an error will occur.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

Input “status” is Boolean and shows if any error occurred.
Input “code” is a 32-bit integer showing the error code.
Input “source” is a String describing an error.

Output “CCS Out” is a cluster containing information about CCS.

Output “Page & Address” is a cluster providing information for the page and address in DSP memory where the symbol indicated by the input “Symbol Name” is saved.
Output “Page & Address” is constituted by the below elements:

Output “Page” is a 16-bit integer, showing the page of the DSP memory in which the symbol is saved.
Output “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory where the symbol is saved.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

Output “status” is Boolean and shows if any error occurred.
Output “code” is a 32-bit integer showing the error code.
Output “source” is a String describing an error.

2.3.7 MEM_Read.vi

The subVI MEM_Read.vi is a polymorphic VI having the capability to read rows of Strings, arrays and single or double precision floating-point numbers, 8-, 16-, 32-bit signed or unsigned integers from the DSP memory by direct memory access. The MEM_Read.vi consists of subVIs that is presented in Table 10 and will be described shortly in detail.

Icon	Name	Icon	Name
F4	MEM_Read_F4.vi	A F4	MEM_Read_A_F4.vi
F8	MEM_Read_F8.vi	A F8	MEM_Read_A_F8.vi
I1	MEM_Read_I1.vi	A I1	MEM_Read_A_I1.vi
I2	MEM_Read_I2.vi	A I2	MEM_Read_A_I2.vi
I4	MEM_Read_I4.vi	A I4	MEM_Read_A_I4.vi
UI1	MEM_Read_UI1.vi	A UI1	MEM_Read_A_UI1.vi
UI2	MEM_Read_UI2.vi	A UI2	MEM_Read_A_UI2.vi
UI4	MEM_Read_UI4.vi	A UI4	MEM_Read_A_UI4.vi
String	MEM_Read_String.vi		

Table10. The subVIs of the MEM_Read.vi

MEM_Read_F4.vi

The subVI MEM_Read_F4.vi, shown in Figure 65, reads 32-bit (4 bytes), single precision floating-point numbers starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 65. The MEM_Read_F4.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is a 32-bit, single precision floating-point number designating the content of the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_F8.vi

The subVI MEM_Read_F8.vi, shown in Figure 66, reads double precision floating-point numbers των 64 bits (8 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 66. The MEM_Read_F8.vi

- Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

- Input “CCS In” is a cluster containing information about CCS.

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:
  - Input “Page” is a 16-bit integer, showing the page of the DSP memory.
  - Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

- Output “CCS Out” is a cluster containing information about CCS.

Output "Data" is a 64-bit, double precision, floating point number designating the content of the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

MEM_Read_I1.vi

The subVI MEM_Read_I1.vi, shown in Figure 67, reads 8-bit, signed integers (1 byte) from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 66. The MEM_Read_I1.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an 8-bit, signed integer designating the content of the DSP memory location indicated by the input “Page & Number”..

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_I2.vi

The subVI MEM_Read_I2.vi, shown in Figure 68, reads 16-bit, signed integers (2 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 68. The MEM_Read_I2.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is a 16-bit, signed integer designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_I4.vi

The subVI MEM_Read_I4.vi, shown in Figure 69, reads 32-bit, signed integers (4 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 69. The MEM_Read_I4.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is a 32-bit, signed integer designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_UI1.vi

The subVI MEM_Read_UI1.vi, shown in Figure 70, reads 8-bit (1 byte), unsigned integers from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 66. The MEM_Read_UI1.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an 8-bit, unsigned integer designating the content of the DSP memory location indicated by the input “Page & Number”..

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_UI2.vi

The subVI MEM_Read_UI2.vi, shown in Figure 71, reads 16-bit, unsigned integers (2 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 71. The MEM_Read_UI2.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many

processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is a 16-bit, unsigned integer designating the content of the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

MEM_Read_UI4.vi

The subVI MEM_Read_UI4.vi, shown in Figure 72, reads 32-bit, unsigned integers (4 bytes) starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 72. The MEM_Read_UI4.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is a 32-bit, unsigned integer designating the content of the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

MEM_Read_String.vi

The subVI MEM_Read_String.vi, shown in Figure 73, reads a string starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access. The number of strings is defined by the input "Size".

Figure 73. The MEM_Read_String.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Size” is a 32-bit integer designating the number of strings that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is a String designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of strings that the output “Data” contains, is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_F4.vi

The subVI MEM_Read_A_F4.vi, shown in Figure 74, reads arrays consisting of 32-bit (4 bytes), single precision, floating-point numbers starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 74. The MEM_Read_A_F4.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Input "Size" is a 32-bit integer designating the number of elements in the array that will be read.

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "Data" is an array consisting of 32-bit, single precision floating-point numbers designating the content of the DSP memory starting from the memory location indicated by the input "Page & Number". The number of elements in the array is defined by the input "Size".

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_F8.vi

The subVI MEM_Read_A_F8.vi, shown in Figure 75, reads arrays consisting of 64-bit, single precision floating-point numbers starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 75. The MEM_Read_A_F8.vi

- Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

- Input “CCS In” is a cluster containing information about CCS.

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:
  - Input “Page” is a 16-bit integer, showing the page of the DSP memory.
  - Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Input “Size” is a 32-bit integer designating the number of elements in the array that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of double precision floating-point numbers των 64 bits designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of elements in the array is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_I1.vi

The subVI MEM_Read_A_I1.vi, shown in Figure 76, reads arrays consisting of 8-bit, signed integers (1 byte) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 76. The MEM_Read_A_I1.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Size” is a 32-bit integer designating the number of elements in the array that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 8-bit, signed integers designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of elements in the array is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_I2.vi

The subVI MEM_Read_A_I2.vi, shown in Figure 77, reads arrays consisting of 16-bit, signed integers (2 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 77. The MEM_Read_A_I2.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the

values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Size” is a 32-bit integer designating the number of elements in the array that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 16-bit, signed integers designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of elements in the array is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_I4.vi

The subVI MEM_Read_A_I4.vi, shown in Figure 78, reads arrays consisting of 32-bit, signed integers (4 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 78. The MEM_Read_A_I4.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Size” is a 32-bit integer designating the number of elements in the array that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 32-bit, signed integers designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of elements in the array is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_UI1.vi

The subVI MEM_Read_A_UI1.vi, shown in Figure 79, reads arrays consisting of 8-bit (1 byte), unsigned integers starting from the DSP memory address and page indicated by the

input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 79. The MEM_Read_A_UI1.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Size” is a 32-bit integer designating the number of elements in the array that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 8-bit, unsigned integers designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of elements in the array is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_UI2.vi

The subVI MEM_Read_A_UI2.vi, shown in Figure 80, reads arrays consisting of 16-bit, unsigned integers (2 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 80. The MEM_Read_A_UI2.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Size” is a 32-bit integer designating the number of elements in the array that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 16-bit, unsigned integers designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of elements in the array is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Read_A_UI4.vi

The subVI MEM_Read_A_UI4.vi, shown in Figure 81 reads arrays consisting of 32-bit, unsigned integers (4 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access. The number of elements in the array is defined by the input “Size”.

Figure 81. The MEM_Read_A_UI4.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process has been completed successfully. If this maximum time period has passed and the read process has not been completed, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the read operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Size” is a 32-bit integer designating the number of elements in the array that will be read.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Data” is an array consisting of 32-bit, unsigned integers designating the content of the DSP memory starting from the memory location indicated by the input “Page & Number”. The number of elements in the array is defined by the input “Size”.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.3.8 MEM_Write.vi

The subVI MEM_Write.vi is a polymorphic VI having the capability to write rows of strings, arrays and single or double precision floating-point numbers, 8-, 16-, 32-bit signed or unsigned integers to DSP memory by direct memory access. The MEM_Write.vi consists of subVIs that is presented in Table 11 and will be described shortly in detail.

Icon	Name	Icon	Name
F4	MEM_Write_F4.vi	AF4	MEM_Write_A_F4.vi
F8	MEM_Write_F8.vi	AF8	MEM_Write_A_F8.vi
I1	MEM_Write_I1.vi	AI1	MEM_Write_A_I1.vi
I2	MEM_Write_I2.vi	AI2	MEM_Write_A_I2.vi
I4	MEM_Write_I4.vi	AI4	MEM_Write_A_I4.vi
UI1	MEM_Write_UI1.vi	A UI1	MEM_Write_A_UI1.vi
UI2	MEM_Write_UI2.vi	A UI2	MEM_Write_A_UI2.vi

Icon	Name	Icon	Name
	MEM_Write_UI4.vi		MEM_Write_A_UI4.vi
	MEM_Write_String.vi		

Table 11. The subVIs of the MEM_Write.vi

MEM_Write_F4.vi

The subVI MEM_Write_F4.vi, shown in Figure 82, writes 32-bit (4 bytes), single precision floating-point numbers starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 82. The MEM_Write_F4.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Input “Data” is a 32-bit, single precision floating-point number that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_F8.vi

The subVI MEM_Write_F8.vi, shown in Figure 83 writes 64-bit (8 bytes), double precision floating-point numbers starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 83. The MEM_Write_F8.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Data” is a 64-bit, single precision, floating-point numberthat will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_I1.vi

The subVI MEM_Write_I1.vi, shown in Figure 84, writes 8-bit, signed integers (1 byte) to the page and memory location indicated by the input “Page & Number” by direct memory access.

Figure 84. The MEM_Write_I1.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Data” is an 8-bit, signed integer that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_I2.vi

The subVI MEM_Write_I2.vi, shown in Figure 85 writes 16-bit, signed integers (2 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 85. The MEM_Write_I2.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write

process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Input "Data" is a 16-bit, signed integer that will be written to the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

MEM_Write_I4.vi

The subVI MEM_Write_I4.vi, shown in Figure 86, writes 32-bit, signed integers (4 bytes) starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 86. The MEM_Write_I4.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Data” is a 32-bit, signed integer that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_UI1.vi

The subVI MEM_Write_UI1.vi, shown in Figure 87, writes 8-bit (1 byte), unsigned integers to the page and memory location indicated by the input “Page & Number” by direct memory access.

Figure 87. The MEM_Write_UI1.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Data” is an 8-bit, unsigned integer that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_UI2.vi

The subVI MEM_Write_UI2.vi, shown in Figure 88, writes 16-bit, unsigned integers (2 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 88 The MEM_Write_UI2.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Input "Data" is a 16-bit, unsigned integer that will be written to the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

MEM_Write_UI4.vi

The subVI MEM_Write_UI4.vi, shown in Figure 89, writes 32-bit, unsigned integers (4 bytes) starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 89. The MEM_Write_UI4.vi

- Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

- Input "CCS In" is a cluster containing information about CCS.

- Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

- Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input "Page & Address" is constituted by the below elements:
  - Input "Page" is a 16-bit integer, showing the page of the DSP memory.
  - Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

- Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:
  - Input "status" is Boolean and shows if any error occurred.
  - Input "code" is a 32-bit integer showing the error code.
  - Input "source" is a String describing an error.

- Input "Data" is a 16-bit, unsigned integer that will be written to the DSP memory starting from the memory location indicated by the input "Page & Number".

- Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

- Output "CCS Out" is a cluster containing information about CCS.

- Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_String.vi

The subVI MEM_Write_String.vi, shown in Figure 90, writes a string starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 90 The MEM_Write_String.vi

- Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

- Input “CCS In” is a cluster containing information about CCS.

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:
  - Input “Page” is a 16-bit integer, showing the page of the DSP memory.
  - Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Input “Data” is a string that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

- Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_A_F4.vi

The subVI MEM_Write_A_F4.vi, shown in Figure 91, writes an array that consists of 32-bit (4 bytes), single precision floating-point numbers starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 91 The MEM_Write_A_F4.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Data” is an array from 32-bit, single precision, floating-point numbers, that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_A_F8.vi

The subVI MEM_Write_A_F8.vi, shown in Figure 92 writes an array that consists of double precision floating-point numbers τον 64 bits (8 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 92 The MEM_Write_A_F8.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Input “Data” is an array from 64-bit, double precision floating-point numbers, that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

- Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

- Output “CCS Out” is a cluster containing information about CCS.

- Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:
  - Output “status” is Boolean and shows if any error occurred.
  - Output “code” is a 32-bit integer showing the error code.
  - Output “source” is a String describing an error.

MEM_Write_A_I1.vi

The subVI MEM_Write_A_I1.vi, shown in Figure 93, writes an array that consists of 8-bit, signed integers (1 byte) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 93 The MEM_Write_A_I1.vi

- Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

- Input “CCS In” is a cluster containing information about CCS.

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Data” is an array from 8-bit, signed integers that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_A_I2.vi

The subVI MEM_Write_A_I2.vi, shown in Figure 94, writes an array that consists of 16-bit, signed integers (2 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 94 The MEM_Write_A_I2.vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write

process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Input "Data" is an array consisted of 16-bits signed integers that will be written to the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

MEM_Write_A_I4.vi

The subVI MEM_Write_A_I4.vi, shown in Figure 95, writes an array that consists of 32-bit, signed integers (8 bytes) starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 95 The MEM_Write_A_I4.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:

- Input “Page” is a 16-bit integer, showing the page of the DSP memory.
- Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Input “Data” is an array consisted of 32-bits signed integers that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_A_UI1.vi

The subVI MEM_Write_A_UI1.vi, shown in Figure 96, writes an array that consists of 8-bit (1 byte), unsigned integers starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 96 The MEM_Write_A_UI1.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Input "Data" is an array of an 8-bit, unsigned integers that will be written to the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

MEM_Write_A_UI2.vi

The subVI MEM_Write_A_UI2.vi, shown in Figure 97, writes an array that consists of 16-bit, unsigned integers (2 bytes) starting from the DSP memory address and page indicated by the input "Page & Address" by direct memory access.

Figure 97 The MEM_Write_A_UI2.vi

Input "CCS_Event_Notif_In" is a cluster that contains information relative to the events that may happen to CCS.

Input "CCS In" is a cluster containing information about CCS.

Input "Timeout" is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "Page & Address" is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input "Page & Address" is constituted by the below elements:

- Input "Page" is a 16-bit integer, showing the page of the DSP memory.
- Input "Address" is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Input "Data" is an array from 16-bit, unsigned integers that will be written to the DSP memory starting from the memory location indicated by the input "Page & Number".

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

MEM_Write_A_UI4.vi

The subVI MEM_Write_A_UI4.vi, shown in Figure 98, writes an array that consists of 32-bit, unsigned integers (8 bytes) starting from the DSP memory address and page indicated by the input “Page & Address” by direct memory access.

Figure 98 The MEM_Write_A_UI4.vi

- Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

- Input “CCS In” is a cluster containing information about CCS.

- Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process has been completed successfully. If this maximum time period has passed and the write process has bot been completed an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

- Input “Page & Address” is a cluster providing information about the page and the address in DSP memory where the write operation will take place. Input “Page & Address” is constituted by the below elements:
  - Input “Page” is a 16-bit integer, showing the page of the DSP memory.
  - Input “Address” is an unsigned integer in hexadecimal representation, showing the address of the DSP memory.

- Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:
  - Input “status” is Boolean and shows if any error occurred.
  - Input “code” is a 32-bit integer showing the error code.
  - Input “source” is a String describing an error.

- Input “Data” is an array from 32-bit, unsigned integers, that will be written to the DSP memory starting from the memory location indicated by the input “Page & Number”.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.3.9 Leds_Read_(DSK6713).vi

The subVI Leds_Read_(DSK6713).vi, shown in Figure 99, return to the output “Leds_Value”, the status of the the four leds on Spectrum Digital’s DSK C6713 in hexadecimal representaion.

The status of the four leds is represented by a 4-digit bynary number, where each bit corresponds to a led. The most significant digit corresponds to led 3 and the least significant to led 0. Each led is supposed to have the value 1 when it’s on otherwise the value is 0. Consequently the output “Leds_Value” takes values from 0 to 15. If the value of the output “Leds_Value” is 10, in hexadecimal representation, only the leds 1 and 3 are on.

Figure 99. The Leds_Read_(DSK6713).vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process of the status of the leds has been completed successfully. If this maximum time period has passed and the status of the leds has not been read, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “Leds_Value” is an 8-bit, unsigned integer indicating the leds’ status in hexadecimal representation on DSK C6713. Output “Leds_Value” takes values from 0 to 15.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.3.10 Leds_Write_(DSK6713).vi

The subVI Leds_Write_(DSK6713).vi, shown in Figure 100, writes the four leds’ status on DSKC6713 according to the value of input “Leds_Value”. Input “Leds_Value” takes values from 0 to 15.

The status of the four leds is represented by a 4-bit binary number, where each bit corresponds to a led. The most significant digit corresponds to led 3 and the least significant to led 0. Each led is supposed to have the value 1 when it’s on otherwise the value is 0. If the value of the input “Leds_Value”, is 12, in hexadecimal representation, the Leds_Write_(DSK6713) will change the leds status, so that only leds 2 and 3 will be on, in spite of their previous status.

Figure 100. The Leds_Write_(DSK6713).vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the write process of the status of the leds has been completed successfully. If this maximum time period has passed and the status of the leds has not been written, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user’s pc, the values of input “Timeout” may have to be increased. Its default value is 10 sec (10000 msec).

Input “Leds_Value” is an 8-bit, unsigned integer defines the new status of the leds on DSK C6713, in hexadecimal representation. Input “Leds_Value” takes values from 0 to 15.

Input “error in” is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input “error in” are:

- Input “status” is Boolean and shows if any error occurred.
- Input “code” is a 32-bit integer showing the error code.
- Input “source” is a String describing an error.

Output “CCS_Event_Notif_Out” is a cluster that contains information relative to the events that may happen to CCS.

Output “CCS Out” is a cluster containing information about CCS.

Output “error out” is a cluster of outputs containing errors information. The elements that constitute output “error out” are:

- Output “status” is Boolean and shows if any error occurred.
- Output “code” is a 32-bit integer showing the error code.
- Output “source” is a String describing an error.

2.3.11 Switches_Read_(DSK6713).vi

The subVI Switches_Read_(DSK6713).vi, shown in Figure 101, return to the output “Switches_Value” the status of the four switches on Spectrum Digital’s DSK C6713, in hexadecimal representation.

The status of the four switches is represented by a 4-digit binary number, where every digit corresponds to a switch. The most significant digit corresponds to switch 3, while the least significant digit corresponds to switch 0. Each switch is supposed to have the value 1 when it’s open (OFF status) otherwise the value is 0. Consequently the output “Switches_Value” takes values from 0 to 15. If the value of output “Switches_Value”, in hexadecimal representation, is 5 means that the switches 0 and 2 are open while 1 and 3 are closed.

Figure 101. The Switches_Read_(DSK6713).vi

Input “CCS_Event_Notif_In” is a cluster that contains information relative to the events that may happen to CCS.

Input “CCS In” is a cluster containing information about CCS.

Input “Timeout” is a 32-bit integer, and it defines the maximum time period in msec, from the moment the execution of the VI started till the moment the read process of the status of the switches has been completed successfully. If this maximum time

period has passed and the status of the switches has not been read, an error occurs. Because of the fact that many processes related to CCS and board communication are depended on the user's pc, the values of input "Timeout" may have to be increased. Its default value is 10 sec (10000 msec).

Input "error in" is a cluster of inputs describing the error that may have occurred before the execution of this specific VI. The elements that constitute input "error in" are:

- Input "status" is Boolean and shows if any error occurred.
- Input "code" is a 32-bit integer showing the error code.
- Input "source" is a String describing an error.

Output "CCS_Event_Notif_Out" is a cluster that contains information relative to the events that may happen to CCS.

Output "CCS Out" is a cluster containing information about CCS.

Output "Switches_Value" is an 8-bit, unsigned integer indicating the status of the swiches on DSK C6713, in hexadecimal representation. Output "Switches_Value" takes values from 0 to 15.

Output "error out" is a cluster of outputs containing errors information. The elements that constitute output "error out" are:

- Output "status" is Boolean and shows if any error occurred.
- Output "code" is a 32-bit integer showing the error code.
- Output "source" is a String describing an error.

3. Using the LabVIEW to CCS Link

With LabVIEW and LabVIEW to CCS Link, the user can develop in a fast and easy way Vis that will control and communicate with DSPs applications. It must be clear that the Vis developed with LabVIEW and LabVIEW to CCS Link do not create executable code for the DSP module. At first, the application must be designed in Code Composer Studio and next the user has to develop the VI that will manage it. In this point we emphasize on the way of developing a VI that will control the application.

In paragraphs that follow, we try to make understandable, the way of using the subVIs included in LabVIEW to CCS Link, according to the category they belong to.

3.1 CCS Setup

The subVIs of the CCS Setup category that were presented above, provide the user with the capability to define in software, through a VI, the board or the DSP that the CCS will use. In fact the subVIs of this category control the CCStudio Setup. Below, the way in which, the user can define one or more DSPs using a VI, is going to be presented.

3.1.1 CCS Setup for one board

In order for the user to develop a VI that would set the CCS, so that the DSKC6713 is to be used, has to think at first what are the exact steps that would be made using the CCStudio Setup manually. These steps that the user would have done manually, to define the DSKC6713 are:

- Open the CCStudio Setup
- Clear all previous settings
- Choose and add the desired board, in this case the DSKC6713, either by using the "Available Boards" window, or by entering the board's / simulator's driver (file with .ccs extension) choosing File → Import and providing the file's name and path.
- Save the settings that were made.
- Close the CCStudio Setup.

All steps above can be implemented with LabVIEW, using the CCS setup category's subVIs included in LabVIEW to CCS Link. Therefore, in the blank VI of LabVIEW the user can place these subVIs that correspond to the steps that would have been done manually. The choice of the appropriate VIs is done following this steps:

- First step is to open the CCStudio Setup, which is done by the CCSetup_ Open.vi.
- Second step is to clear all previous settings of CCStudio Setup, which is done by the CCSetup_Clear.vi.
- Third step is to choose and add a board, which is done by the CCSetup_Add_Board.vi.
- Fourth step is to save these setting, which is done by the CCSetup_Save.vi.
- Fifth step is to close the CCSetup Studio, which is done by the CCSetup_Close.vi.

When the above subVIs are placed in the blank VI, the user will have to connect them and add the necessary controls, indicators and constants in order for the VI to work. It must be noticed that the subVIs' inputs that stay disconnected, take their defaults values. In Figure 102, the final block diagram of the VI that defines the DSKC6713 through CCStudio Setup is presented.

Figure 102. The block diagram of the VI that defines DSKC6713.

The Boolean constant with the value False that was connected to the "Visible" input of the CCSetup_ Open.vi defines that CCStudio Setup will actually open but won't be visible to the user. The control "error in" that was connected to the corresponding input of CCSetup_Open.vi is optional since the thing that matters is if this specific VI is used as subVI. The constant "Driver Path" contains the full path of the DSKC6713 drives. If CCS is installed to a different location than the default, then the content of the constant "Driver Path", which is connected to the corresponding input of CCSetup_Add_ Board.vi, has to be changed as well. Finally the indicator "error out" will inform the user if any error has occurred during the VI's execution.

Figure 103. The CCStudio Setup window after the execution of the VI defining the DSKC6713.

If the above VI is executed and after that the CCStudio Setup is opened, it will be noticed that the CCS setup for using the DSKC6713 is done. The CCStudio Setup window will look like Figure 103.

3.1.2 CCS Setup for multiple boards

CCS has the advantage of using more than one board or combination of different boards and one simulator at the same time. However, it does not allow the use of more that one simulators.

Following the pattern described in paragraph 3.1.1 a VI named CCS_Setup_for_multiple_boards.vi was created, the block diagram of which is presented in Figure 104. This VI defines the simultaneous use of the DSKC6713 board, DSKC6711 board and the C6713 Device Cycle Accurate Simulator. It also alters the name of the DSKC6711 board to my_board and the name of the processor of the DSKC6713 board tomy_CPU.

Figure 104. The block diagram of CCS_Setup_for_multiple_boards.vi

The CCS_Setup_for_multiple_boards.vi at first opens the CCStudio Setup which is visible to the user since the default value of input "Visible" in CCSetup_Open.vi is True. Next, the CCSetup_Clear.vi clears all previous CCStudio Setup settings while the CCS_Setup_Add_Board.vi enters the board that corresponds to the driver indicated by the constant "Driver Path". The board is DSKC6713. The CCSetup_Rename_Board.vi alters the board's name which was entered (meaning the DSKC6713) into the one containing in the constant "New Name" which is "my_board". The CCS_Setup_Add_Board.vi is used again in order to enter the board, the driver's path of which is indicated by the constant "Driver Path1". The constant "Driver Path1" contains the DSKC6711 drivers' path, that uses the SPP378 address of the parallel port, so the board that is inserted now is the DSKC6711. The CCSetup_Rename_Processor.vi alters the name of the board's CPU, which was entered (meaning the DSKC6711) into the one included in the constant "ProcName" , that is "my_CPU". The CCS_Setup_Add_Board.vi which is used once again enters the simulator, the driver's path of which was defined by the constant "Driver Path2". The constant "Driver Path2" contains the drivers' path of the C6713 Device Cycle Accurate Simulator, so the simulator for the TMS320C6713 DSP will be entered. The new settings will be saved after the execution of the CCSetup_Save.vi. Finally the CCS_Setup_for_multiple_boards.vi execution is completed by closing the CCStudio Setup, which is done by the CCSetup_Close.vi.

After the successful execution of the CCS_Setup_for_multiple_boards.vi the settings are saved and the CCS is ready to use the DSKC6713 with its new name, the DSKC6711 with the new processor's name and the C6713 Device Cycle Accurate Simulator. The CCStudio Setup will look like Figure 105.

Figure 105. The CCStudio Setup window after the execution of the CCS_Setup_for_multiple_boards.vi

3.2 CCS Automation

The subVIs of the CCS Automation category that were presented in previous section, provide the user with the capability for software control, through a VI, of the Code Compose Studio and consequently the DSP. Below, the way in which, the user can develop a VI that will control one or more DSPs, is presented.

3.2.1 Automate CCS to control one DSP

In order for the user to develop a VI that would control a TI's DSP through CCS, has to think at first what are the exact steps that would be made using the CCStudio Setup manually. These steps that the user would have done manually, to control a DSP (as long as the CCS project has already been created) are:

- Open CCS.
- Open the desired CCS project.
- Build the project.
- Command CCS to connect with the DSP.
- Command the DSP to reset.
- Download the executable file (with .out extension) to DSP.
- Enable the RTDX technology if the project is using it.
- Command the DSP to start the execution of the program.
- As long as the desire procedure is completed, the user has to command the DSP to terminate the execution of the program.
- While the RTDX technology is enabled, the user must disable it.
- Command CCS to disconnect the board.
- Close the CCS project.
- Close the CCS.

All steps above can be implemented with LabVIEW, using the CCS Automation category's subVIs included in LabVIEW to CCS Link. Therefore, in the blank VI of LabVIEW the user can place these subVIs that correspond to the steps that would have been done manually. The choice of the appropriate VIs is done following this steps:

- The first step is to open CCS, which is done by the CCS_Open.vi.
- The second step is to open the desired CCS project, which is done by the CCS_Open_Project.vi.
- The third step is to build this project, which is done by the CCS_Build_All.vi. If the user wants to watch the results, during the building process, to the VI that it will be developed, he has to use the CCS_Build_ Result.vi
- The fourth step is to connect CCS with the DSP, which is done by the CCS_ Connect.vi.
- The fifth step is to reset the DSP, which is done by the CCS_Reset.vi.
- The sixth step is to download the executable code to DSP, which is done by the CCS_Download.vi.
- The seventh step is to enable the RTDX technology, which is done by the CCS_RTDX_Enable.vi.
- The eighth step is for the DSP to start the execution of the main program, which is done by the CCS_Run.vi.
- The ninth step is for the DSP to stop the execution of the main program, which is done by the CCS_Halt.vi.
- The tenth step is to disable the RTDX technology, which is done by the CCS_Disable_RTDX.vi.
- The eleventh step is to disconnect the board, which is done by the CCS_ Disconnect.vi.
- The twelfth step is to close the project, which is done by the CCS_Close_ Project.vi.
- The thirteenth step is to close CCS, which is done by the CCS_Close.vi.

When the above subVIs are placed in the blank VI, the user will have to connect them and add the necessary controls, indicators, constants and structures in order for the VI to work. It must be noticed that the subVIs' inputs that stay disconnected, take their defaults values. In Figure 106 the block diagram of a typical VI, that controls the CCS and consequently the DSP, is presented.

Figure 106. The block diagram of a typical VI, that controls the CCS and consequently the DSP

In the block diagram of Figure 106 the constant "Project Path" which is connected to the CCS_Open_Project.vi input "Project Path In" would have to contain the project's full path that is about to be opened in CCS. The indicator "Build Result" connects with to the CCS_Build_Result.vi output "Build Result" showing the results from the building process of

the project. A While structure is between CCS_Run.vi and CCS_Halt.vi in which the main code of the VI has to be written. The main code will have to implement the data process and transfer from / to the DSP. The way in which the main code could exchange data with the DSP is going to be presented in a following paragraph. As Figure 106 shows, the While structure stops if any error occurs or if the control "Stop" takes the value True. However, as long as the While structure is executed, the subVIs that are next can not be executed. This fact ensures that the DSP will continue the execution of the main program till it is completed or till the control "Stop" takes the value True.

It has to be noticed that in the block diagram the CCS_Connect.vi and CCS_Disconnect.vi must not be placed, if CCS is set to use any of the simulators, since the dynamic connection and disconnection is for boards and not for simulators.

If CCS is set to use any board then the CCS_Connect.vi must be placed before the subVIs that are related to the DSP, such as CCS_Run.vi, CCS_ Reset.vi etc, since the connection between the CCS and the board has to be completed. Additionally the CCS_ Disconnect.vi must be placed before the subVIs that cause the project and the CCS to close and after the subVIs that are related to the DSP, since the CCS has to be disconnected from the board.

In case of the CCS project does not use the RTDX technology the subVIs that are related to the RTDX technology, such as the CS_ RTDX_Enable.vi, CCS_RTDX_Disable.vi etc, are useless. If the Non Continuous methods of the RTDX technology are used, then the CCS_RTDX_Logfile_Configuration.vi could also be used, since the defaults settings of the logfile are not affected by the user. The use of the CCS_RTDX_Logfile_Configuration.vi is meaningless if the Continuous method of the RTDX technology is chosen, since in that case the logfile is not used at all. The choice of the RTDX method is carried out by the CCS_RTDX_Enable.vi.

3.2.2 Automate CCS to control more that one DSPs

CCS has the advantage to manage simultaneously more that one boards or a combination between boards and one simulator, through Parallel Debug Manager (PDM). The PDM is shown only if two boards (at least) or one board and one simulator are defined through CCStudio Setup. In Figure 107 the PDM window is presented, when DSK C6713 and C6711 are defined in CCStudio Setup.

Figure 107. The Parallel Debug Manager window

For the simultaneous management of the CPUs that the boards contain, CCS has the capability to open a Debug Window for each CPU. This feature provides the user with the opportunity to program and mage each board separately.

The LabVIEW to CCS Link together with the CCS Automation category's subVIs provide the user with the capability to control different Debug Window, therefore different boards. The subVI CCS_Open_Project.vi defines which card and which processor is to be used by the subVIs that follows, according to the value of input "Boards & Processors". The input "Boards & Processors" is a cluster of inputs consisting of two: the "Board" input and the "Processor" input that are accepting 8-bit, unsigned integers. Input "Board" declares the board that is to be controlled and the input "Processor" declares the processor of the board that is to be controlled. For the Debug Window of the DSK C6713 to open in the case that the PDM looks like Figure 107, the input "Boards" will have to take the value 1 and the input "Processor" the value 0. Therefore the output CCS_Out of the CCS_Open_Project.vi, would contain information related to the DSK C6713 Debug Window, so all the subVIs that will be connected directly or indirectly to this output will refer to the DSKC6713. In the same way, the output "CCS_Event_Notif_Out" contain information related to events that occur in the Debug Window of the DSK C6713.

The subVIs of the LabVIEW to CCS Link can be used many times in a single VI. Therefore, a VI can use two or more that two times the CCS_Open_Project.vi in order to open more that one Debug Windows. The user must be careful, before connects a subVI to an output "CCS_Out" or "CCS_Event_Notif_Out", to understand in what Debug Window and consequently in what CPU the specific output is referring to.

In Figure 108 the block diagram of a VI that controls two boards at the same time is presented. The boards that the VI will manage depend on the PDM. In case of the PDM looking like Figure 107 then, the VI will control simultaneously the DSK C6711 (Board 0) and the C6713 (Board 1). In the block diagram of Figure 108 the CCS_Open_Project.vi which takes as input the constant "Project Path", opens the Debug Window for the DSK C6711 (Board 0) and loads the project, the path of which is indicated by the constant "Project Path". The VIs that follow are depended on the same pattern that the automation and the control of the CCS is being done, when only one board is used as described in paragraph 3.2.1.

The CCS_Open_Project.vi that takes as input the constant "Project Path1"opens the Debug Window of DSK C6713 (Board 1) and loads the project, the path of which is indicated by the constant "Project Path1", therefore the VIs that follow are referring to the DSK C6713.

Figure 108. The block diagram of a VI that controls two DSPs through CCS

Following the pattern that described above Vis that control more that one board at the same time, can be easily developed.

3.3 CCS Communication

The subVIs of the CCS Communication are developed for data exchange between a VI and a DSP. These subVIs that manage the communication, support reading from and writing to the DSP memory either by direct access or by using the RTDX technology. The subVIs that are handling the communication with the CCS and consequently the DSP, are part of the VI's main code that automates the CCS as described in paragraph 3.2. This means that these subVIs are usually placed inside the While structure of Figures 106 and 108, so the continuous communication between CCS and DSP is ensured.

The data transfer, by direct DSP memory access, has the advantage that does not require the code modification in CCS project, while the main disadvantage of this method of data transfer is that the DSP has to interrupt the execution of the program that is running, till the data transfer is completed. The time of this interrupt depends on the data size.

TI, by developing the RTDX technology that has embodied to its DSPs, has managed to achieve faster data transfers without interrupting the execution of the program on the DSP, for great time periods, so the data transfer will be completed successfully. However, the disadvantage of this method is that the C code in a CCS project (that did not support Real Time Data eXchange) has to be modified. The H data transfer using the RTDX technology is very useful in real-time applications.

3.3.1 Direct DSP memory access

Data transfer by direct DSP memory access, provides the user with the capability to read and write data to any location in the DSP memory. This is very important for the control of an existing application, since it does require code modification in CCS. The only demand is that the variables of the project that contain the data to be transferred have to be global.

Reading from the DSP memory

In order for the value of a double type variable to be read, the MEM_Read.vi has to be used and the MEM_Read_F8 has to be chosen. Moreover, the page and the address in the DSP memory, from where the read process will begin, have to be declared, in input "Page & Address". If the user does not know the memory location where the variable is saved, then he has to use the MEM_Get_Address.vi and define in "Symbol Name" input the name of the variable. The MEM_Get_Address.vi will return the mage and address in the DSP memory, where the variable is saved. Output "Data" of MEM_Read.vi will contain the value of the variable that will be read.

In Figure 109(a) the While structure of Figure 106 is presented, as it should be modified in order to read a double type variable (8-byte, double precision, floating point number), that is saved in address 8000C1FD (hex) of page 0 in the DSP memory.

(a) **(b)**

Figure 109. Reading a double type variable by direct DSP memory access
(a) knowing its page and address, **(b)** knowing only its name

In Figure 109(b) the While structure of Figure 106 is presented, as it should be modified in order to read a double type variable (8-byte, double precision, floating point number), according to its page and address in the DSP memory, located by the MEM_Get_Address.vi using the variable's name.

With the procedure described above the reading of a variable of any type can be accomplished. In order for an array or a string to be recognized from the DSP memory the procedure presented above is followed with the difference that in input "Size" of MEM_Read.vi the number of the array's or string's elements has to be declared.

Writing to the DSP memory

In order to write data in an integer (int) variable, the MEM_Write.vi has to be used the MEM_Write_I4 has to be chosen. Moreover, the page and the address in the DSP memory, from where the write process will begin, have to be declared, in input "Page & Address". If the user does not know the memory location where the variable is saved, then he has to use the MEM_Get_Address.vi and define in "Symbol Name" input the name of the variable. The MEM_Get_Address.vi will return the mage and address in the DSP memory, where the variable is saved. Output "Data" of MEM_Write.vi will contain the value of the variable that will be written.

In Figure 110(a) the While structure of Figure 106 as it should be modified in order to write data that the control "Data" contains, in an integer (4-byte signed integer) variable which is saved in address 8000FFE0 (hex) of page 0 in the DSP memory.

In Figure 110(b) the While structure of Figure 106 as it should be modified in order to write data that the control "Data" contains, in an integer (4-byte signed integer) variable according to its page and address in the DSP memory, located by the MEM_Get_Address.vi using the variable's name.

(a) **(b)**

Figure 110. Writing an integer variable by direct DSP memory access
(a) knowing its page and address, **(b)** knowing only its name

With the procedure described above the writing of data in a variable of any type, in an array or in a string which is saved to DSP memory, can be accomplished.

3.3.2 Using the RTDX technology

The RTDX technology supported by TI's DSPs, achieves faster data transfer speeds towards direct memory access. Moreover, the RTDX technology allows data transfer from and to the DSP without interrupting the execution of the program on the DSP, for great time periods. This is the reason why in real-time applications the use of RTDX technology is necessary especially for data transfer of great size

The subVIs that belong to the CCS Communication category provides the user with the capability to transfer data from and to the DSP using the RTDX technology as well as control in software the RTDX channel. Below, the way in which data can be sent and received through RTDX channels, despite the RTDX method (Continuous ή Non Continuous) that is in use.

Receiving data through the RTDX channel.

The RTDX channels that are declared in CCS are defined as input and output channels. The input channels are transferring data to the DSP, while the output channels are transferring data from the DSP. The data of an output RTDX channel will be recognized by LabVIEW using the RTDX_Read.vi. The RTDX_Read.vi is a polymorphic VI which has the ability to read numbers and arrays of all kind from the output channel that is indicated by the input "Channel". The RTDX channels are not two-way channels, so the channel that is indicated by the input "Channel" has to be defined in CCS as output channel, otherwise an error will occur.

Figure 111. Reading from the RTDX channel

In order for an array with float type (4-byte, single precision, floating point numbers) elements to be read from an output channel named "Out_chan" the RTDX_Read.vi has to be used and the RTDX_Read_SA_F4 has to be chosen. Input "Channel" will have to be connected with constant or a control, that will contain the name of the output channel (meaning Out_chan) transferring the array.

In Figure 111 the While structure of Figure 106 is presents, as it should be modified to read data that the RTDX channel "Out_chan" contains.

Sending data through the RTDX channel

The input RTDX channels transfer data to DSP. Using the RTDX_Write.vi the writing of data from LabVIEW to a RTDX channel is possible. The RTDX_Write.vi is a polymorphic VI which has the ability to numbers and arrays of all kinds to the output channel indicated by the input "Channel". The RTDX channels are not two-way channels, so the channel that is indicated by the input "Channel" has to be defined in CCS as output channel, otherwise an error will occur.

In order for an array with unsigned short type (2-byte, unsigned integers) elements to be written from an input channel named "In_chan" the RTDX_Write.vi has to be used and the RTDX_Write_SA_UI2 has to be chosen. Input "Channel" will have to be connected with constant or a control, that will contain the name of the input channel (meaning In_chan) transferring the array.

In Figure 111 the While structure of Figure 106 is presents, as it should be modified to read data that the control "Data" contains, to the RTDX channel "In_chan".

Figure 112. Writing to a RTDX channel

4. Applications

In order to understand the capabilities of the LabVIEW to CCS Link toolkit, two complete projects will be presented. The first is a graphical three-band equalizer while the second is an application on digital image processing. Both of the applications are implemented using CCS and the DSKC6713. The control and the results of each application are realized by the corresponding VI, which has been developed in LabVIEW using the LabVIEW to CCS Link.

4.1 A Three-band Graphical Equalizer

In the following paragraphs, the theoretical guidelines about graphical equalizers will be explained. Furthermore, the design and the implementation of a three-band graphical equalizer using MATLAB and CCS will be presented compendiously. We emphasize to the development of the corresponding VI that will control and show the results of this particular equalizer.

4.1.1 Guidelines for graphical equalizers

The process of the equalization of a signal consists in amplifying and attenuating specific frequency bands of the signal. This process is very useful in cases where some frequency bands of the signal have been attenuated, related to others, and they must be amplified (as in the long-distance transmission of an audio signal through a wire). Moreover, equalizers are also used for the reverse process, which means that some frequency bands have to be attenuated in order for other frequencies to rise. The equalizers are used in almost every modern audio reproduction system, to give the desired sound's hue.

Graphical equalizers achieve the division of the spectrum of a signal in different frequency bands by using a group o f low-pass filters. An ideal low-pass filter must allow to frequencies, around a center frequency, to pass, while cutting-off all other frequencies. Unfortunately such an ideal filter is not possible to implement. In Figure 113, the respond of the filters used in graphical equalizers is presented. The Q factor of every filter is defined as the ratio of the center frequency ω_0 to the bandwidth $\Delta\omega=\omega_2-\omega_1$,

Figure 113. Frequency Response of a low-pass filter

and it is described by the equation:

$$Q = \frac{\omega_0}{\Delta\omega} = \frac{\omega_0}{\omega_2 - \omega_1} \tag{1}$$

The Q factor shows how heavy is the response of the filter.

Using N low-pass filters, the signal's frequency spectrum is divided into N regions. These filters are connected in parallel, as shown in Figure 114. The equalizer's output is the sum of all the filters' outputs. The amplification or attenuation of each region is achieved by multiplying the output of the corresponding filter with a gain factor *G*. In case where *G* is greater than one the specific frequency region is amplified, otherwise ($G<1$) is attenuated. The center frequency of each filter as well as the number of the filters depends on the application of the equalizer and on the sampling frequency of the signal (in case of a digital signal).

Figure 114. N-region digital equalizer

Usually, the main frequencies of two sequential filters are differed to an octave (one filter has ω_0 , while the others has 2 x ω_0). For audio signals (where equalizers are heavily used) ten filters (ten frequency bands) then main frequencies of which start form 31 Hz and reach 16 kHz, are used. In Figure 115 the responds of the ten low-pass filters of such an equalizer is presented. These filters are second-order IIR filters. As shown in Figure 115, there has been an overlapping in the filters' responses, because the filters are no ideal. So, when the output of a filter is amplified, not only the frequencies around the main one are amplified, but also all the other frequencies. The difference is that these frequencies (apart from the passband) have already been degraded from the filter, so in the final result only the passband frequencies are amplified. Furthermore, because of the non-ideal filters, when all the gain factors are 1 the final frequency response of the equalizer is not linear, as someone would expect.

Figure 115. Filter's responses of a ten-band equalizer

The low-pass filters of an equalizer usually are IIR, because for their implementation less coefficients are needed compared to the corresponding FIR filters. In addition, most of the times these are second-order filters, which means that they consist of only one biquad (Figure 116). As a result, only 5 multiplications and two sums are required for each filter. The transfer function of such a biquad is:

$$H_i(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{2}$$

The use of second-order IIR filters contributes to the reduction of time needed by the processor, to produce the output.

Figure 116. A biquad

4.1.2 Specifications

The three-band graphical equalizer that will be shortly implemented, consists of three second-order IIR filters. The specifications of these three filters are presented in Table 12.

Filter Type	Filter Order	Designing Method	Sampling Frequency	Passband
Low-Pass Butterworth	2nd	Bilinear Transformation	Fs = 48 kHz	< 5.5 kHz
High-Pass Butterworth	2nd	Bilinear Transformation	Fs = 48 kHz	5.5 – 11 kHz
Band-Pass Butterworth	2nd	Bilinear Transformation	Fs = 48 kHz	> 11 kHz

Table 12. Specifications of the three-band graphical equalizer

The graphical equalizer will accept a hypothetical signal, a composition of three sinusoidals with frequencies at 200Hz, 7kHz and 14kHz respectively, created with MATLAB. This hypothetical signal will be used to control the equalizer through MATLAB and to compare the results with these that will rise from the equalizer's CCS implementation.

The control of the equalizer, implemented with CCS for the DSKC6713, will be done through a VI, developed with LabVIEW using the LabVIEW to CCS Link. The communication between the VI, the CCS and the DSKC6713 will be achieved by using the RTDX technology.

4.1.3 Design and control of the graphical equalizer using MATLAB

The design of the equalizer using MATLAB, comprises the filters' coefficients computation according to the given specifications. τρων. The control of the equalizer will be done with a hypothetical signal created in MATLAB.

Filters' coefficients computation.

For the IIR filters' coefficients computation that the equalizer includes, has been made by the following m-file.

```
clear all;
close all;

n=1;            %Number of Biquads
fs = 48000;     %Sampling frequency

%Initialize coefficients' Table
scoefs = zeros(3*n,6);

%Axes limits
nxmin = 10;
nxmax = fs/2;
nymin = -50;            %log scale
```

```
nymax = 10;             %log scale

%Computation and representation of the low-pass filter
[b,a]=butter(2*n,2*5500/fs);
scoefs(1:n,:) = tf2sos(b,a);
[h,w]=freqz(b,a,fs/2,fs);
figure
        semilogx(20*log10(abs(h)))
        axis([nxmin nxmax nymin nymax])

% Computation and representation of the band-pass filter

[b,a]=butter(n,[2*5500/fs 2*11000/fs]);
scoefs(1:n,:) = tf2sos(b,a);
[h,w]=freqz(b,a,fs/2,fs);
hold on
        semilogx(20*log10(abs(h)))

% Computation and representation of the high-pass filter

 [b,a]=butter(2*n,2*11000/fs,'high');
scoefs((n+1):2*n,:) = tf2sos(b,a);
[h,w]=freqz(b,a,fs/2,fs);
hold on
semilogx(20*log10(abs(h))); title('frequency response (dB)')

coefs(:,1) = -scoefs(:,5);        % -a1
coefs(:,2) = -scoefs(:,6);        % -a2
coefs(:,3) =  scoefs(:,1);        %  b0
coefs(:,4) =  scoefs(:,2);        %  b1
coefs(:,5) =  scoefs(:,3);        %  b2

%Save the filters' coefficients to a .cof file
save_equ_coefs(coefs,n);
```

Program 1. The m-file that calculates the IIR filters' coefficients of the equalizer

In the above m-file the function *save_equ_coefs()* is called, which saves the filter;s coefficients to a .cof file with a name decided by the user. Next , the *save_equ_coefs()* function is presented.

```
function save_equ_coefs(coefs,SOS);

[filename,pathname,filterindex]=uiputfile('*.cof','Save the equalizer cofficients');
if (filterindex ~= 0 )
   fid=fopen([pathname,filename],'w');

   fprintf(fid,'/* Equalizer coefficients in float format */\r\n');
   fprintf(fid,'\r\n#define sections %d \r\n',SOS);

   for i=1:1:size(coefs,1)/SOS
   fprintf(fid,'\r\nfloat coefs%d[%d] = \r\n',i-1,5*SOS);
```

```
        fprintf(fid,'{\r\n');
        for j=1:1:SOS
            fprintf(fid,'\t%6d\t,\t%6d\t,\t%6d\t,\t%6d\t,\t%6d,\t\r\n',...
            coefs((i-1)*SOS+j,1),coefs((i-1)*SOS+j,2),coefs((i-1)*SOS+j,3),...
            coefs((i-1)*SOS+j,4),coefs((i-1)*SOS+j,5));
        end
        fprintf(fid,'};\r\n');
        end
    fclose(fid);
end
```

Program 2. The save_equ_coefs() function

When the above m-file (Program 1) is executed, a window will appear asking for the name of the file to be created that will contain the filters' coefficients. If the name of this file is equalizer.cof, it will look like Figure 117. The m-file will also design the filters' response of the equalizer (Figure 118).

Figure 117. The filters' coefficient file of the equalizer, named equalizer.cof

Figure 118. The filters; response of the equalizer

From Figure 118 it is concluded that the filters of the equalizer meet the desired specifications.

Generation of a hypothetical signal

In order to control the function of the equalizer, a hypothetical signal will be used consisting of three sinusoidal with frequencies 200Hz, 7kHz and 14kHz respectively. The sampling frequency of this signal is 48kHz. The m-file that is shown below generates the hypothetical signal, represents the signal, calculates and represents the 512-point FFT of the signal. Finally, it saves the signals' points to a header file (with .h extension), so as to be used for the control of the equalizer through.

```
close all;
clear all;

f1=200; f2=7000; f3=14000;  %Sinusoidals' frequencies
fs = 48000;                 %Sampling frequency

%Generation of the hypothetical signal
t = 0:1/fs:1;
x1 = sin(2*pi*t*f1);
x2 = sin(2*pi*t*f2);
x3 = sin(2*pi*t*f3);
xtot = (x1 + x2 +x3)/3;
xs = xtot(1:512);
xq = round(xs*(2^15-1));

%Representation of the  signal
figure(1)
plot(xq)
axis([0 512 -inf inf])

%Calculation and representation of the FFT
y = abs(fft(xq));
figure(2)
plot(y)
axis([0 512 -inf inf])

%Saving the  signal
save_signal(xq)
```

Program 3. The m-file that generates the hypothetical signal

The function *save_signal()* is used in order for the user decide, from the dialog window that will appear, the name of the header file of the hypothetical signal. The function *save_signal()* is shown below.

```
function save_signal(data);

[filename,pathname,filterindex]=uiputfile('*.h','Save the signal');
if (filterindex ~= 0 )
fid=fopen([pathname,filename],'w');
```

```
signal_size = length(data);

fprintf(fid,'/* Signal data*/\r\n');

fprintf(fid,'\r\nshort signal[%d] = \r\n',signal_size);
fprintf(fid,'{\r\n');
for i=1:(signal_size)

fprintf(fid,'\t%6d,\r\n', data(i));
end
fprintf(fid,'};\r\n');
fclose(fid);
end
```

Program 4. The function *save_signal()*

When the above m-file is executed (Program 3), a dialog window will appear asking for the file's name to be created and will contain the 512 points of the hypothetical signal. If the name is decided to be signal_200_7k_14k.h. the file will look like Figure 119.

Figure 119. The file signal_200_7k_14k.h that contains the 512 points of the hypothetical signal

The hypothetical signal that will be crated by the m-file is presented in Figure 120. The 512-point FFT of the hypothetical signal is presented in Figure 121. It is concluded that the generated signal actually contains three the three sinusoidals (200Hz, 7kHz and 14kHz)

Figure 120. The hypothetical signal

Figure 121. The 512-point FFT of the hypothetical signal

MATLAB results of the equalizer

The m-file shown below calculates and represents the output of each filter of the equalizer when the input is the signal of Figure 120.

```
close all;
clear all;

f1=200; f2=7000; f3=14000;   %Sinusoidals frequencies
fs=48000;                    %Sampling frequency

%Generation of hypothetical signal
t = 0:1/fs:10;
x1 = sin(2*pi*t*f1);
x2 = sin(2*pi*t*f2);
x3 = sin(2*pi*t*f3);
xtot = round(((x1 + x2 +x3)/3)*(2^15-1));
xs = xtot(1:(10*fs));
xs1= xtot(1:512);

%Low-pass filtering
[b,a]=butter(2,2*5500/fs);
y1=filter(b,a,xs1);
figure;
plot(y1);

%Band-pass filtering
[b,a]=butter(1,[2*5500/fs 2*11000/fs]);
y2=filter(b,a,xs1);
figure;
plot(y2);

%High-pass filtering
[b,a]=butter(2,2*11000/fs,'high');
y3=filter(b,a,xs1);
figure;
plot(y3);
```

Program 5. The m-file that calculates the filters' output

The filters' outputs of the equalizer that are presented in Figures 122, 123 and 124 corresponds to the equalizer's output when the amplification in each filter's frequency band is 1, while the amplification to all others bands is 0

Figure 122. The output of the low-pass filter

Figure 123. The output of the band-pass filter

Figure 124. The output of the high-pass filter

4.1.4 Implementation of the graphical equalizer in CCS

The graphical equalizer that is going to be implemented in CCS for DSKC6713 will have as an input the hypothetical signal, created in paragraph 4.1.3. The hypothetical signal is being used in order to compare the results of the equalizer with those raised from MATLAB to confirm its accurate function. Furthermore, by using this hypothetical signal, the function of the equalizer is becoming more understandable to the user - student.

The C code of the sim_equ3_rtdx.c, implementing the three-band graphical equalizer, is shown below.

```
// Includes

#include <rtdx.h>                    // Defines RTDX target API calls
#include "target.h"                  // Defines TARGET_INITIALIZE()
#include "equalizer.cof"             // Filters coefficients
#include "signal_200_7k_14k.h"       // Input signal

// Defines

#define sgain 32768                  // Compensates for the out sample shifting
#define S 512                        // Length of input signal

// Global variables

float f0_dly[2*sections] = {0.0};    // 1st filter delay line
float f1_dly[2*sections] = {0.0};    // 2nd filter delay line
float f2_dly[2*sections] = {0.0};    // 3rd filter delay line

short gains[3] = {0, 0, 0};

int fsignal[S];

// Functions


float iir_cas5(float input, float *c, float *d, int n)
{
  float k0;
  float temp;
  int i;

  temp = input;
  for (i=0; i<n; i++)
  {
      k0 = temp + c[5*i+1]*(d[2*i+1]) + c[5*i+0]*(d[2*i+0]);
      temp = c[5*i+4]*(d[2*i+1]) + c[5*i+3]*(d[2*i+0]) + (c[5*i+2]*k0);

      d[2*i+1] = d[2*i+0];
      d[2*i+0] = k0;
  }
  return temp*sgain;
}

//Defines RTDX channels

RTDX_CreateOutputChannel( signal_chan );
RTDX_CreateOutputChannel( fsignal_chan );
RTDX_CreateInputChannel( gains_chan );

// Main

void main()
{
  float input_sample;
```

```
    int j;
    int data_out[3];
    int data_sum;

    TARGET_INITIALIZE();

    RTDX_enableInput(&gains_chan);
    RTDX_enableOutput(&fsignal_chan);
    RTDX_enableOutput(&signal_chan);

    fsignal[0]=0;

    while(1)
    {

        RTDX_read( &gains_chan, gains, sizeof(gains));

        for (j=1; j<S; j++)
        {
            input_sample = (float)(signal[j]/10);

            data_out[0] = (int)(gains[0]*iir_cas5(input_sample, coefs0, f0_dly, sections));
            data_out[1] = (int)(gains[1]*iir_cas5(input_sample, coefs1, f1_dly, sections));
            data_out[2] = (int)(gains[2]*iir_cas5(input_sample, coefs2, f2_dly, sections));


            data_sum = data_out[0] + data_out[1] + data_out[2];
            fsignal[j] =(int)(((data_sum)>>15));
        }

        RTDX_write( &signal_chan, signal, sizeof(signal) );
        RTDX_write( &fsignal_chan, fsignal, sizeof(fsignal) );
    }
}
```

Program 5. The C code of the sim_equ3_rtdx.c

In Program 5, firstly the files r*tdx.h* , *target.h*, *equalizer.cof* and *signal_200_7k_14k.h* are included. The file *rtdx.h* includes the declarations of the functions related to the RTDX technology. The file *target.h* includes the declaration of function TARGET_INITIALIZE(), which initializes the DSP and activates the interrupts so for the RTDX technology to be enabled. The file *equalizer.cof* contains the filters' coefficients of the equalizer as they were calculated in MATLAB. The file *signal_200_7k_14k.h* created in MATLAB and contains complex signal that will be used as an input to the equalizer.

Before the declaration of the *main()* function , the RTDX channels that will transfer information from and to the DSP, are defined. The channel "gains_chan" transfer a three-number array to the DSP stating the amplification of each frequency. The channels "fsignal_chan" and "signal_chan" transfer to the VI, controlled by the DSP, the input and output signal of the equalizer.

In *main()* function, the DSP is initialized, the RTDX technology and the RTDX channels are enabled. Inside the while structure, after the content of the channel "gains_chan" is read, the filters' output of the equalizer is calculated, for every input signal, by calling *iir_cas5()* function. The filters; outputs after they are amplified properly, they are summed in order to

calculate the corresponding sample of the output signal of the equalizer. All 512 samples of the output signal are calculated, and all 512 samples of the input and output signal are written to the respective RTDX channels.

The project that will be created in CCS, for the implementation of the equalizer, apart from the file sim_equ3_rtdx.c that was described above, must also include the files *intvecs.asm*, *rtdx_buf.c* and *rtdx.cmd*. In file *rtdx_buf.c* the size of the buffer, that the DSP is going to use to transfer data using the RTDX technology, is defined. In this particular case the size of this buffer should be greater than 512*4 + 8 =2056 bytes. The files *intvecs.asm*, *rtdx_buf.c* and *rtdx.cmd* are developed by TI and they are included in CCS. In the project the libraries *rtdx.lib* (or *rtdxsim.lib* if a simulator is being used) and *rts6701.lib*) must also be added. In Figure 125 the View Window of the project in CCS is presented.

Figure 125. The View Window of the equalizer project in CCS

4.1.5 Implementation of a VI to control the graphical equalizer

In order to control the graphical equalizer, a VI named Equalizer.vi has been developed. This VI sets, controls and communicates with the CCS automatically, so it orivdes the user with the capability to alter the amplification of each frequency band without interruptin the function of the equalizer. The user,also can watch the input and output signals of the equalizer at any time.

The front panel of the Equalizer.vi

In Figure 126 the front panel of the Equalizer.vi that controls the equalizer is presented.

Figure 126. The front panel of Equalizer.vi that controls the graphical equalizer

At the front panel there are three sliders named “Low_pass”, “Band_pass” and “High_pass” determining the amplification of the respective frequency band. Below these sliders an array has been placed with three elements named “gains”. Each element corresponds to a slider’s value. Below of the sliders a radio button has also been placed that nulls the value of the respective slider and deactivates it. The Radio Buttons’ names are “Low_pass_mute”, “Band_pass_mute” and “High_pass_mute”. The “Stop” button terminates the function of the VI, when pushed. The indicator “Build Result” shows the results of the build process of the project in CCS. The indicator “Input_signal” depicts the equalizer’s input signal; similarly the indicator “Output_signal” depicts the equalizer’s output signal. The indicator “Output & Input signals” depicts the input (green color) and output (red color) signal simultaneously.

The block diagram of the Equalizer.vi

The block diagram of the Equalizer.vi, shown in Figure 127, has been separated into four phases, in order for its function to become more understandable.

Figure 127. The block diagram of the Equalizer.vi

In the first phase of the block diagram, shown in Figure 128, the CCS is being set up so the DSKC6713 is to be used.

Figure 128. The first phase of the block diagram of Equalizer.vi

The subVI CCS_ Setup_Open.vi loads the CCStudio Setup without being visible by the user, because in the input Visible of the subVI a Boolean constant has been connected with value False. The CS_Setup_ Clear.vi clears all previous settings from the CCStudio Setup. The CCS_Setup_Add_ Board.vi loads the DSKC6713 drivers to the CCStudio Setup according to the path indicated by the constant “Path”, since it is connected with the input “Driver Path” of the subVI. The CCS_Setup_Save.vi saves the settings made and the CCS_ Setup_Close.vi closes the CCStudio Setup.vi. In this point the first phase of the Equalizer.vi has been completed. Therefore, the CCS is now set to use the DSKC6713.

In the second phase of the block diagram, shown in Figure 129, the actions that have to be made, in order for the DSKC6713 to start the execution of the program (the equalizer), are automated.

Figure 129. The second phase of the block diagram of Equalizer.vi

The Equalizer.vi should be saved in the same folder that the CCS project has also been saved, so for the path of the project to be correct. The Current VI's Path.vi outputs the path of the Equalizer.vi that is given as input to the Strip Path.vi. The output "stripped path" of the Strip Path.vi contains the path that was given as input without the past part of it. For example, if the path of the Equalizer.vi was C:\dsp_applications\DSKC6713\sim_equ3_rtdx\ Equalizer.vi then the output "striped path" of the Strip Path.vi would contain the path C:\dsp_applications\DSKC6713\ sim_equ3_rtdx. The output "striped path" of the Strip Path.vi is connected with the input "Base Path" of Build Path .vi. In input "name or relative path" of the Build path .vi the string constant "project name" is connected, that includes the name of the CCS project, which is sim_equ3_rtdx.pjt. The Build Path.vi outputs in "appended path" a new path that consists of the input "Base Path" and the content of input "name or relative path". If the content of input "Base Path" is C:\dsp_applications\DSKC6713\ sim_equ3_rtdx The content of the output "appended path" would be C:\dsp_applications\DSKC6713\ sim_equ3_rtdx\ sim_equ3_rtdx.pjt. The output "appended path" of Build Path.vi is converted to string using the Path to String.vi. The output of the Path to String.vi is connected to input "Project Path IN" of the CCS_Open_Project.vi. In this way the CCS_Open_Project.vi takes the input of the CCS project as long as the Equalizer.vi has been saved in the same folder as the CCS project. This pattern is followed to VIs that use file paths in order to be transferred to different work stations without any extra modifications.

The CCS_Open.vi loads the CCS and the CCS_Open_Project.vi loads the CCS project according to the path included in input "Project Path In". The CCS_Connect.vi connects the CCS with the board, which is the DSKC6713. The CCS_DSP_Reset.vi resets the board. The CCS_Build.vi orders the CCS to build the project that was loaded, so for the executable file will be generated. The build process is completed, the CCS_Build_Result.vi outputs the building result to the indicator "Build Result". While no errors have occurred during the building process, the CCS_Download.vi orders the CCS to download the executable code to the DSP. Because the project uses the RTDX technology the CCS_RTDX_Enable.vi is used, that enables and controls the RTDX parameters. The constants cluster "RTDX Settings" that is connected to the respective input of the CCS_RTDX_Enable.vi contains the settings related to the RTDX technology. The constant "Buffer Size" defines the size of the host buffer that is going to be used and it will be 2056 bytes, the constant "Num Of Buffers" defines the number of the host buffers, that will be 4 and the constant "Mode" defines the RTDX method that will be used, which is the Non Continuous one. Finally the second phase of the block diagram is completed with the CCS_Run.vi, which through CCS, commands the DSP initiate the execution of the program.

The third phase of the block diagram of Equalizer.vi, shown in Figure 130, comprises the main code of the VI. The third phase is the content of the While structure, that allows

constant communication of the VI with the DSP, until some error occurs or until the user pushes the “Stop” button.

Figure 130. The third phase of the block diagram of Equalizer.vi

The sliders’ terminals “Low_pass”, “Band_pass” and “High_pass” are connected to the Build Array.vi that creates an array (sliders’ array) with three elements. The first array element contains the value of slider “Low_pass”, the second contains the value of slider “Band_pass” and th ethord one contains the value of slider “High_pass”.

The Radio Buttons “Low_pass_mute”, “Band_pass_mute” and “High_pass_mute” whe the have a True value (meaning they are selected in the front panel by the user) should coause the respective slider’s value in the sliders array to null and to disable the slider. When the Radio Buttons have a False value (are not selected) should cause no change to the respective slider’s value εν while they should enable the slider if it is disabled. The function of the Radio Buttons is identical, so only the Radio Button “Low_pass_mute” will be analysed.

The “Low_pass_mute” is connected to the control input of the corresponding Case structure, shown in Figure 131. When the value of “Low_pass_mute” is True, the True case of the Case structure will be executed, in which the first element of the sliders array, that corresponds to slider “Low_pass”, will take the value 0 and the slider’s attribute “Disabled” will take the value 2. The elements’ value in the sliders arry changes with the use of the Replace Array Subset.vi. When the value of “Low_pass_mute” is False (not selected) the False case of the Case structure will be executed, in which the sliders array is not affected and

the attribute "Disabled" of slider "Low_pass" will take the value 0. The sliders' attribute "Disabled" defined if the respective slider will be enabled or disabled. If the attribute "Disabled" takes the value 0 the respective slider will be enabled, while if it takes the value 2 it will be disabled and turn into gray. This is the way that all other Radio Buttons work.

(a) **(b)**

Figure 131. The Case structure controlled by the Low_pass_mute **(a)** True case **(b)** False case

The indicator "gains" represents the sliders' array, by taking into consideration the values of the Radio Buttons. The sliders' array after it is multiplied by the constant 10 using the Multiply.vi it is converted to an array the elements of which are 2-byte (16 bits) signed integers using the Word Integer.vi. The To Word Integer.vi output contains now the data to be transferred to the DSP and they represent the amplification of each frequency band of the equalizer.

The data transfer from the VI to the DSP and vice versa is achieved in Flat Sequence structure. In the first Frame of the structure, the RTDX_Write.vi has been placed, in which the RTDX_Write_SA_I2 choice is selected allowing arrays consisting of 2-byte (16 bits) signed integers to be send. In input "Data" of the RTDX_Write.vi the output of To Word Integer.vi is connected that contains the data to be send. The string constant "Channel" that is connected to the RTDX_Write.vi indicates that for the data transfer the RTDX channel "gains_chan" is used.

In the second Frame of the Flat Sequence structure a delay of 1500 msec is occurred during the function of the Equalizer.vi. While in this time period, the DSP should receive the data, process them and send the process results to the VI through the respective RTDX channels. The data transfer speed from and to the DSP (which is the most time consumed process) depends on the capabilities of the specific work station, so the above time period could be easily reduced. The function delay of the VI is achieved using the Wait(ms).vi. The arithmetic constant connected to input "milliseconds to wait" defines the time-delay of the VI.

In the third Frame of the Flat Sequence structure, data sent by the DSP are received and represented. The RTDX_Read.vi in which the RTDX_Read_SA_I2 choice is selected reads data by the RTDX channel indicated by the string constant "Channel", which are the data in RTDX channel "signal_chan". The RTDX channel "signal_chan" contains a 512-element array (2-byte signed integers) that represents the input signal. The output "Data" of the RTDX_Read.vi is connected to the indicator "Input_signal" that represents input signal. The output "Data" is connected to the input of "To Long Integer" in order for the elements of the array to be converted to 4-byte signed integers. The RTDX_Read.vi is called again to but now the RTDX_Read_SA_I4 choice is selected, to read the data in the RTDX channel indicated by the constant "Channel1" , which are the data in the RTDX channel "fsignal". The RTDX channel "fsignal" transfers a 512-element array (4-byte signed integers) representing the output signal of the equalizer. The output "Data" of the RTDX_Read.vi is connected to the indicator "Output_signal" to represent the output signal. The output "Data" of the

RTDX_Read.vi and the output of the To Long Integer.vi are connected to inputs of Build Array.vi. The Build Array.vi creates a two-dimension array and its output is connected to the indicator "Output & Input signals" in order to represent the input and output signal of the equalizer, simultaneously.

After the third Frame of the structure Flat Sequence is completed a check is being made to see if any error occurred or if the "Stop" button was pushed in order to terminate the process of the While structure (third phase of the block diagram of the Equalizer.vi). This is the reason why the "Status" outputs of the output cluster "error out" that is returned from the Flat Sequence structure and the terminal icon of the "Stop" button are connected to the inputs of the Or.vi. The Or.vi works just like an OR gate, therefore the output of the Or.vi takes the True value if any error occurs or if the "Stop" button is pushed. The output of the Or.vi is connected with the conditional terminal of the While structure, so if the output of the Or.vi is True the While structure is not going to be executed again otherwise the process that was just described will be repeated. In this point, not only the third phase of the Equalizer.vi is completed but the maid code of the VI as well.

In the fourth phase of the Equalizer.vi, shown in Figure 132, the termination of the project and the CCS is being done.

Figure 132. The fourth phase of the block diagram of the Equalizer.vi

The CCS_DSP_Halt.vi commands the DSP, through CCS, to stop the execution of the program, while the CCS_RTDX_Disable.vi disables the RTDX technology. The CCS_Close_Project.vi closes the CCS project and the CCS_Close.vi closes the CCS. In this point the fourth and final phase of the Equalizer.vi is completed.

4.1.6 Results – Conclusions

In order to control the function of the graphical equalizer, a comparison is going to be made, between the output, for different sliders' values, as represented in Equalizer.vi and the output of the respective filters in MATLAB.

In Figure 133 the front panel of the Equalizer.vi is presented, when it functions and the amplification of the low frequencies is selected to be 1, while the amplification of the middle-range and high frequencies is 0.

Figure 133. The front panel of the Equalizer.vi

In Figure 134 the input signal of the equalizer is shown as represented by the Equalizer.vi and in MATLAB. From Figure 134 is concluded that the input signal is exactly the same in both cases.

(a) **(b)**

Figure 134. The input signal of the equalizer **(a)** in Equalizer.vi **(b)** in MATLAB

When the amplification of the low-frequencies is selected to be 1, while the amplification of the middle-range and high frequencies is 0, the output signal of the equalizer is the output of a low-pass filter as the Figure 135 shows.

(a) **(b)**

Figure 135. (a) The output signal of the equalizer in Equalizer.vi
(b) The output of the low-pass filter in MATLAB

In Figure 136 the front panel of the Equalizer.vi is presented when it functions and the amplification of the middle-range frequencies is selected to be 1, while the amplification of the low and high frequencies is 0.

Figure 136. The front panel of the Equalizer.vi

When the amplification of the middle-range frequencies is selected to be 1, while the amplification of the low and high frequencies is 0, the output signal of the equalizer is the output of a band-pass filter as Figure 137 shows.

(a) **(b)**

Figure 137. (a) The output signal of the equalizer in Equalizer.vi
(b) The output of the band-pass filter in MATLAB

In Figure 138 the front panel of the Equalizer.vi is presented when it functions and the amplification of the high frequencies is selected to be 1, while the amplification of the middle-range and low frequencies is 0.

Figure 138. The front panel of Equalizer.vi

When the amplification of the high frequencies is selected to be 1, while the amplification of the middle-range and low frequencies is 0, the output signal of the equalizer is the output of a high-pass filter as the Figure 139 shows.

(a) **(b)**

Figure 139. (a) The output signal of the equalizer in Equalizer.vi
(b) The output of the high-pass filter in MATLAB

From the above Figures it can be easily concluded that the graphical equalizer described above functions properly since results of the Equalizer.vi and MATLAB are in perfect match.

In order to design a tenth -or more- band graphical equalizer in MATLAB, firstly the m-file that calculates the filter's coefficients of the equalizer, must be altered to meet the new specifications. For the creation of the file that will contain the filters' coefficients *save_equ_coefs()* function will be used without any modification.

In order to design a tenth -or more- band graphical equalizer in CCS, the delays (f0_dly, f1_dly, f2_dly etc.) for each filter must be defined in the C code of the project, described in paragraph 4.1.4. Furthermore the gains and data_out arrays must have as many elements as the bands of the equalizer (for a tenth-band equalizer they should have ten elements). The function that implements the IIR filters is the *iir_cas5()* and no modification is needed, while it must be called inside the main function as many times as the equalizer's bands. The

computation of each output sample of the equalizer is made by the sum of all filters' results (the results of the *iir_cas5()* function).

In order to control the tenth -or more- band graphical equalizer in LabVIEW, the Equalizer.vi, described in paragraph 4.1.5, should be properly modified. In the front panel of the Equalizer.vi the number of the sliders, the Radio Buttons and the "gains" array's elements must be equal to the equalizer's bands. In the block diagram, the pattern presented for the "Low_pass" slider, and the Radio Button "Low_pass_mute", should be repeated for each slider and Radio Button.

With the modifications described above, the three-band graphical equalizer can be easily be converted in a tenth -or more- band graphical equalizer.

4.2 A Digital Image Processing Application

The digital image processing application that will be described below deals with implementation of simple image processing algorithms on DSKC6713 controlled by a GUI that is created in LabVIEW. In this application, the histogram equalization on a color image can be easily accomplished, while edge detection or direct and inverse Discrete Cosine Transform – DCT will be applied on a grayscale image. Furthermore, an image will be encoded and decoded according to the JPEG standard. It must be noticed that the image as well as its type and the processing algorithm are selected through the GUI.

Form each processing algorithm a CCS project has been created, therefore the project and the respective code that will be loaded to CCS and executed by the DSP is selected through the GUI. The image is loaded to the DSP during the building process through one or three header files according to the type of the image. The GUI though should create one header file that will contain the grayscale image or three header files that will contain the R, G and B components of the colored image. The processing results are transferred to the GUI using the RTDX technology.

In the following paragraphs the creation of the CCS project for each algorithm as well as the development of the GUI in LabVIEW is described thoroughly.

4.2.1 Edge Detection

One of the most simple image processes is the edge detection. Edges are detected in points, where obvious differences in brightness, between neighbor pixels of an image, appear. The detection of these edges is achieved by correlating the image with a linear filter which is an approximation of the first derivative, since if the first derivative is calculated along the image, then the points where it is appeared to be maximum, corresponds to its edges.

Therefore, the edge detection in every direction is archived by correlating the image with a linear filter. This filter is nothing more than a 3x3 window (or mask) which scans sequentially all the image. As already known, the edges are computated in two directions (horizontal and vertical), one 3x3 window is required for each.

The most common pattern for edge detection is the Sobel edge detector. In fact it is the correlation of two 3x3 masks (Figure 140) with an image that will find the edges on it.

-1	-2	-1
0	0	0
1	2	1

(a)

-1	0	1
-1	0	2
-1	0	1

(b)

Figure 140 Sobel masks for edge detection on an image:
(a) along vertical direction (Gy) and **(b)** along horizontal direction (Gx)

Implementation of Sobel edge detection

The C code of the DSP_Sobel_rtdx.c that implements the Sobel edge detection on a grayscale image is presented below.

```
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <rtdx.h>                    /* RTDX_Data_Read          */
#include <stdio.h>                   /* printf                  */
#include "target.h"                  /* TARGET_INITIALIZE       */

#define IMAGE_SIZE 65536
#define H 256
#define W 256
#define MAX_MESSAGES 256
#define MAX_ELEMENTS 256

#include "scenary.h"                 /* contains input image as a 1D array */

#pragma DATA_SECTION (image_in, "IM_in")
#pragma DATA_SECTION (image_out,"IM_out")
#pragma DATA_SECTION (Gx, "Gx_var")
#pragma DATA_SECTION (Gy, "Gy_var")
#pragma DATA_SECTION (message, "msg_var")

unsigned char image_in[IMAGE_SIZE];
unsigned char image_out[IMAGE_SIZE];
unsigned char Gx[IMAGE_SIZE];
unsigned char Gy[IMAGE_SIZE];
unsigned char message[MAX_ELEMENTS];

RTDX_CreateOutputChannel(ochan); /* Channel to use to write data */

// Find Sobel edges

void Sobel_edges()
{
  int i;
  int w00, w01, w02;
  int w10,       w12;
```

```
    int w20, w21, w22;
    int x_edge, y_edge;
    int f_edge ;

    for(i=0; i<IMAGE_SIZE; i++)                    // Clear arrays
    {
        Gx[i] = 0;
        Gy[i] = 0;
        image_out[i] = 0;
    }

    for (i=0; i<(W*(H-2)-2); i++)
    {
        w00 = image_in[i];
        w01 = image_in[i+1];
        w02 = image_in[i+2];
        w10 = image_in[W+i];
        w12 = image_in[W+i+2];
        w20 = image_in[(2*W)+i];
        w21 = image_in[(2*W)+i+1];
        w22 = image_in[(2*W)+i+2];

        x_edge = - w00 - 2*w10 - w20 + w02 + 2*w12 + w22 ;
        y_edge = - w00 - 2*w01 - w02 + w20 + 2*w21 + w22 ;
        f_edge = (abs(x_edge) + abs(y_edge))/2 ;

        if (f_edge>255) f_edge = 255;
        if (f_edge<0)     f_edge = 0;

        image_out[i+1] = f_edge;

        if (x_edge>255) x_edge = 255;              // View Gx & Gy edges
        if (x_edge<0)     x_edge = 0;
        if (y_edge>255) y_edge = 255;
        if (y_edge<0)     y_edge = 0;
        Gx[i+1] = x_edge;
        Gy[i+1] = y_edge;
    }                                              // end for loop
}

// Main program

void main ()
{
    int i, j;

    TARGET_INITIALIZE();              // Target initialization for RTDX

    Sobel_edges();

    RTDX_enableOutput(&ochan);        // Enable the output channel, "ochan"

    for (i=0; i<IMAGE_SIZE; i+=MAX_ELEMENTS)
    {
        for (j=i; j<(i+MAX_ELEMENTS); j++)
```

```
        {
            message[j-i] = (unsigned char) image_out[j];          //write one row (256 elements)
        }

        if (!RTDX_write( &ochan, message, sizeof(message)))   //Send the data to the host
        {
            fprintf(stderr, "\nError: RTDX_write() failed!\n");
            abort();
        }
    }

    puts("\n Program Completed!");

}                                          // End main program
```

Program 6. The C code of the DSP_Sobel_rtdx.c

In Program 6 the files *stdlib.h*, *string.h*, *math.h*, *rtdx.h*, *stdio.h*, *target.h*, and *scenary.h* are included. The *rtdx.h* contains the declarations of the functions related to the RTDX technology. The *target.h* contains the declaration of the TARGET_INITIALIZE() function that initializes the DSP and activate s the interrupts in order to enable the RTDX technology. The *scenary.h* the image to be processed with dimension of 256 x 256 pixels. The *#pragma* directives define the memory areas, where the arrays related to the image are saved. The allocation of the DSP memory is achieved by the *rtdx_sobel.cmd* (Program 7)

Before the declaration of the *main()* function the RTDX channel "ochan" is defined that will transfer the processing result from DSP to GUI. The channel "ochan" transfers from DSP an array of 256 numbers, that represents a line from the final image, to GUI.

In the *main()* function, firstly the DSP is initialized and the RTDX technology is enabled. Then the edges of the image are detected by using the *Sobel_edges()* function, which is declared right before the *main()* function. The RTDX channel output "ochan" is enabled and the image is being transferred from the DSP to GUI. This transfer of the final image is carried out gradually meaning that each time only one line of the final image is written to the RTDX channel.

```
-c
-heap  0x1000
-stack 0x1000
-u __vectors
-u _auto_init

_HWI_Cache_Control = 0;
_RTDX_interrupt_mask = ~0x000001808;

MEMORY
{
  VECS:     o=00000000h   l=00000200h   /* interrupt vectors       */
  PMEM:     o=00000200h   l=0000FE00h   /* Internal RAM (L2) mem   */
  BMEM:     o=80000000h   l=01000000h   /* CE0, SDRAM, 16 Mbytes   */
}

SECTIONS
```

```
{
  .intvecs    >  0h
  .text       >  BMEM
  .rtdx_text  >  BMEM
  .far        >  BMEM
  .stack      >  BMEM
  .bss        >  BMEM
  .cinit      >  BMEM
  .pinit      >  PMEM
  .cio        >  BMEM
  .const      >  BMEM
  .data       >  BMEM
  .rtdx_data  >  BMEM
  .switch     >  BMEM
  .sysmem     >  BMEM

  IM_in       >  BMEM
  IM_out      >  BMEM
  Gx_var      >  BMEM
  Gy_var      >  BMEM
  msg_var     >  BMEM
}
```

Program 7. The rtdx_sobel.cmd

The project that will be created for the edge detection on a grayscale image is RTDX_Sobel_edges.pjt. Apart from the files *DSP_Sobel_rtdx.c* and *rtdx_sobel.cmd* decribed above, the *intvecs.asm,* that was developed by TI and is included in CCS, should also be added. For the DSP_Sobel_rtdx.c the local optimization File (-o3) is selected. Furthermore, the libraries *rtdx.lib* (or *rtdxsim.lib* if a simulator is used) and *rts6701.lib* should be added as well. In Figure 141 the View Window of the CCS project is presented.

Figure 141. The CCS View Window for the RTDX_Sobel_edges.pjt

4.2.2 Direct and Inverse Discrete Cosine Transformation

The Discrete Cosine Transform or DCT represents the most common option for the transformation of an image from the spatial domain to the frequency domain, because of it's attributes such as the energy compaction. This is the reason why it is adopted by the most coding standards regarding not only still images, like JPEG and MPEG-7, but the video like MPEG-1, MPEG-2 and MPEG-4, as well.

One of the characteristics of the DCT is the energy compaction. This means that when it is applied in every 8x8 block of the image, the major percentage of the energy is compacted to low frequencies and especially to the 1° coefficient of each block, called DC coefficient. The rest 63 coefficients are called AC coefficients and they cover very small percentage of energy compared to the DC coefficient. The arrangement of the 8x8 DCT coefficients is shown in Figure 142.

The low frequencies are located to the upper left corner of the block, while the high frequencies are located to the lower right corner of the block. The energy of a DCT coefficient corresponds actually to the information included, so the DC coefficient is the one that contain the major percentage of information of each 8x8 block, while the AC coefficients contain a very small percentage of the information that corresponds to the details of the 8x8 block of image. Therefore, by applying the DCT in every 8x8 block of an image, the energy (and consequently the information) of each block is compacted to the low frequencies, which is actually the DC coefficient, few of the neighbor AC coefficients (AC_{01}, AC_{10}, AC_{11}, etc). The rest AC coefficients, mainly in high frequencies are redundant information and they can be removed (take 0 values) during quantization.

Figure 142. Arrangement of DCT coefficients in each 8x8 block

The two-dimension DCT is a very demanding algorithm, since it requires a great number of calculations (additions, multiplications and abstractions). So, the implementation of the 2D-DCT on an image based on its definition (equation 9.1) has no meaning at all, since it cannot be applied in real-time. For this reason, the 2D-DCT is carried out with two 1D-DCTs first on lines and then on rows of each 8x8 block of the image. This implementation is consider to be slow (seconds), but it is though fastest than the direct implementation of the 2D-DCT. An optimized algorithm that implements the DCT, based on McGovern's algorithm is presented below.

In order to control the accurate function of the algorithms, apart from the direct DCT, the inverse DCT (IDCT) is also applied. The DCT belongs to the orthogonal transformations, a basic attribute of which is the reversibility. This means that by expressing the DCT as a product of arrays, meaning $D' = W*D$ (where W is the transform array, D is the array of data)

and multiplying with the inverted transformation array W^T, the initial array D is obtained again. So by applying on an image the direct DCT and then the IDCT, the result is the rebuild of the initial image.

Implementation of the 2D-DCT using the 1D-DCT

For the implementation of the 2D-DCT using the 1D-DCT, the files *Dct_Main_rtdx.c*, *dct_main.h*, *Dct.c*, *Idct.c* and *rtdx_DCTslow.cmd* have been created and they are presented below.

Program 8 shows the *Dct_Main_rtdx.c*, in which the *stdlib.h*, *time.h*, *rtdx.h* , *stdio.h*, *target.h*, *dct_main.h* and *scenary.h* are included. The header file *rtdx.h* contains the declarations of the functions related to the RTDX technology. The *target.h* includes the declaration of function TARGET_INITIALIZE(), which initializes the DSP and activates the interrupts so for the RTDX technology to be enabled. The *dct_main.h* contains the declarations of some constants, such as IMAGE_LEN and BLOCK_LEN and it is presented in Program 9. The scenary.h the image to be processed with dimension of 256 x 256 pixels. The #pragma directives define the memory areas, where the arrays related to the image are saved.. The allocation of the DSP memory is carried out in file *rtdx_DCTslow.cmd* (Program 13)

Before the declaration of the *main()* function, the RTDX channel "ochan" that transfers the processing result from DSP to GU, is defined. The channel "ochan" transfers from DSP an array of 256 numbers that represents a line from the final image, to GUI.

In *main()* function, the DSP is initialized DSP and the RTDX technology is enabled.In each 8x8 block of the image, the direct DCT is applied by calling the *dct()* function. Similarly the IDCT is applied by calling the *dct()* function. The *dct()* and *idct()* functions are going to be presented below. The RTDX channel output "ochan" is enabled and the image is being transferred from the DSP to GUI. This transfer of the final image is carried out gradually meaning that each time only one line of the final image is written to the RTDX channel.

```
#include <stdlib.h>
#include <time.h>
#include <rtdx.h>                    // RTDX_Data_Read
#include <stdio.h>                   // printf
#include "target.h"                  // TARGET_INITIALIZE

#define DSK6711_cps 150000000        //150 MHz C6711 CPU clock (cps -> Clocks Per Second)

#include "dct_main.h"                //Includes and Constants used
#pragma DATA_SECTION (image_in,"myvar0")
#pragma DATA_SECTION (image_out,"myvar1")
#include "scenary.h"                 //An h file containing input image as a 1D array

unsigned char image_out[IMAGE_SIZE];
short block[BLOCK_SIZE];

clock_t start, stop;
double duration;

#pragma DATA_SECTION (message, "msg_var")

RTDX_CreateOutputChannel(ochan);          //Channel to use to write data
```

```
#define MAX_MESSAGES 256
#define MAX_ELEMENTS 256

unsigned char message[MAX_ELEMENTS];

//Q12 DCT coefficients (actual coefficient x 2^12 )

const short coe[8][8]=
        {
            4096,   4096,   4096,   4096,   4096,   4096,   4096,   4096,
            5681,   4816,   3218,   1130,  - 1130,  -3218,  -4816,  -5681,
            5352,   2217,  -2217,  -5352,  -5352,  -2217,   2217,   5352,
            4816,  -1130,  -5681,  -3218,   3218,   5681,   1130,  -4816,
            4096,  -4096,  -4096,   4096,   4096,  -4096,  -4096,   4096,
            3218,  -5681,   1130,   4816,  -4816,  -1130,   5681,  -3218,
            2217,  -5352,   5352,  -2217,  -2217,   5352,  -5352,   2217,
            1130,  -3218,   4816,  -5681,   5681,  -4816,   3218,  -1130
        };

//FUNCTIONS USED

void dct(void);                         // dct.c
void idct(void);                        // idct.c

//MAIN FUNCTION
void main()
{
   int row, col, x, y;
   int i,j;

   TARGET_INITIALIZE();                 //Target initialization for RTDX

   start = clock();

//FORWARD DCT/ INVERSE DCT
   // block by block processing
   for (row=0; row<IMAGE_LEN; row+=BLOCK_LEN)
   {
        for (col=0; col<IMAGE_LEN; col+=BLOCK_LEN)
        {
            for (y=0, i=0; y<BLOCK_LEN; y++)        // get the block from the input image
            {
                for (x=0; x<BLOCK_LEN; x++, i++)
                    block[i] = (short) image_in[(col+x)+(row+y)*IMAGE_LEN];
            }
            dct();                                          //perform FDCT on this block
            idct();                                         //perform IDCT on this block
            for (y=0, i=0; y<BLOCK_LEN; y++)                // store block to output image
            {
                for (x=0; x<BLOCK_LEN; x++, i++)
                {                       //Quick fix for errors occurring due to negative a values
                    if(block[i]<0)      //occurring after IDCT!*/
                        image_out[(col+x)+(row+y)*IMAGE_LEN]=(unsigned char) (-block[i]);
                    else
                        image_out[(col+x)+(row+y)*IMAGE_LEN]=(unsigned char) block[i];
```

```
                }
            }
        }
    }

    stop = clock();
    duration = (double) (stop - start) / DSK6711_cps;

    RTDX_enableOutput(&ochan);                  // Enable the output channel, "ochan"

    for (i=0; i<IMAGE_SIZE; i+=MAX_ELEMENTS)
    {
        for (j=i; j<(i+MAX_ELEMENTS); j++)
        {
            message[j-i] = (unsigned char) image_out[j];          //Write one row (256 elements)
        }

        if (!RTDX_write( &ochan, message, sizeof(message) ) ) //Send the data to the host
        {
            fprintf(stderr, "\nError: RTDX_write() failed!\n");
            abort();
        }
    }

    puts("\nProgram Completed!");
    printf( "\n %s ", " The algorithm implementation completed in");
    printf( "%5.5f %s \n\n", duration, "seconds" );
    puts(" Completed Successfully!\n");
}
```

Program 8. The C code of the dct_Main_rtdx.c

In Program 9 the header file *dct_main.h* is presented, which contains the declarations of some constants, such as IMAGE_LEN and BLOCK_LEN, with values of 256 and 8 respectively. In addition, the constants IMAGE_SIZE and BLOCK_SIZE (with values of 65536 and 64) that correspond to image's size and block's size used, are also defined.

```
#ifndef DCT_MAIN_H

#define IMAGE_LEN 256
#define IMAGE_SIZE (IMAGE_LEN*IMAGE_LEN)

#define BLOCK_LEN 8
#define BLOCK_SIZE (BLOCK_LEN*BLOCK_LEN)

#endif
```

Program 9. The dct_main.h header file

In Program 10 the *Dct.c* is presented, which contains the *dct()* function that implements the 2D-DCT in an 8x8 block of the image, using the 1D-DCT. In the beginning of the code the arrays image_in and image_out are declared as **extern** image_in and image_out (they contain the data of the initial and the rebuilt image respectively) as well as the arrays block

and coe. The block array contains 64 elements that correspond to the 64 DCT coefficients of one 8x8 block, while the array coe contains the 64 values of the $C'(k)\cos[(2i+1)k\pi/16]$ terms, multiplied by 2^{12} (Q-12 format).

A variable is declared with the key-word extern, when it is needed to be visible by other code files that are included in the same project. This is happening in a case where a variable is called in more than one C code files of the same project. In this specific case, the four arrays that have been declared as extern are visible so they can be used by the *Dct.c*, *Idct.c* and *Dct_Main.c*.

```
#include "dct_main.h"
extern unsigned char image_in[IMAGE_SIZE];
extern unsigned char image_out[IMAGE_SIZE];
extern short block[BLOCK_SIZE];
extern const short coe[8][8];

void dct(void)
{
  int i,j,x,y;
  int value[8];

  for(j=0;j<8;j++)                //perform 1D DCT on the columns
  {
      for(y=0;y<8;++y)
      {
          value[y]=0;
          for(x=0;x<8;++x)
              value[y] += (int)(coe[y][x]*block[j+(x*8)]);
      }
      for(y=0;y<8;++y)
          block[j+(y*8)] = (short)(value[y]>>12);
  }

  for(i=0;i<64;i+=8)              // perform 1D DCT on the resulting rows
  {
      for(y=0;y<8;++y)
      {
          value[y] = 0;
          for(x=0;x<8;++x)
              value[y] += (int)(coe[y][x]*block[i+x]);
      }
      for(y=0;y<8;++y)
          block[i+y] = (short)(value[y]>>15);
  }
}
```

Program 10. The C code of the Dct.c

In Program 11 that follows, the implementation code of the 2D-IDCT is described. Here, like the direct DCT, the arrays image_in, image_out, block and coe are declard as extern. This file implements the *idct()* function, which actually is the 8x8 block 2D-IDCT and replaces to the block the 64 values of the DCT coefficients with the 64 values of the rebuilt pixels.

```
#include "dct_main.h"
extern unsigned char image_in[IMAGE_SIZE];
extern unsigned char image_out[IMAGE_SIZE];
extern short block[BLOCK_SIZE];
extern const short coe[8][8];

void idct(void)
{
  int i,j,x,y;
  int value[8];

  for(j=0;j<8;j++)      // perform 1D IDCT on the columns
  {
      for(y=0;y<8;++y)
      {
          value[y] = 0;
          for(x=0;x<8;++x)
              value[y] += (int)(coe[x][y]*block[j+(x*8)]);
      }
      for(y=0;y<8;++y)
          block[j+(y*8)] = (short)(value[y]>>12);
  }

  for(i=0;i<64;i+=8)            // perform 1D IDCT on the resulting rows
  {
      for(y=0;y<8;++y)
      {
          value[y] = 0;
          for(x=0;x<8;++x)
              value[y] += (int)(coe[x][y]*block[i+x]);
      }
      for(y=0;y<8;++y)
          block[i+y] = (short)(value[y]>>15);
  }
}
```

Program 11. The C code of the Idct.c

The memory allocation is completed in *rtdx_DCTslow.cmd* and it is presented in Program 12.

```
-c
-heap  0x1000
-stack 0x1000
-u __vectors
-u _auto_init


_HWI_Cache_Control = 0;
_RTDX_interrupt_mask = ~0x000001808;

MEMORY
{
```

```
    VECS:     o=00000000h    l=00000200h    /* interrupt vectors      */
    PMEM:     o=00000200h    l=0000FE00h    /* Internal RAM (L2) mem  */
    BMEM:     o=80000000h    l=01000000h    /* CE0, SDRAM, 16 MBytes  */

  }

  SECTIONS
  {
    .intvecs    >   0h
    .text       >   BMEM
    .rtdx_text  >   BMEM
    .far        >   BMEM
    .stack      >   BMEM
    .bss        >   BMEM
    .cinit      >   BMEM
    .pinit      >   PMEM
    .cio        >   BMEM
    .const      >   BMEM
    .data       >   BMEM
    .rtdx_data  >   BMEM
    .switch     >   BMEM
    .sysmem     >   BMEM

    myvar0      >   BMEM
    myvar1      >   BMEM
  }
```

Program 12. The rtdx_DCTslow.cmd

Figure 143. The View Window of CCS for the RTDX_DCT_Slow.pjt

In Figure 143 the View Window of the CCS project, named RTDX_DCT_Slow.pjt, for the direct DCT and IDCT application is presented. It should be noticed that these applications are implemented on a grayscale image, using the 1D-DCT. Apart from the files *DCT_Main.c*, *Dct.c*, *Idct.c* and *rtdx_DCTslow.cmd* used in this project and described above, the files *intvecs.asm* that was developed by TI, must also be used. The local optimization File (-o3) is chosen for the *DCT_Main.c*, *Dct.c* and *Idct.c*. In addition, the libraries *rtdx.lib* (ot *rtdxsim.lib* for simulator use) and *rts6701.lib*, must be added as well.

Implementing the 2D_DCT using the McGovern algorithm

In order to implement the 2D-DCT using the McGovern algorithm, the *Dct_Main_rtdx.c*, *dct_main.h*, *Dct.c*, *Idct.c* and *rtdx_DCTfast.cmd* have been developed and they are analyzed below.

In Program 13 the *Dct_Main_rtdx.c*, which contains the files *stdlib.h*, *time.h*, *rtdx.h* , *stdio.h*, *target.h*, *dct_main.h* and *scenary.h*, is described. The *rtdx.h* contains the declarations of the functions related to the RTDX technology. The *target.h* includes the declaration of function TARGET_INITIALIZE(), which initializes the DSP and activates the interrupts so for the RTDX technology to be enabled. The *dct_main.h* that was analyzed in Program 9 contains the declarations of some constants. The *scenary.h* contains the 256x256 image to be processed.

Before the declaration of the *main()* function, the RTDX channel "ochan" that will transfer the process results from the DSP to GUI, is defined. The channel "ochan" transfers from DSP an array of 256 numbers, that represents a line from the final image, to GUI.

In the *main()* function, firstly the DSO is initialized and the RTDX technology is enabled. In each 8x8 block of the image the direct DCT is applied,by calling the *dct()* function and the and IDCT, by calling the *idct()* function. The *dct()* and *idct()* functions will be presebted below. The RTDX channel output "ochan" is enabled and the image is being transferred from the DSP to GUI. This transfer of the final image is carried out gradually meaning that each time only one line of the final image is written to the RTDX channel.

```
#include <stdlib.h>
#include <time.h>
#include <rtdx.h>                    // RTDX_Data_Read
#include <stdio.h>                   // printf
#include "target.h"                  // TARGET_INITIALIZE

#define DSK6711_cps 150000000        // 150 MHz C6711 CPU clock (cps -> Clocks Per Second)

#include "dct_main.h"                // Includes and Constants used
#pragma DATA_SECTION (image_in,"myvar0")
#pragma DATA_SECTION (image_out,"myvar1")
#include "scenary.h"                 // An h file containing input image as a 1D array

unsigned char image_out[IMAGE_SIZE];
short block[BLOCK_SIZE];

clock_t start, stop;
double duration;

#pragma DATA_SECTION (message, "msg_var")
```

```
RTDX_CreateInputChannel(ichan);     // Channel to receive data from
RTDX_CreateOutputChannel(ochan); // Channel to use to write data

#define MAX_MESSAGES 256
#define MAX_ELEMENTS 256

unsigned char message[MAX_ELEMENTS];

// Q12 DCT coefficients (actual coefficient x 2^12 )
const short coe[12]={3135,2217,7568,8410,-1598,6149,-10498,4816,-3686,-12586,8035,-1223};

// FUNCTIONS USED

void dct(void);                         // dct.c
void idct(void);                        // idct.c

//MAIN FUNCTION

void main()
{
  int i,j,x;

  TARGET_INITIALIZE();                  // Target initialization for RTDX

  start = clock();

//FORWARD DCT/ INVERSE DCT
  //block by block processing
  for(i=0;i<IMAGE_SIZE;i+=BLOCK_SIZE)
  {
      // get the block from the input image
      for(x=0;x<BLOCK_SIZE;++x)
          block[x] = (short) image_in[i+x];

      dct();                            // perform FDCT on this block
      idct();                           // perform IDCT on this block

      // store block to output image
      for(x=0;x<BLOCK_SIZE;++x)
      {
          if(block[x]<0)
              image_out[i+x]=(unsigned char) (-block[x]); //Quick fix for errors occurring due to
          else                                            //negative a values occurring after IDCT!
              image_out[i+x]=(unsigned char) block[x];
      }
  }

  stop = clock();
  duration = (double) (stop - start) / DSK6711_cps;

  RTDX_enableOutput(&ochan);                    //Enable the output channel, "ochan"

  for (i=0; i<IMAGE_SIZE; i+=MAX_ELEMENTS)
  {
      for (j=i; j<(i+MAX_ELEMENTS); j++)
```

```
        {
            message[j-i] = (unsigned char) image_out[j];//Write one row (256 elements)
        }

        if (!RTDX_write( &ochan, message, sizeof(message) ) ) //Send the data to the host
        {
            fprintf(stderr, "\nError: RTDX_write() failed!\n");
            abort();
        }
    }

    puts("\nProgram Completed!");
    printf( "\n %s ", " The algorithm implementation completed in");
    printf( "%5.5f %s \n\n", duration, "seconds" );
    puts(" Completed Successfully!\n");
}
```

Program 13. The C code of the DCT_Main_rtdx.c

In Program 14 the *Dct.c* is presented, in which the operation of the *dct()* function is defined. This function applies the 2D-DCT on a 8x8 block, using the fast 1D-DCT according to the McGovern algorithm.

```
#include "dct_main.h"

extern unsigned char image_in[IMAGE_SIZE];
extern unsigned char image_out[IMAGE_SIZE];
extern short block[BLOCK_SIZE];
extern const short coe[12];

void dct(void)
{
  short ADD[20];      /* Table of the addition coefficients */
  int M[12];          /* Table of the results of the multiplication */
  int postadd1,postadd2;
  int i,j;

  for(j=0;j<8;j++)
  {                   /* first set of additions */
    ADD[0]= (block[j]+block[56+j]);                /* x(0)+x(7) */
    ADD[1]= (block[24+j]+block[32+j]);             /* x(3)+x(4) */
    ADD[2]= (block[8+j]+block[48+j]);              /* x(1)+x(6) */
    ADD[3]= (block[16+j]+block[40+j]);             /* x(2)+x(5) */
    ADD[4]= (block[j]-block[56+j]);                /* x(0)+x(7) */
    ADD[5]= (block[48+j]-block[8+j]);              /* x(6)-x(1) */
    ADD[6]= (block[24+j]-block[32+j]);             /* x(3)-x(4) */
    ADD[7]= (block[16+j]-block[40+j]);             /* x(2)-x(5) */

    /* second set of additions, this is done so previous additions do not need to be repeated */
    ADD[8]= (ADD[0]+ADD[1]);
    ADD[9]= (ADD[0]-ADD[1]);
    ADD[10]=(ADD[2]+ADD[3]);
```

```
    ADD[11]=(ADD[2]-ADD[3]);
    ADD[12]=(ADD[4]+ADD[6]);
    ADD[13]=(ADD[5]+ADD[7]);
    ADD[14]=(ADD[9]+ADD[11]);
    ADD[15]=(ADD[4]+ADD[5]);
    ADD[16]=(ADD[12]+ADD[13]);
    ADD[17]=(ADD[6]+ADD[7]);
    ADD[18]=(ADD[8]+ADD[10]);
    ADD[19]=(ADD[8]-ADD[10]);

    * Multiplications carried out, note: here 14. Includes one over root eight term */
    M[0] = (int)(coe[0]*ADD[9]);
    M[1] = (int)(coe[1]*ADD[14]);
    M[2] = (int)(coe[2]*ADD[11]);
    M[3] = (int)(coe[3]*ADD[4]);
    M[4] = (int)(coe[4]*ADD[15]);
    M[5] = (int)(coe[5]*ADD[5]);
    M[6] = (int)(coe[6]*ADD[12]);
    M[7] = (int)(coe[7]*ADD[16]);
    M[8] =(int)(coe[8]*ADD[13]);
    M[9] =(int)(coe[9]*ADD[6]);
    M[10]=(int)(coe[10]*ADD[17]);
    M[11]=(int)(coe[11]*ADD[7]);

    /* post multiplication, additions + subtractions */
    block[j]=ADD[18];                                   /* y(0) */
    block[32+j]=ADD[19];                                /* y(4) */
    block[16+j]=(short)((M[0]+M[1])>>12);               /* y(2) */
    block[48+j]=(short)((M[1]-M[2])>>12);               /* y(6) */
    postadd1= M[6]+M[7];
    postadd2= M[7]+M[8];
    block[56+j]= (short)((M[3]+M[4]+postadd1)>>12);     /* y(7) */
    block[40+j]= (short)((M[4]+M[5]+postadd2)>>12);     /* y(5) */
    block[8+j]= (short)((M[9]+M[10]-postadd1)>>12);     /* y(1) */
    block[24+j]= (short)((postadd2-M[10]-M[11])>>12);   /* y(3) */
    }

  for(i=0;i<64;i+=8)
  {
    /* first set of addtions */
    ADD[0]=(block[i]+block[i+7]);                       /* x(0)+x(7) */
    ADD[1]=(block[i+3]+block[i+4]);                     /* x(3)+x(4) */
    ADD[2]=(block[i+1]+block[i+6]);                     /* x(1)+x(6) */
    ADD[3]=(block[i+2]+block[i+5]);                     /* x(2)+x(5) */
    ADD[4]=(block[i]-block[i+7]);                       /* x(0)-x(7) */
    ADD[5]=(block[i+6]-block[i+1]);                     /* x(6)-x(1) */
    ADD[6]=(block[i+3]-block[i+4]);                     /* x(3)-x(4) */
    ADD[7]=(block[i+2]-block[i+5]);                     /* x(2)-x(5) */

    /* second set of addtions, this is done so previous additions do not need to be repeated */
    ADD[8]=(ADD[0]+ADD[1]);
    ADD[9]=(ADD[0]-ADD[1]);
    ADD[10]=(ADD[2]+ADD[3]);
    ADD[11]=(ADD[2]-ADD[3]);
    ADD[12]=(ADD[4]+ADD[6]);
```

```
        ADD[13]=(ADD[5]+ADD[7]);
        ADD[14]=(ADD[9]+ADD[11]);
        ADD[15]=(ADD[4]+ADD[5]);
        ADD[16]=(ADD[12]+ADD[13]);
        ADD[17]=(ADD[6]+ADD[7]);
        ADD[18]=(ADD[8]+ADD[10]);
        ADD[19]=(ADD[8]-ADD[10]);

        /* Multiplications carried out, note: here 14. Includes one over root eight term */
        M[0]= (int)(coe[0]*ADD[9]);
        M[1]= (int)(coe[1]*ADD[14]);
        M[2]= (int)(coe[2]*ADD[11]);
        M[3]= (int)(coe[3]*ADD[4]);
        M[4]= (int)(coe[4]*ADD[15]);
        M[5]= (int)(coe[5]*ADD[5]);
        M[6]= (int)(coe[6]*ADD[12]);
        M[7]= (int)(coe[7]*ADD[16]);
        M[8]=(int)(coe[8]*ADD[13]);
        M[9]=(int)(coe[9]*ADD[6]);
        M[10]=(int)(coe[10]*ADD[17]);
        M[11]=(int)(coe[11]*ADD[7]);

        /* post multiplication, additions + subtractions */
        block[i]=(short)(ADD[18]>>3);                          /* y(0) */
        block[i+4]=(short)(ADD[19]>>3);                        /* y(4) */
        block[i+2]=(short)((M[0]+M[1])>>15);                   /* y(2) */
        block[i+6]=(short)((M[1]-M[2])>>15);                   /* y(6) */
        postadd1= M[6]+M[7];
        postadd2= M[7]+M[8];
        block[i+7]= (short)((M[3]+M[4]+postadd1)>>15);         /* y(7) */
        block[i+5]= (short)((M[4]+M[5]+postadd2)>>15);         /* y(5) */
        block[i+1]= (short)((M[9]+M[10]-postadd1)>>15);        /* y(1) */
        block[i+3]= (short)((postadd2-M[10]-M[11])>>15);       /* y(3) */
    }
}
```

Program 14. The C code of the Dct.c

In Program 15 the *Idct.c*, is presented, in which the operation of the *idct( )*is defined. This function applies the 2D-DCT on a 8x8 block, using the fast 1D-DCT according to the McGovern algorithm.

```
#include "dct_main.h"

extern unsigned char image_in[IMAGE_SIZE];
extern unsigned char image_out[IMAGE_SIZE];
extern short block[BLOCK_SIZE];
extern const short coe[12];

void idct(void)
{
```

```
short z[8],ADD[7];
int PA[4];
int M[13];
int i,j;

for(j=0;j<8;j++)
{                    /* pre-additions */
  ADD[0]=block[56+j]+block[40+j];          /* y(7)+y(5) */
  ADD[1]=block[56+j]-block[8+j];           /* y(7)-y(1) */
  ADD[2]=block[24+j]+block[40+j];          /* y(3)+y(5) */
  ADD[3]=block[j]+block[32+j];             /* y(0)+y(4) */
  ADD[4]=block[16+j]+block[48+j];          /* y(2)+y(6) */
  ADD[5]=block[24+j]-block[8+j];           /* y(3)-y(1) */
  ADD[6]=ADD[0]+ADD[5];

  /* multiplications */
  M[0]=(int)(coe[0] * block[16+j]);        /* A*y(2) */
  M[1]=(int)(coe[1] * ADD[4]);
  M[2]=(int)(coe[2]*block[48+j]);          /* C*y(6) */
  M[3]=(int)(coe[3]*block[56+j]);          /* D*y(7) */
  M[4]=(int)(coe[4]*ADD[0]);
  M[5]=(int)(coe[5]*block[40+j]);          /* F*y(5) */
  M[6]=(int)(coe[6]*ADD[1]);
  M[7]=(int)(coe[7]*ADD[6]);
  M[8]=(int)(coe[8]*ADD[2]);
  M[9]=(int)(coe[9]*block[8+j]);           /* J*y(1) */
  M[10]=(int)(coe[10]*ADD[5]);
  M[11]=(int)(coe[11]*block[24+j]);        /* L*y(3) */
  M[12]=(int)(block[32+j]<<1);             /* 2*y(4) */

  /* post additions */
  PA[0]=((M[0]+M[1])>>12);
  PA[1]=(int)ADD[3]-M[12];
  PA[2]=M[4]+M[7];
  PA[3]=M[7]-M[10];

  z[0]=ADD[3]+(short)PA[0];
  z[1]=ADD[3]-(short)PA[0];
  z[2]=(short)PA[1]+(short)((M[1]-M[2])>>12);
  z[3]=(short)PA[1]+(short)((M[2]-M[1])>>12);
  z[4]=(short)((PA[2]+M[6]+M[3])>>12);
  z[5]=(short)((PA[2]+M[5]+M[8])>>12);
  z[6]=(short)((PA[3]+M[6]+M[9])>>12);
  z[7]=(short)((PA[3]+M[8]-M[11])>>12);

  block[j]=z[0]+z[4];                      /* x(0) */
  block[8+j]=z[2]-z[5];                    /* x(1) */
  block[16+j]=z[3]+z[7];                   /* x(2) */
  block[24+j]=z[1]+z[6];                   /* x(3) */
  block[32+j]=z[1]-z[6];                   /* x(4) */
  block[40+j]=z[3]-z[7];                   /* x(5) */
  block[48+j]=z[2]+z[5];                   /* x(6) */
  block[56+j]=z[0]-z[4];                   /* x(7) */
  }
for(i=0;i<64;i+=8)
```

```
    {               /* pre-additions */
    ADD[0]=block[i+7]+block[i+5];                /* y(7)+y(5) */
    ADD[1]=block[i+7]-block[i+1];                /* y(7)-y(1) */
    ADD[2]=block[i+3]+block[i+5];                /* y(3)+y(5) */
    ADD[3]=block[i]+block[i+4];                  /* y(0)+y(4) */
    ADD[4]=block[i+2]+block[i+6];                /* y(2)+y(6) */
    ADD[5]=block[i+3]-block[i+1];                /* y(3)-y(1) */
    ADD[6]=ADD[0]+ADD[5];

    /* multiplications */
    M[0]=(int)(coe[0]*block[i+2]);               /* A*y(2) */
    M[1]=(int)(coe[1]*ADD[4]);
    M[2]=(int)(coe[2]*block[i+6]);               /* C*y(6) */
    M[3]=(int)(coe[3]*block[i+7]);               /* D*y(7) */
    M[4]=(int)(coe[4]*ADD[0]);
    M[5]=(int)(coe[5]*block[i+5]);               /* F*y(5) */
    M[6]=(int)(coe[6]*ADD[1]);
    M[7]=(int)(coe[7]*ADD[6]);
    M[8]=(int)(coe[8]*ADD[2]);
    M[9]=(int)(coe[9]*block[i+1]);               /* J*y(1) */
    M[10]=(int)(coe[10]*ADD[5]);
    M[11]=(int)(coe[11]*block[i+3]);             /* L*y(3) */
    M[12]=(int)(block[i+4]<<1);                  /* 2*y(4) */

    /* post additions */
    PA[0]=(int)((M[0]+M[1])>>12);
    PA[1]=(int)((ADD[3]-M[12])>>3);
    PA[2]=M[4]+M[7];
    PA[3]=M[7]-M[10];

    z[0]=(ADD[3]+(short)PA[0])>>3;
    z[1]=(ADD[3]-(short)PA[0])>>3;
    z[2]=(short)PA[1]+(short)((M[1]-M[2])>>15);
    z[3]=(short)PA[1]+(short)((M[2]-M[1])>>15);
    z[4]=(short)((PA[2]+M[6]+M[3])>>15);
    z[5]=(short)((PA[2]+M[5]+M[8])>>15);
    z[6]=(short)((PA[3]+M[6]+M[9])>>15);
    z[7]=(short)((PA[3]+M[8]-M[11])>>15);

    block[i+0]=(z[0]+z[4]);                      /* x[0] */
    block[i+1]=(z[2]-z[5]);                      /* x[1] */
    block[i+2]=(z[3]+z[7]);                      /* x[2] */
    block[i+3]=(z[1]+z[6]);                      /* x[3] */
    block[i+4]=(z[1]-z[6]);                      /* x[4] */
    block[i+5]=(z[3]-z[7]);                      /* x[5] */
    block[i+6]=(z[2]+z[5]);                      /* x[6] */
    block[i+7]=(z[0]-z[4]);   /* x[7] */
    }
}
```

Program 15. The C code of the Idct.c

The memory allocation is completed in *rtdx_DCTfast.cmd* and it is presented in Program 16.

```
-c
-heap  0x1000
-stack 0x1000
-u __vectors
-u _auto_init

_HWI_Cache_Control = 0;
_RTDX_interrupt_mask = ~0x000001808;

MEMORY
{
   VECS:    o=00000000h   l=00000200h    /* interrupt vectors        */
   PMEM:    o=00000200h   l=0000FE00h    /* Internal RAM (L2) mem    */
   BMEM:    o=80000000h   l=01000000h    /* CE0, SDRAM, 16 MBytes        */
}

SECTIONS
{
   .intvecs        >    0h
   .text           >    BMEM
   .rtdx_text      >    BMEM
   .far            >    BMEM
   .stack      >    BMEM
   .bss        >    BMEM
   .cinit          >    BMEM
   .pinit          >    PMEM
   .cio        >    BMEM
   .const      >    BMEM
   .data           >    BMEM
   .rtdx_data>     BMEM
   .switch         >    BMEM
   .sysmem         >    BMEM

   myvar0          >    BMEM
   myvar1          >    BMEM
   msg_var         >    BMEM
}
```

Program 16. The rtdx_DCTfast.cmd

Figure 144 the View Window of the CCS project, named RTDX_DCT_Fast.pjt, for the direct DCT and IDCT application is presented. It should be noticed that these applications are implemented on a grayscale image, using the 1D-DCT. Apart from the files *DCT_Main.c*, *Dct.c*, *Idct.c* and *rtdx_DCTfast.cmd* used in this project and described above, the files *intvecs.asm* that was developed by TI, must also be used. The local optimization File (-o3) is chosen for the *DCT_Main.c*, *Dct.c* and *Idct.c*. In addition, the libraries *rtdx.lib* (ot *rtdxsim.lib* for simulator use) and *rts6701.lib*, must be added as well.

Figure 144. The View Window of CCS for the RTDX_DCT_Fast.pjt

4.2.3 JPEG standard encoding and decoding

JPEG is one of the most common known standards for image encoding. Its full name is Joint Photographic Experts Group, which is the group that developed it. It is the first international image compression standard for color as well as grayscale pictures.

The JPEG standard defines four encoding modes. Of these four modes, the sequential, the progressive, and the hierarchical are lossy coding modes while the predictive mode is lossless. In this section, a part of the baseline JPEG coding is described, which follows the sequential codec because it is the one used more often. In Figure 145 the structure of one JPEG encoder.

Figure 145. The JPEG encoder

As shown in Figure 145, the JPEG encoding consists of three main steps: (i) The DCT transformation, (ii) the quantization) and (iii) the entropy encoding. Some specific operations on the original image must be done in order for it to enter encoder. The coefficient 128 must be abstracted from the value of each image's pixel, shifting the range of luminance values from [0 255] to [-128 127], with mean value zero. Next, the image is subdivided into 8x8

blocks, where the DCT transform is applied on each block, producing 64 new values that are called the DCT coefficients (one DC and 63 AC). Any algorithm that implements the DCT can be used, since the JPEG standard does not define a specific algorithm.

As it is already explained, the information of each 8x8 block has been limited to low frequencies, meaning the DC coefficient and some AC around it. Therefore the AC coefficients, in high frequencies mainly, can be pulled to zero, achieving better compression. The simplest way of pulling to zero the coefficients that do not carry any important information, is by dividing them with the appropriate values. This procedure is called quantization of the DCT coefficients. These values are included in the quantization table introduced by the JPEG standard, and it is shown in Figure 146. The size of this quantization table isι 8x8 because a 8x8 image block is processed every time.

JPEG quantization table

Luminance

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Figure 146. The quantization table that corresponds to luminance component

The inverse procedure of quantization is called dequantization and is completed during the decoding of the image. The dequantization procedure includes the multiplication of the already quantized coefficients (of every 8x8 block) with the respective values of the quantization table.

The third and last step of a JPEG encoder is the entropy encoding as shown in Figure 145. After the quantization of the DCT coefficients, many of them (in high frequencies mainly) have been pulled to zero. Therefore, the rest coefficients can be encoding using the Huffman or Arithmetic coding. It is must be noticed, that the DC coefficient of each 8x8 block is encoding in a different way (DPCM) than the AC coefficients (RLE). This is because the DC coefficients require more bits for its encoding compared to the AC coefficients. The part of the entropy encoding is not included in the JPEG encoding that follows, so it will not be described in detail.

After the entropy encoding, a bit sequence will arise, which is the JPEG encoded image (*.jpg file). In order for someone to see such an image, it must be decoded following the three steps shown in Figure 147.

Figure 147. The JPEG decoder

As shown in Figure 147, The decoding procedure consists of (i) the entropy decoding, (ii) the dequantization and finally the inverse DCT for the image reconstruction.

Partial Implementation of JPEG image encoding

In order to implement a part of JPEG encoder and decoder, the *jpeg.c* and *jpeg_main.h* have been developed, and they are described below.

In Program 17 the *jpeg.c* is presented, partially implementing the JPEG encoder and decoder, for a binary 256x256 image. The parts of the JPEG encoder that are implemented are the direct DCT, as well as the quantization procedure. The parts of the JPEG decoder that are implemented are the dequantization procedure and the IDCT for image reconstruction.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <rtdx.h>                          /* RTDX_Data_Read */
#include "target.h"                        /* TARGET_INITIALIZE */
#include "jpeg_main.h"                     /* Includes and Constants used */
#pragma DATA_SECTION (image_in,"myvar0")
#pragma DATA_SECTION (image_out,"myvar1")
#pragma DATA_SECTION (array,"myvar2")
#pragma DATA_SECTION (array2,"myvar3")

#include "scenary.h"                 /* An h file containing input image as a 1D array */
#pragma DATA_SECTION (message, "msg_var")

RTDX_CreateOutputChannel(ochan);           /* Channel to use to write data */

#define MAX_MESSAGES 256
#define MAX_ELEMENTS 256

unsigned char message[MAX_ELEMENTS];
unsigned char image_out[IMAGE_SIZE];
float array[H][W];
float array2[H][W];
int q_table[8][8] ={ {16,11,10,16,24,40,51,61},
                     {12,12,14,19,26,58,60,55},
                     {14,13,16,24,40,57,69,56},
                     {14,17,22,29,51,87,80,62},
                     {18,22,37,56,68,109,103,77},
                     {24,35,55,64,81,104,113,92},
                     {49,64,78,87,103,121,120,101},
                     {72,92,95,98,112,100,103,99}};

float cosine[8][8]= {
  {1.00000,  0.98078,  0.92388,  0.83147,  0.70711,  0.55557,  0.38268,  0.19509 },
  {1.00000,  0.83147,  0.38268, -0.19509, -0.70711, -0.98078, -0.92388, -0.55557 },
  {1.00000,  0.55557, -0.38268, -0.98078, -0.70711,  0.19509,  0.92388,  0.83147 },
  {1.00000,  0.19509, -0.92388, -0.55557,  0.70711,  0.83147, -0.38268, -0.98078 },
  {1.00000, -0.19509, -0.92388,  0.55557,  0.70711, -0.83147, -0.38268,  0.98078 },
  {1.00000, -0.55557, -0.38268,  0.98078, -0.70711, -0.19509,  0.92388, -0.83147 },
  {1.00000, -0.83147,  0.38268,  0.19509, -0.70711,  0.98078, -0.92388,  0.55557 },
```

```
  {1.00000, -0.98078,  0.92388, -0.83147,  0.70711, -0.55557,  0.38268, -0.19509 }};


/* image_in 1d --> 2d array */
void read_image()
{
   int i, j;

   for(i=0; i<H; i++)
        for(j=0; j<W; j++)
        {
             array[i][j]=image_in[W*i+j];
        }
}

/*DCT 8x8 blocks by 1-dimensional way. First rows and then columns. */
void dct()
 {
   int x,y,i,m,n;
   float t_array[8];
   float temp,c1;

 /*** 1D DCT on rows ***/
   for(m=0; m<H; m+=8) {
        for(n=0; n<W; n+=8) {
             for(x=m; x<m+8; x++) {
                  for(y=n; y<n+8; y++) {
                       temp = 0.0;
                       for(i=n; i<n+8; i++) {
                            temp += array[x][i] * cosine[i-n][y-n];
                       }
                        if((y-n)==0) c1 = invsqrt2;
                        if((y-n)>0)  c1 = 1;
                        array2[x][y] = (float)(temp * c1/2);
                  }
             }
        }
   }

 /*** 1D DCT on cols ***/
   for(n=0; n<W; n+=8) {
        for(m=0; m<H; m+=8) {
             for(y=n; y<n+8; y++) {
                  for(x=m; x<m+8; x++){
                       temp = 0.0;
                       for(i=m; i<m+8; i++){
                            temp += array2[i][y] * cosine[i-m][x-m];
                       }
                       if((x-m)==0) c1 = invsqrt2;
                       if((x-m)>0)  c1 = 1;
                       t_array[x-m] = (float)(temp * c1/2);
                  }
                  for(i=0; i<8; i++){
                       array2[m+i][y] = t_array[i];
                  }
```

```
            }
        }
    }
}

/* Quantization */
void quantization()
{
    int i, j, m, n;

    for(m=0; m<H; m+=8) {
        for(n=0; n<W; n+=8)     {
            for(i=m; i<m+8; i++) {
                for(j=n; j<n+8; j++) {
                    array2[i][j] = ROUND((float)(array2[i][j]/q_table[i-m][j-n]));
                }
            }
        }
    }
}

/*  Dequantization */
void dequantization()
 {
    int i, j, m, n;

    for(m=0; m<H; m+=8) {
        for(n=0; n<W; n+=8) {
            for(i=m; i<m+8; i++) {
                for(j=n; j<n+8; j++) {
                    array2[i][j]*=(q_table[i-m][j-n]);
                }
            }
        }
    }
}

/*  Inverse DCT 1-dimensional way (idct) */
void idct()
 {
    int x,y,i,m,n;
    float t_array[8];
    float temp,c1;

 /* IDCT on Rows */
    for(m=0; m<H; m+=8) {
        for(n=0; n<W; n+=8) {
            for(x=m; x<m+8; x++) {
                for(y=n; y<n+8; y++) {
                    temp = 0.0;
                    for(i=n; i<n+8; i++){
                        if((i-n)==0) c1 = invsqrt2;
                        if((i-n)>0)  c1 = 1;
                        temp += (float) (array2[x][i] * cosine[y-n][i-n] * c1/2);
                    }
```

```
                    t_array[y-n] = temp;
                }
                for(i=0; i<8; i++){
                    array2[x][n+i] = t_array[i];
                }
            }
        }
    }
/* IDCT on cols */
    for(n=0; n<W; n+=8) {
        for(m=0; m<H; m+=8){
            for(y=n; y<n+8; y++){
                for(x=m; x<m+8; x++){
                    temp = 0.0;
                    for(i=m; i<m+8; i++){
                        if((i-m)==0) c1 = invsqrt2;
                        if((i-m)>0)  c1 = 1;
                        temp += (float)(array2[i][y] * cosine[x-m][i-m] * c1/2);
                    }
                    array[x][y] = temp;
                }
            }
        }
    }
}

/* Rounds output image pixel values   */
void write_image_out()
 {
    int i,j;
    int temp;

    for(i=0;i<IMAGE_LEN;i++)
        for(j=0;j<IMAGE_LEN;j++) {
            temp =(int)(ROUND(array[i][j]));
            if(temp<0) temp=0;
            if(temp>255) temp=255;
            image_out[IMAGE_LEN*i+j] = (unsigned char) temp;
        }
}


/* main */
void main()
{
    int i,j;

    TARGET_INITIALIZE();            /* Target initialization for RTDX */

    read_image();
    dct() ;                         /*  DCT (1-d way)  */
    quantization();
    dequantization();
    idct();                         /* IDCT (1-d way) */
    write_image_out();
```

```
    RTDX_enableOutput(&ochan);        /* Enable the output channel,"ochan" */

    for (i=0; i<IMAGE_SIZE; i+=MAX_ELEMENTS) {
        for (j=i; j<(i+MAX_ELEMENTS); j++) {
            message[j-i] = (unsigned char) image_out[j];  /* write one row (256 elements) */
        }
  /* Send the data to the host  */
    if (!RTDX_write( &ochan, message, sizeof(message) ) ) {
        fprintf(stderr, "\nError: RTDX_write() failed!\n");
        abort();
    }
  }
  /* write a string to stdout */
    puts(" Completed Successfully!\n");
  }
```

Program 17. The C code of the jpeg.c

In the *jpeg.c*, the files*stdio.h*, *stdlib.h*, *math.h*, *rtdx.h* , *target.h*, *jpeg_main.h* and *scenary.h* are included. The *rtdx.h* contains the declarations of the functions related to the RTDX technology. The *target.h* includes the declaration of function TARGET_INITIALIZE(), which initializes the DSP and activates the interrupts so for the RTDX technology to be enabled. The *jpeg_main.h*, which is presented in Program 18, and contains declarations of some constants. The *scenary.h* contains the 256 x 256 image to be processed.

Before the declarations of the *main()* function, the RTDX channel "ochan" is defined that will transfer the process result from DSP to GUI. The channel "ochan" transfers from DSP an array of 256 numbers, that represents a line from the final image, to GUI.

In addition, before the *main()* function, the *read_image()*, *dct()*, *quantization()*, *dequantization()*, *idct()* and *write_image_out()* functions are described.

In the *main()* function the DSP is initialized and the RTDX Technology is enabled. By calling the *read_image()* function, the image is copied to a two-dimension. The direct DCT is applied on the image by calling the *dct()* function. The quantization stage is achieved by calling the *quantization()* function. In this point, the JPEG encoder has been completed. The *dequantization()* function, realizes the dequantization. The JPEG decoder is completed by calling the *idct()* function, with which the IDCT is applied on the image. The *write_image()* function converts the data of the reconstructed image to integers between [0 255] and and inputs them to the array image_out. The RTDX channel output "ochan" is enabled and the image is being transferred from the DSP to GUI. This transfer of the final image is carried out gradually meaning that each time only one line of the final image is written to the RTDX channel.

The *jpeg_main.h* is presented in Program 18, and contains the declarations of some useful constants such as IMAGE_LEN and BLOCK_LEN, with values of 256 and 8 respectively. Furthermore, the constants W and H, that correspond to the image's dimensions, as well as the constant invsqrt2, that equals to $1/\sqrt{2}$, are declared. Finally the *ROUND()* function is defined, which rounds a number to its nearest integer. This function is used for rounding the values during the quantization stage.

```
#ifndef JPEG_MAIN_H

#define IMAGE_LEN 256
#define IMAGE_SIZE (IMAGE_LEN*IMAGE_LEN)

#define W 256
#define H 256

#define BLOCK_LEN 8
#define BLOCK_SIZE (BLOCK_LEN*BLOCK_LEN)

#define invsqrt2  0.70710678

#define ROUND( a )      ( ( (a) < 0 ) ? (int) ( (a) - 0.5 ) : \
                                  (int) ( (a) + 0.5 ) )

#endif
```

Program 18. The jpeg_main.h

The memory allocation is completed in *rtdx_JPEG.cmd* that is presented in Program 19.

```
-c
-heap  0x1000
-stack 0x1000
-u __vectors
-u _auto_init

_HWI_Cache_Control = 0;
_RTDX_interrupt_mask = ~0x000001808;

MEMORY
{
  VECS:     org =          0h,  len =       0x220
  IRAM:     org = 0x00000220,  len = 0x0000FE00  /*internal memory*/
  SDRAM:    org = 0x80020000,  len = 0x00fdffff  /*external memory*/
  SDRAM0:   org = 0x80000000,  len = 0x00010000
  SDRAM1:   org = 0x80010000,  len = 0x00010000
  FLASH:    org = 0x90000000,  len = 0x00020000  *flash memory*/
}

SECTIONS
{
  .intvecs    >   0h
  .text       >   SDRAM
  .rtdx_text  >   SDRAM
  .far        >   SDRAM
  .stack      >   SDRAM
  .bss        >   SDRAM
  .cinit      >   SDRAM
  .pinit      >   IRAM
  .cio        >   SDRAM
  .const      >   SDRAM
```

```
    .data        >   SDRAM
    .rtdx_data   >   SDRAM
    .switch      >   SDRAM
    .sysmem      >   SDRAM

    myvar0       >   SDRAM0
    myvar1       >   SDRAM1
    myvar2       >   SDRAM
    myvar3       >   SDRAM
    msg_var      >   SDRAM
}
```

Program 19. The rtdx_JPEG.cmd

The View Window of the CCS project, named RTDX_JPEG.pjt, is presented in Figure 144, which implements the JPEG encoder and decoder RTDX_JPEG.pjt. Apart from the files *jpeg.c*, and *rtdx_JPEG.cmd* that were described above, it should also include the *intvecs.asm* that is developed by TI. The local optimization File (-o3) is selected for the *jpeg.c*. In addition, the libraries *rtdx.lib* (or rtdxsim.lib id a simulator is used) and *rts6701.lib* must also be added to the project.

Figure 148. The View Window of CCS for the RTDX_JPEG.pjt

4.2.4 Histogram Equalization

One of the most simple and effective technique for improving the quality of an image is the histogram equalization. The purpose of this procedure is to generate an improved image, the histogram of which will be uniformly. The histogram equalization has very good results on images with low contrast, meaning that most luminance values are limited in a small range between [0 255].

Histogram equalization on a grayscale image

Analytically, the result of the histogram equalization process is to expand its luminance distribution, in order to take the whole range between [0 255] (as long as a grayscale image is concerned) and not only a small part of it. In this case the image is visually.

The histogram equalization process, can be completed through these four steps:

(1) Histogram computation of original image
(2) Accumulative histogram computation
(3) Calculation of new, normalized luminance values
(4) Transform the initial image to the final image

Histogram equalization on a grayscale image

The simplest way of histogram equalization on a color image is to apply the histogram equalization algorithm separately, on each one of the three R G and B components. It is obvious that the image that will arise after the histogram equalization, in each component will have more intense color , and consequently will be more pleasant to human. However, this method is not supposed to be effective, because of its calculation complexity which is increased with the size of the image, and because of the contrast growth that leads to color contrast even in areas that has color homogeny.

It is commonly known, that before a color image is processed, many times is converted from the RGB space to another, such as HSI, YC_bC_r, LAB etc. These spaces have certain advantages over the RGB space, and this is the reason why many algorithms that deal with color pictures, suppose that the pictures are already converted to some other space. One of these spaces that is widely used (mainly in image encoding, as the JPEG encoding) is the YC_bC_r space. The advantage of the color representation of an image in this space is that the greatest portion of information is included in the Y component, which is called luminance. On the other hand, the C_b and C_r components are call chrominance, and they contain less information, compared to Y component. Something like that, is not really happening in the RGB space where each channel has the same i.

With the advantage of the YC_bC_r space, where the greatest portion of information is included in the Y component, the histogram equalization can be applied only on this component and produce the same and even better results compared to RGB space. The transformation from the RGB space to theYC_bC_r space, is giver from equation 3.

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ Cb &= -0.169R - 0.331G + 0.500B \\ Cr &= 0.500R - 0.419G - 0.081B \end{aligned} \tag{3}$$

Respectively the inverse transformation, ($YC_bC_r \rightarrow RGB$) is giver from equation 34:

$$\begin{aligned} R &= 1.000Y & & + 1.402Cr \\ G &= 1.000Y & - 0.344Cb & - 0.714Cr \\ B &= 1.000Y & + 1.772Cb & \end{aligned} \tag{4}$$

Histogram equalization implementation on a color image in RGB space

Below the histeq_RGB_rtdx.c that implements the histogram equalization of a color image in the RGB space, is presented.

```
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <rtdx.h>           /* RTDX_Data_Read */
#include <stdio.h>          /* printf */
#include "target.h"         /* TARGET_INITIALIZE */

#define IMAGE_SIZE 65536
#define H 256
#define W 256

#include "red.h"            /* R_1d[IMAGE_SIZE] data */
#include "green.h"          /* G_1d[IMAGE_SIZE] data */
#include "blue.h"           /* B_1d[IMAGE_SIZE] data */

#pragma DATA_SECTION (R_1d,"R1_var")
#pragma DATA_SECTION (G_1d,"G1_var")
#pragma DATA_SECTION (B_1d,"B1_var")

#pragma DATA_SECTION (R,"R_var")
int R[IMAGE_SIZE];

#pragma DATA_SECTION (RGB_out,"RGB_out_var")
unsigned char RGB_out[3*IMAGE_SIZE];

#pragma DATA_SECTION (im_1d,"im_var")
unsigned char im_1d[IMAGE_SIZE];

#pragma DATA_SECTION (message, "msg_var")

RTDX_CreateOutputChannel(ochan);        /* Channel to use to write data */

#define MAX_MESSAGES 758
#define MAX_ELEMENTS 256

unsigned char message[MAX_ELEMENTS]; /* 256 pixels each message */

int ch_count;

/* Histogram Equalization. */
```

```
void hist_equalization()
{
   int i;
   int hist[256];
   int sum_hist[256];
   int sum;

/* Clear hist[256] & sum_hist[256] */

   for (i=0; i<256; i++) {
         hist[i] = 0;
         sum_hist[i] = 0;
   }

  /* Calculate image histogram */

   for(i=0; i<IMAGE_SIZE; i++) {
         hist[(int) im_1d[i]]++;
   }

  /* Calculate normalized sum of hist */

   sum = 0;

   for(i=0; i<256; i++) {
         sum = sum + hist[i];
         sum_hist[i] = sum * 255 ;
   }

   for(i=0; i<256; i++) {
         sum_hist[i] = sum_hist[i] >> 16;
   }

   /* Transform Image using sum_hist as a LUT */

   for(i=0; i<IMAGE_SIZE; i++) {
         R[i]= sum_hist[(unsigned char)im_1d[i]];
   }
}

/*  Writes output image              */
void write_image()
{
   int i;

   for (i=0; i<IMAGE_SIZE; i++) {
         if (R[i]<0) R[i]=0;
               if (R[i]>255) R[i]=255;
                     RGB_out[i + ch_count*IMAGE_SIZE] = (unsigned char) R[i];
   }
}

/* main program */

void main ()
```

```
{
   int i, j;
   ch_count=0;

/* Equalize RED channel */

   for (i=0; i<IMAGE_SIZE; i++) { im_1d[i] = R_1d[i]; }

   hist_equalization();
   ch_count=0;
   puts("\n histeq RED ok!");
   write_image();

/* Equalize GREEN channel */

   for (i=0; i<IMAGE_SIZE; i++) { im_1d[i] = G_1d[i]; }

   hist_equalization();
   ch_count=1;
   puts("\n histeq GREEN ok!");
   write_image();

/* Equalize BLUE channel */

   for (i=0; i<IMAGE_SIZE; i++) { im_1d[i] = B_1d[i]; }

   hist_equalization();
   ch_count=2;
   puts("\n histeq BLUE ok!");
   write_image();

/* Send RED GREEN BLUE data */

   TARGET_INITIALIZE();            /* Target initialization for RTDX */

   RTDX_enableOutput(&ochan);

   for (i=0; i<(3*IMAGE_SIZE); i+=MAX_ELEMENTS) {
        for (j=i; j<(i+MAX_ELEMENTS); j++) {

             message[j-i] = (unsigned char) RGB_out[j];
        }
/* Send the data to the host */

        if (!RTDX_write( &ochan, message, sizeof(message) ) ) {
             fprintf(stderr, "\nError: RTDX_write() failed!\n");
             abort();
        }
   }

   puts("\nProgram Completed!");
}
```

Program 20. The C code of the histeq_RGB_rtdx.c

In the C code of the *histeq_RGB_rtdx.c* that is presented in Program 20, the files *stdlib.h*, *string.h*, *math.h*, *rtdx.h*, *stdio.h*, *target.h*, *red.h*, *green.h* and *blue.h*, are included. The file *rtdx.h* contains the declarations of the functions related to the RTDX technology. The file *target.h* includes the declaration of function TARGET_INITIALIZE(), which initializes the DSP and activates the interrupts so for the RTDX technology to be enabled. The header files *red.h green.h* and *blue.h* contain the R, G and B component of the color image that the histogram equalization will be applied on. The image to be processed has 256 x 256 pixels.

Before the declaration of the *main()* function the RTDC channel "ochan" is defined, that will transfer the process result from DSP to GUI. The channel "ochan" transfers from DSP an array of 256 numbers, that represents a line from the final image, to GUI.

In addition, before the *main()* function the *hist_equalization()* and *write_image()* functions are defined.

In the *main()* function the DSP is initialized and the RTDX Technology is enabled. For each component the functions *hist_equalization()* and *write_image()* are called. By calling the *hist_equalization()* function the histogram equalization is applied on each component. By calling the *write_image()* the content of each equalized component is converted into integers between [0 255] and they are placed into the 1D-array RGBout. When the equalization process of all three components has been completed successfully the array RGBout with 256x256x3 = 196608 elements contains all three components of the equalized image. The first 65536 elements correspond to R component, the next 65536 to G component and the last 65536 elements to B component. Afterwards, the RTDX channel output "ochan" is enabled and the image is transferred from DSP to GUI. This transfer of the final image is carried out gradually meaning that each time only one line of the final image is written to the RTDX channel.

The next necessary project file is *rtdx_histeqRGB.cmd* and is presented in Program 21. In this file, three memory areas of the DSK are defined, in which a name is given (i.e. BMEM), while in the SECTIONS field every variable that is declared in the C code is connected with the #pragma directive in one of these areas.

```
-c
-heap  0x1000
-stack 0x1000
-u __vectors
-u _auto_init

_HWI_Cache_Control = 0;
_RTDX_interrupt_mask = ~0x000001808;

MEMORY
{
   VECS:     o=00000000h   l=00000200h   /* interrupt vectors */
   PMEM:     o=00000200h   l=0000FE00h   /* Internal RAM (L2) mem */
   BMEM:     o=80000000h   l=01000000h   /* CE0, SDRAM, 16 MBytes */

}

SECTIONS
{
   .intvecs        >   0h
   .text           >   BMEM
   .rtdx_text      >   BMEM
```

```
    .far          >  BMEM
    .stack        >  BMEM
    .bss          >  BMEM
    .cinit        >  BMEM
    .pinit        >  PMEM
    .cio          >  BMEM
    .const        >  BMEM
    .data         >  BMEM
    .rtdx_data    >  BMEM
    .switch       >  BMEM
    .sysmem       >  BMEM

    msg_var       >  BMEM
    im_var        >  BMEM
    R1_var        >  BMEM
    G1_var        >  BMEM
    B1_var        >  BMEM
    R_var         >  BMEM
    RGB_out_var   >  BMEM
}
```

Program 21. The rtdx_histeqRGB.cmd

In Figure 149, the View Window of the CCS project RTDX_histeqRGB.pjt, is presented. This project implements the histograms equalization of an image in the RGB space. Apart from the files *histeq_RGB_rtdx.c*, and *rtdx_histeqRGB.cmd* that were described above, the intvecs.asm, that was developed by TI, must also be added. For the *histeq_RGB_rtdx.c* the local optimization File (-o3) is selected. In addition in the project the libraries *rtdx.lib* (or rtdxsim.lib if a simulator is used) and *rts6701.lib* should be added as well.

Figure 149. The View Window of CCS for the RTDX_histeqRGB.pjt

Histogram equalization implementation on a color image in YCbCr space

The C code of the histeq_Y_rtdx.c that implements the histpgram equalization of a color image in the YCbCr space, is presented below.

```
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <rtdx.h>               /* RTDX_Data_Read */
#include <stdio.h>              /* printf */
#include "target.h"             /* TARGET_INITIALIZE.*/

#define IMAGE_SIZE 65536
#define H 256
#define W 256

#include "red.h"                /* R_1d[IMAGE_SIZE] data */
#include "green.h"              /* G_1d[IMAGE_SIZE] data */
#include "blue.h"               /* B_1d[IMAGE_SIZE] data */

#pragma DATA_SECTION (R_1d,"R1_var")
#pragma DATA_SECTION (G_1d,"G1_var")
#pragma DATA_SECTION (B_1d,"B1_var")
#pragma DATA_SECTION (Y,  "Y_var")
#pragma DATA_SECTION (Cb,"Cb_var")
#pragma DATA_SECTION (Cr,"Cr_var")
#pragma DATA_SECTION (R, "R_var")
#pragma DATA_SECTION (G, "G_var")
#pragma DATA_SECTION (B, "B_var")
#pragma DATA_SECTION (RGB_out,"RGB_var")

unsigned char RGB_out[3*IMAGE_SIZE];
int  Y[IMAGE_SIZE];
int Cb[IMAGE_SIZE];
int Cr[IMAGE_SIZE];
int R[IMAGE_SIZE];
int G[IMAGE_SIZE];
int B[IMAGE_SIZE];

#pragma DATA_SECTION (message, "msg_var")

RTDX_CreateOutputChannel(ochan);  /* Channel to use to write data */

#define MAX_MESSAGES 256
#define MAX_ELEMENTS 256

unsigned char message[MAX_ELEMENTS]; /* 256 pixels each message */

/* RGB to YCbCr color conversion */
void RGB_to_YCbCr()
{
  int i;
```

```
    for(i=0; i<IMAGE_SIZE; i++) {

    /* weights ->(weights *1000) */
        Y[i] =  299*R_1d[i] + 587*G_1d[i] + 114*B_1d[i];
        Cb[i] = -169*R_1d[i] - 331*G_1d[i] + 500*B_1d[i];
        Cr[i] =  500*R_1d[i] - 419*G_1d[i] -  81*B_1d[i];
    }

    /* devide with 1024=2^10 */
    for(i=0; i<IMAGE_SIZE; i++) {
          Y[i] = (int) (Y[i]/1000);
          Cb[i] = (int) (Cb[i]/1000);
          Cr[i] = (int) (Cr[i]/1000);
    }
}

/* Histogram Equalization. */
void hist_equalization()
{
    int i;
    int hist[256];
    int sum_hist[256];
    int sum;

/* Clear hist[256] & sum_hist[256] */

    for (i=0; i<256; i++) {
          hist[i] = 0;
          sum_hist[i] = 0;
    }

/* Calculate image histogram */

    for(i=0; i<IMAGE_SIZE; i++) {
          hist[(int) Y[i]]++;
    }

/* Calculate normalized sum of hist */

    sum = 0;

    for(i=0; i<256; i++) {
          sum = sum + hist[i];
          sum_hist[i] = sum * 255 ;
    }

    for(i=0; i<256; i++) {
          sum_hist[i] = sum_hist[i] >> 16;
    }

/* Transform Image using sum_hist as a LUT */

    for(i=0; i<IMAGE_SIZE; i++) {
          Y[i]= sum_hist[(unsigned char)Y[i]];
```

```
    }
  }

  /* YCbCr to RGB color conversion */
  void YCbCr_to_RGB()
  {
    int i;

    for(i=0; i<IMAGE_SIZE; i++) {

    R[i] = 10000*Y[i] -     9*Cb[i] + 14017*Cr[i];
    G[i] = 10000*Y[i] -  3437*Cb[i] -  7142*Cr[i];
    B[i] = 10000*Y[i] + 17722*Cb[i] + 10*Cr[i];
     }

    for(i=0; i<IMAGE_SIZE; i++) {
          R[i] = (int) (R[i]/10000);
          G[i] = (int) (G[i]/10000);
          B[i] = (int) (B[i]/10000);
    }
  }


  /*  Writes output image              */
  void write_image()
  {
    int i;

     for (i=0; i<IMAGE_SIZE; i++) {
          if (R[i]<0) R[i]=0;
          if (G[i]<0) G[i]=0;
          if (B[i]<0) B[i]=0;
          if (R[i]>255) R[i]=255;
          if (G[i]>255) G[i]=255;
          if (B[i]>255) B[i]=255;

          RGB_out[i] = (unsigned char) R[i];
          RGB_out[i+IMAGE_SIZE] = (unsigned char) G[i];
          RGB_out[i+2*IMAGE_SIZE] = (unsigned char) B[i];
    }
  }

  /* main program */
  void main ()
  {
    int i, j;

    RGB_to_YCbCr();
    hist_equalization();

    puts("\n Equalization ok!");

    YCbCr_to_RGB();
    write_image();
```

```
/* Send RED GREEN BLUE data */

   TARGET_INITIALIZE();   /* Target initialization for RTDX */

   RTDX_enableOutput(&ochan);

   for (i=0; i<(3*IMAGE_SIZE); i+=MAX_ELEMENTS) {
        for (j=i; j<(i+MAX_ELEMENTS); j++) {
             message[j-i] = (unsigned char) RGB_out[j];
        }

/* Send the data to the host */
        if (!RTDX_write( &ochan, message, sizeof(message) ) ) {
             fprintf(stderr, "\nError: RTDX_write() failed!\n");
             abort();
        }
   }

   puts("\nProgram Completed!");
}
```

Program 22. The C code of the histeq_Y_rtdx.c

In the C code of the *histeq_Y_rtdx.c* that is resented in Program 22, the files *stdlib.h*, *string.h*, *math.h*, *rtdx.h*, *stdio.h*, *target.h*, *red.h*, *green.h* and *blue.h* are included. The *rtdx.h* contains the declarations of the functions related to the RTDX technology. The header file *target.h* includes the declaration of function TARGET_INITIALIZE(), which initializes the DSP and activates the interrupts so for the RTDX technology to be enabled. The header files *red.h green.h* and *blue.h* contain the R, G and B component of the color image that the histogram equalization will be applied on. The image to be processed has 256 x 256 pixels.

Before the declaration of the *main()* function the RTDC channel "ochan" is defined, that will transfer the process result from DSP to GUI. The channel "ochan" transfers from DSP an array of 256 numbers, that represents a line from the final image, to GUI.

In addition, before the *main()* the functions *RGB_to_YCbCR()*, *hist_equalization()*, *YCbCr_to_RGB()* and *write_image()* are described.

In the *main()* function, the image is convertred from the RGB to the YCbCr space using the function *RGB_to_YCbCR()*. With the *hist_equalization()* function the equalization of the Y component (luminance) is achieved. Then, by calling *YCbCr_to_RGB()* function the equalized image is coverted to the RGB space. By calling the *write_image()* the content of each equalized component is converted into integers between [0 255] and they are placed into the 1D-array RGBout. When the equalization process of all three components has been completed successfully the array RGBout with 256x256x3 = 196608 elements contains all three components of the equalized image. The first 65536 elements correspond to R component, the next 65536 to G component and the last 65536 elements to B component. Afterwards, the RTDX channel output "ochan" is enabled and the image is transferred from DSP to GUI. This transfer of the final image is carried out gradually meaning that each time only one line of the final image is written to the RTDX channel.

The next necessary project file is *rtdx_histeqY.cmd* and is presented in Program 21. In this file, three memory areas of the DSK are defined, in which a name is given (f.e. BMEM), while in the SECTIONS field every variable that is declared in the C code is connected with the #pragma directive in one of these areas.

```
-c
-heap  0x1000
-stack 0x1000
-u __vectors
-u _auto_init

_HWI_Cache_Control = 0;
_RTDX_interrupt_mask = ~0x000001808;

MEMORY
{
  VECS:   o=00000000h   l=00000200h     /* interrupt vectors       */
  PMEM:   o=00000200h   l=0000FE00h     /* Internal RAM (L2) mem   */
  BMEM:   o=80000000h   l=01000000h     /* CE0, SDRAM, 16 MBytes   */
}

SECTIONS
{
  .intvecs    >   0h
  .text       >   BMEM
  .rtdx_text  >   BMEM
  .far        >   BMEM
  .stack      >   BMEM
  .bss        >   BMEM
  .cinit      >   BMEM
  .pinit      >   PMEM
  .cio        >   BMEM
  .const      >   BMEM
  .data       >   BMEM
  .rtdx_data  >   BMEM
  .switch     >   BMEM
  .sysmem     >   BMEM
  msg_var     >   BMEM
  im_var      >   BMEM
  R1_var      >   BMEM
  G1_var      >   BMEM
  B1_var      >   BMEM
  Y_var       >   BMEM
  Cb_var      >   BMEM
  Cr_var      >   BMEM
  R_var       >   BMEM
  G_var       >   BMEM
  B_var       >   BMEM
  RGB_var     >   BMEM
}
```

Program 22. The rtdx_histeqY.cmd

In Figure 150, the View Window of the CCS project RTDX_histeqY is presented. This project implements the histograms equalization of an image in the YCbCr space. Apart from the files *histeq_Y_rtdx.c*, and *rtdx_histeqY.cmd* that were described above, the intvecs.asm, which was developed by TI, must also be added. For the *histeq_Y_rtdx.c* the local

optimization File (-o3) is selected. In addition in the project the libraries *rtdx.lib* (or rtdxsim.lib if a simulator is used) and *rts6701.lib* should be added as well.

Figure 150. The View Window of CCS for RTDX_histeqY.pjt

4.2.5 Implementing the VI to control the application

The VI that will be presented below is ImageGUI.v. It controls and communicate with CCS and consequently the DSP of DSKC6713. With ImageGUI.vi the user can choose the image that will be loaded to the DSP as well as the processing algorithm that will be applied on this image. This VI shows the selected image before and after the processing.

The ImageGUI.vi front panel

The front panel of the ImageGui.vi that is presented in Figure 151 provides to indicators named “Image_in” and Image_Out” that represent te image to be loaded and processed to DSP and the processing result image. The control “Image Type” provides two radio buttons, the “Grayscale” and the “Color (RGB)” with which the user can define the type of the image that will be loaded to DSP. With the control “Algorithm Selection”, that has the sates shown in Table 13, the user can select the image to be processed. As Table 13 shows, some states of the control “Algorithm Selection” correspond to color images and some others to grayscale images. Therefore, the states of this control that do not correspond to the value of the control “Image Type” are disabled and presented with a light grey color for user’s convenience. In front panel of Figure 151 the active states of the control “Algorithm Selection” are shown, when from the control “Image Type” has been selected that the image to be used is actually a grayscale image.

Value	State	Description
0	Algorithm Selection	Nothing happen
1	Sobel Edge Detection (Gray)	The edges of a grayscale image are detected .
2	Histogram Equalization RGB (Color)	The histogram of a color image in the RGB space is equalized.
3	Histogram Equalization Y (Color)	The histogram of a color image in the YCbCr space is equalized.
4	DCT/IDCT slow (Gray)	The direct DCT and the IDCT is applied on a grayscale image by using the 1D-DCT .
5	DCT/IDCT fast (Gray)	The direct DCT and the IDCT is applied on a grayscale image by using the McGovern algorithm.
6	JPEG encoder/decoder (Gray)	A grayscale image is encoded and decoded according to the JPEG standard.

Table 13. The states of the control input "Algorithm Selection"

Figure 151. The front panel of ImageCUI.vi

When the control "Algorithm Selection" is in the state of Algorithm Selection, then the VI is in standby mode. If in this moment the user desires to terminate its function, the "Stop" button will be pressed. I any state of the control "Algorithm Selection" is selected except for the Algorithm Selection then the VI will exit the standby mode and a dialog window will appear automatically, being the Windows explorer, asking to select and load the desired image. After that the image will appear in the indicator "Image in". When the DSP successfully completes the process define by the control "Algorithm Selection" the final, processed image will appear in the "Image Out".

The block diagram panel of the ImageGUI.vi

In Figure 152, the block diagram of ImageGUI.vi is shown, which consists of six phases that will be described thoroughly below, so the function of this VI to be understood.

Figure 152. The block diagram of ImageGUI.vi

The first phase of the block diagram is shown in Figure 153 and it implements the standby mode of the VI, since it runs constantly till the button "Stop" is pushed or the state of the control "Algorithm Selection" is changed. Firstly the Not Equal To 0?.vi is selected if the value of the control "Algorithm Selection" is different than 0 (which means that the state Algorithm Selection is not selected). If the output of the Not Equal To 0?.vi is False then the value of the control "Algorithm Selection" is 0. The "Stop" button and the output of Not Equal To 0?.vi are connected as inputs on the Compound Arithmetic.vi (that is defined to execute the OR operation). Therefore, the output of Compound Arithmetic.vi that is connected to the control input of the While structure, terminates the execution of the first phase of the block diagram when the "Stop" button is pushed or the value of the control "Algorithm Selection" is changed (different than 0).

(a) **(b)**

Figure 153. The first phase of the block diagram of ImageGUI.vi
(a) for a color image **(b)** foe a grayscale image

In the first phase of the block diagram, the states of the control "Algorithm Selection" that are not correspond to the value of the control "Image Type", are disabled. The deactivation of the control "Algorithm Selection" states is achieved with the DisabledItems[] property. This property takes as input an array that shows the values of the states to be disabled. Through the Case structure the input of the DisabledItems[] property is altered according to the value of the control "Image Type" that is connected to the control input of the Case structure. Therefore, when a color image is selected by the control "Image Type", the states of the control "Algorithm Selection" with values 1, 4, 5 and 6, will be disabled. In the

same way when the user selects a grayscale image, the states of the control "Algorithm Selection" with values 2, and 3, will be disabled.

The phases 2, 3, 4, 5, and 6 are the False "case" of a Case structure that is controlled by the "Stop" button. The True "case" of this structure is empty, since when the "Stop" button is pushed all other phases are bypassed and the function of the VI is terminated.

In Figure 154, the second phase of the ImageGUI.vi has been completed, in which the path of the CCS project to be loaded, is defined. In this phase the reading process, the data collection and the representation of the image selected by the user, through the respective dialog window, are realized.

Figure 154. The second phase of the block diagram of ImageGeUI.vi

The ImageGUI.vi should be saved in the directory that contains the folders with the CCS projects' files that execute the respective process on the image. The output of the Current VI's Path that indicates the path where the VI is saved, is connected to the input of Strip Path.vi that cuts off the last part of the path. If the path of the VI is C:\dsp_projects\ image_projects\ImageGUI.vi then the output of the Strip Path.vi is C:\dsp_projects\ image_projects. The output of Strip Path.vi is connected to input "base path" of Build Path.vi while in input " name or relative path" the output of Case structure. The Case structure alters the input " name or relative path" of Build Path.vi according to the CCS project that has to be loaded according to the state of the control "Algorithm Selection. The cases of the Case structure are presented in Figure 155. Therefore, if the state of the control "Algorithm Selection" is Sobel Edge Detection (Gray) then the input " name or relative path" of the Build Path.vi will take the RTDX_Sobel_edges\ RTDX_Sobel_edges.pjt.so the output of the Build Path will be the CCS project path for the edge detection. In this way the path of the CCS project that will implement the processing algorithm indicated by the control "Algorithm Selection", is created

Figure 155. The cases of the Case structure ijn the second phase of the block diagram of ImageGUI.vi

In the second phase of the ImageGUI.vi the Read BMP and get data.vi is executed. It reads a BMP image, shows it and collects its data in order to create three 1-D arrays that will represent the three components of a color image. In the case of a grayscale image then the three arrays that will be created by the Read BMP and get data.vi will contain the luminance values of this specific image. At the ImageGUI.vi, in input "Original Image" of the Read BMP and get data.vi a reference to the control "Image in" is connected while in input "Image Type" the control "Image Type" is connected. In outputs "Image Data 1", "Image Data 2" and Image Data 3" contain the data of the read image. In output "Original Image 2" the indicator "Image in" is connected in order to represent the image just read. The output "cancelled" indicates that whether the "Cancel" button is pushed in the dialog window that pops up for the image selection.

In Figure 156 the block diagram of the Read BMP and get data.vi, is shown where the input "Path" is checked foe being empty or not containing any paths. If it is empty, like the case where the Read BMP and get data.vi is called by the ImageGUI.vi, the True case of the structure "Case Structure 1" will be executed. In this point the File Dialog.vi is executed that will show the dialog window for the image selection. The output "path" of File Dialog.vi will include the path of the chosen image. The output "cancelled" indicates that whether the "Cancel" button is pushed in the dialog window that pops up for the image selection and is connected to the output "cancelled" of the Read BMP and get data.vi and with the control case of the structure "Case Structure 2". If in the dialog window the "Cancel" button is pushed by the user then the True case of the structure "Case Structure 2" will be selected , which is empty and the function of the Read BMP and get data.vi will be terminated. In the False case of the structure "Case Structure 2" the output "path" of File Dialog.vi will be connected to the input "Path to BMP File" of Read BMP File.vi which is going to read the BMP image. The output cluster "Image data" of the Read BMP File.vi is connected to the corresponding input of the Draw Flattened Pixmap.vi. The Draw Flattened Pixmap.vi will represent the image through the indicator "Original Image 2".

In the control case of the structure "Case Structure 3" the input "Image Type" is connected. Therefore, the case "Color(RGB)" of the structure "Case Structure 3" is executed when a color image is used. In this point, the output "image" of the output cluster "Image data" of the Read BMP File.vi is an array with 196608 elements (256x256x3 = 196608 for a 256x256 image) and contains the values f the R, G and B components of the image. In this array, the first three elements correspond to the R, G and B components of the first pixel, the

next three elements correspond to the three components of the second pixel etc. In the case of "Color(RGB)" of the structure "Case Structure 3" the separation of the three components of the image must be done. For this, a For loop is used that is repeated 65536 times (for a 256 x 256 image) and takes as input the output "image" of the output cluster "Image data" in the Read BMP File.vi. With the help of the Index.vi and the pattern presented in the block diagram of the Read BMP and get data.vi this separation of the three image components is achieved. The R, G and B components after the separation will be three 1-D arrays consisting of 65536 elements (for a 256x256 image) that are included in the outputs "Image Data 1", "Image Data 2" and "Image Data 3".

Figure 156. The block diagram of the Read BMP and get data.vi

In case where a grayscale image is used, the case "Grayscale" of the structure "Case Structure 3" is executed in which the output "image" of the output cluster "Image data" of the Read BMP File.vi, which is an array with 65536 elements (256x256 = 65536 for a 256x256 image) andcontains the luminance value of each image pixel, is connected to the outputs "Image Data 1", "Image Data 2" and "Image Data 3" of Read BMP and get data.vi.

The input "Original Image" of the Read BMP and get data.vi that contains reference to the indicator "Image in" defines the element, which the "DrawAreaSize" property is addressed to. The "DrawAreaSize" property addresses only to images controls and indicators and modifies their size according to the value of the cluster connected o this property. In this specific case, the outputs "Rectangle.right" and "Rectangle.left" are grouped (with the Bundle.vi) and connect to the "DrawAreaSize" property. In this way is assured that the image indicator (specifically the control "Image in") will be of the same size as the image that will represent.

The second phase of the block diagram of ImageGUI.vi is completed with the indicator "Image Out" to be cleared. This is achieved by connecting the constant "Empty Picture" to the local variable of the indicator "Image Out".

The phases 3, 4, 5, and 6 comprise the False case of a Case structure that I scontrolled by the output "cancelled" of the Read BMP and get data.vi. In the True case of this structure that is executed only when the "Cancel" button, in the dialog window for the image selection is pressed, the reset of the control "Algorithm Selection" to the state Algorithm Selection is

carried out. The reset of the control “Algorithm Selection” is achieved by connection the constant “Algorithm Selection” to the local variable of this specific control.

As long as the “Cancel” button, in the dialog window for the image selection, is not pressed and the second phase of the block diagram of ImageGUI.vi is completed, the False case of the above structure that includes the phases 3, 4, 5 and 6, will be executed.

In the third phase of the block diagram of ImageGUI.vi that is presented in Figure 157, the reset of the control “Algorithm Selection” to the state Algorithm Selection is carried out and the creation of three header files (Red.h, Green.h and Blue.h) that contain the three components of a color image or the creation of a single header file (scenary.h) that contain the luminance values of a grayscale image according to the value of control “Image Type” is taking place.

(a) **(b)**

Figure 157. The third phase of the block diagram of ImageGUI.vi
(a) for a color image **(b)** for a grayscale image

The reset of the control “Algorithm Selection” is achieved by connecting the constant “Algorithm Selection” to the local variable of this specific control.

When he use of a color image is selected, then the case “Color(RGB) of the Case structure in the third phase of the block diagram, is executed. In this case three header files must be created the Red.h, Green.h and Blue.h, which they contain the three arrays R_1d, G_1d and B_1d respectively. These are unsigned short arrays and they include the components R, G and B of the image. These files should be created in order to load the image to be processed on the DSP during the building process of the respective CCS project. The files Red.h, Green.h and Blue.h must be saved in the same directory as the respective project.

The path where the header files Red.h, Green.h and Blue.h will be saved is defined by the CCS project path in the first phase. The path of the CCS project acts as an input to the Strip Path.vi and the last part of it (name of the project) is cut off. The output of Strip Path.vi is connected to the input “base path” of the Build Path.vi while to the input “name or relative path” the string constant is connected, which contains the name of the file to be created. In this way the path of the Red.h, Green.h and Blue.h is defined. The Write data to file.vi, that creates and saves these files, takes as input the path where the respective file is saved, the name of the array that will contain this file and array’s data. In order to create the header file Red.h, to input “File Path” of Write to data file.vi the output of Build Path.vi that creates the path of this file, is connected. To input “Array Name” of Write to data file.vi the string

constant, that contains the name of the array such as R_1d, is connected, while the input "Data" is connected to output "Image Data 1" of the Read BMP and get data.vi, which was described in the second phase, and contains the R component of the image. Th esam eprocess is followed for the creation of the Green.h and Blue.h header files.

When the use of a grayscale image is selected, then the case "Grayscale" of the Case structure is executed. In this case, following the process that creates the Red.h, the heder file scenary.h is created, which contains the array image_in. This is a unsigned short array and contains the value of luminance for each pixel of the image. This file must be created in order to load the image to be processed on the DSP during the building process of the respective CCS project. The file scenary.h is saved in the same directory as the respective project.

Figure 158. The block diagram of Write data to file.vi

The Write data to file.vi is used to create the Red.h, Green.h, Blue.h and scenary.h , the block diagram of which is presented in Figure 158. The input "File Path" contains the path and the name of the file to be created. In the folder where the project is, a file with name like the one indicated by the input "File Path" may already exist, so this is why the input "File Path" is connected to Delete.vi in order to delete this file. The output "dup path" of Delete.vi contains exactly the same path and name of the file indicated by the input "File Path" and is connected to input "file path" of Write Characters to File.vi in order to create a new file in the same path with the same name. The input "Data" contains the image data that will be saved to the new file. The input "Data" is connected to input "array" of the Array To Spreadsheet String.vi in order for the array to be converted to a string, where an element will be separated by the other with a comma ",". The input "Array Name" contains the name of the array that will be declared to the file that is created by the Write data to file.vi. The content of this file is a string generated by the Concatenate Strings.vi that takes as inputs three string constants, the input "Array Name" and the output of Array To Spreadsheet String.vi. If the input "Array Name" contains the name R_1d and the input "Data" contains the elements 1, 2, 3, and 4 then the output of Concatenate Strings.vi will be :

```
unsigned char R_1d[IMAGE_SIZE] = {
1 , 2 , 3 , 4
};
```

The output of Concatenate Strings.vi is connected to the input "character string" of Write Characters to File.vi in order for the data to be written to the new file indicated by the input "File Path" of Write data to file.vi.

The third phase of block diagram of ImageGUI.vi is completed with the presentation and description of the function of Write data to file.vi .

In the fourth phase of the block diagram of ImageGUI.vi that is presented in Figure 159, CCS is setting up to use the DSKC6713 and to be controlled in order to initiate the image processing. This phase is completed when the DSP finishes the process of the image.

Figure 159. The forth phase of the block diagram of ImageGUI.vi

The CCS_ Setup_Open.vi loads the CCStudio Setup.The CCS_Setup_ Clear.vi all previous setting of the CCStudio Setup. The CCS_Setup_Add_ Board.vi loads to CCStudio Setup the DSKC6713 drivers according to the path indicated by the constant "Path", since it is connected to the input "Driver Path" of subVI. The CCS_Setup_Save.vi saves all selected settings and the CCS_ Setup_Close.vi closes the CCStudio Setup.vi. In this point the CCS has been set for the use of the DSKC6713.

The CCS_Open.vi loads the CCS and the CCS_Open_Project.vi load the CCS project according to the path indicated by the input "Project Path In". The path that the input "Project Path In" of the CCS_Open_Project.vi contains, was defined in the second phase of the block diagram of ImageGUI.vi and converted to a string by using the Path To String.vi. The CCS_Build.vi commands the CCS to build the loaded project in order to generate the executable fle. When the building process is completed, the CCS_Build_Result.vi outputs the building result to the indicator "Build Result". The CCS_Connect.vi commands the CCS to connect to the DSKC6713, while the CCS_DSP_Reset.vi resets the board. As long as no errors have occurred during the building process the CCS_Download.vi commands the CCS to download the executable to the DSP that also contains the mage to be processed. The project uses the RTDX technology, so the CCS_RTDX_Enable.vi is used that sets the RTDX parameters and enables the RTDX technology. In this specific case, the default settings of the RTDX technology are used. The CCS_Run.vi, commands the DSP to initiate the execution of the program. Inside the While structure the Is_DSP_Running.vi has been placed that controls if the DSP is executing the program. The output "Running" of Is_DSP_Running.vi take the False value when the DSP stops the execution of the program which means that the process is complete. By using the Wait.vi that is connected to an arithmetic constant 200, the execution of the While structure every 200 msec, is achieved. The While structure stops to execute when the output "Running" of Is_DSP_Running.vi takes the False value. This is when the furth phase of the block diagram of ImageGUI.vi is completed.

In the fifth phase of the block diagram of ImageGUI.vi the image is being read, which is the DSP process result, through the RTDX channel "ochan". The fifth phase is actually consisting of a Case structure that is controlled by the control "Image Type".

When the use of a color image is selected the case Color(RGB) of the Case structure is executed, that is presented in Figure 160.

Figure 160. The fifth phase of the block diagram of ImageGUI.vi for a color image

The transfer of a color image is executed by gradually writing data to the respective RTDX channel. In fact, an array with 256 elements is written each time to the RTDX channel, so in order to complete the transfer of a color image the DSP has made 256 x 3 = 768 writes. Therefore, the RTDX channel contains 768 messages when the process of a color image is completed. In order to retrieve this color image 768 readings has to be made from the RTDX channel "ochan". This is achieved with a For loop that contains the RTDX_Read.vi, in which the RTDX_Read_SA_UI1 is selected because the content of the RTDX channel is arrays with 1-byte unsigned integers. To the input "Channel" of RTDX_Read.vi a string constant is connected that contains the name of the RTDX channel from which the reading of the image is going to be made. The output of the For loop is a 768x256 array that with the Reshape Array.vi is converted to an 1-D array with 768x256=196608 elements. The first 65536 elements of the array comprise the R component of the image, the next 65536 elements comprise the G component and the final 65536 elements the B component. The separation of the image's components is achieved by the use of the Array Subset.vi and with the pattern described in Figure 160. The outputs of the Array Subset.vi is the outputs of the Case structure as well and contain the components of the image retreived.

When the use of a grayscale image is selecte, the case Grayscale of Case structure is executed, that is presented in Figure 161.

Figure 161. The fifth phase of the ImageGUI.vi block diagram for a grayscale image

The transfer of a color image is executed by gradually writing data to the respective RTDX channel. In fact, an array with 256 elements is written each time to the RTDX channel, so in order to complete the transfer of a color image the DSP has made 256 x 3 = 768 writes. Therefore, the RTDX channel contains 768 messages when the process of a color image is completed. In order to retrieve this color image 768 readings has to be made from the RTDX channel "ochan". This is achieved with a For loop that contains the RTDX_Read.vi, in which the RTDX_Read_SA_UI1 is selected because the content of the RTDX channel is arrays with 1-byte unsigned integers. To the input "Channel" of RTDX_Read.vi a string constant is connected that contains the name of the RTDX channel from which the reading of the image is going to be made. The output of the For loop is a 768x256 array that with the Reshape Array.vi is converted to an 1-D array with 768x256=196608 elements that contain the luminance value of each pixel in the retrieved image. The outputs of the Array Subset.vi is connected to the three data outputs of the Case structure. In this point the fifth phase of the block diagram of ImageGUI.vi is completed.

In the sixth phase of the block diagram of ImageGUI.vi that is presented in Figure 162 , the termination of the CCS and the representation of the retrieved image is realized.

Figure 162. The sixth phase of the block diagram of ImageGUI.vi

The CCS_DSP_Halt.vi commands the DSP through CCS to stop the termination of the program. The use of CCS_DSP_Halt.vi in this specific case is since if no error has occurred or if no relative command has been given manually, the DSP will have already terminate the program's execution. The CCS_RTDX_Disable.vi disables the RTDX technology. The CCS_Close_Project.vi closes the project, opened by CCS and the CCS_Close.vi closes the CCS

With RGB2picture.vi the retrieved image from the DSP is represented to control "Image Out". The inputs "Red Channel", "Green Channel" and "Blue Channel" of the RGB2picture.vi are connected to the respective data outputs of Case structure in the fifth phase of the block diagram. To input "reference" a reference to the indicator "Image Out" is connected, while to the input cluster "Draw Area Size" is connected a constant cluster that defines the image size which in this case is 256x256. The indicator "Image Out" that represents the retrieved image is connected to the output "Image Out" of RGB2picture.vi.

Figure 163. The block diagram of RGB2picture.vi

In Figure 163 the block diagram of the RGB2picture.vi is presented where the inputs "Red Channel", "Green Channel" and "Blue Channel" contain the R, G and B components of a color image as 1-D arrays with 1-byte unsigned integers. By using the Join Numbers.vi the arrays that are included to inputs "Red Channel", "Green Channel" and "Blue Channel" are converted to a single array that has the same number of elements with them. The new array is connected to output "Flatten RGB data" and its elements are 4-bytes integers. The most significant byte of each element in this array has a zero value, the second byte of each element in this array has the value of the respective element in the R component, the third byte of each element in this array has the value of the respective element in the G component and the fourth byte of each element in this array has the value of the respective element in the B component. By using the Reshape Array.vi the above array is converted to an array with dimension defined by the input "Draw Area Size" meaning the dimensions of the image, which in this case are 256x256. The output of the Reshape Array is connected to the output "Unflatten RGB data" and to the input "Data (RGB format)" of Draw Unflattened Pixmap.vi. The Draw Unflattened Pixmap.vi is a polymorphic VI in which the Draw True-Color Pixmap is selected and to the output "new picture" the image, the data of which are included in input "Data (RGB format)". The output "new picture" of Draw Unflattened Pixmap.vi is connected to the output "Image Out" of RGB2picture.vi

The input "reference" of RGB2picture.vi that contains a reference to the indicator "Image Out" defines the element that the "DrawAreaSize" property refers to. . The "DrawAreaSize" property addresses only to images controls and indicators and modifies their size according to the value of the cluster connected o this property. In this specific case, the input cluster "Draw Area Size" that includes the dimensions of the image is connected to the "DrawAreaSize" property. In this way is assured that the image indicator (specifically the control "Image out") will be of the same size as the image that will represent.

The RGB2picture.vi can represent apart from color images and grayscale images as well. The only think necessary is to connect the single array that includes information about the image, to inputs "Red Channel", "Green Channel" and "Blue Channel" at the same time.

In this point the description of the sixth phase and of the block diagram of the Image GUI.vi is completed.

4.2.6 Results – Conclusions

In Figure 164, the front panel of the ImageGUI.vi is shown, after the edge detection on the image cameraman.bmp is completed, which is represented to indicator “Image in”. The process result is shown in the image represented by the indicator “Image Out”.

Figure 164. Thefront panel of ImageGUI.vi

In Figure 165 the front panel of the ImageGUI.vi is shown, after the direct DCT and the IDCT is applied on the image peppers.bmp, which is represented to indicator “Image in”. The process result is shown in the image represented by the indicator “Image Out”.

Figure 165. The front panel of ImageGUI.vi

In Figure 16 the front panel of ImageGUI.vi is shown, after the McGovern algorithm of the direct DCT and the IDCT is applied on the image peppers.bmp, which is represented to indicator “Image in”. The process result is shown in the image represented by the indicator “Image Out”.

Figure 166. The front panel of ImageGUI.vi

In Figure 167 the front panel of ImageGUI.vi is shown, after the image boat.bmp is encoded and decoded according to the JPEG standard, which is represented to indicator “Image in”. The process result is shown in the image represented by the indicator “Image Out”.

Figure 167. The front panel of ImageGUI.vi

In Figure 168, the front panel of the ImageGUI.vi is shown, after the histogram equalization of the image house_256rgb.bmp in the RGB space. The initial image is represented to indicator “Image in”. The process result is shown in the image represented by the indicator “Image Out”.

Figure 168. The front panel of ImageGUI.vi

In Figure169, the front panel of the ImageGUI.vi is shown, after the histogram equalization of the image house_256rgb.bmp in the YCbCr space. The initial image is represented to indicator “Image in”. The process result is shown in the image represented by the indicator “Image Out”.

Figure 169. The front panel of ImageGUI.vi

5. Bibliography

[1] National Instruments, "LabVIEW Test Integration Toolkit for TI DSP", Semptember 2003.

[2] The Mathworks, "Link fot Code Composer Studio Development Tools, User's Guide version 2", 2006.

[3] National Instruments, "LabVIEW User Manual", April 2003.

[4] Texas Instruments, "Code Composer Studio IDE, Getting Started Guide", SPRU509F, May 2005.

[5] Texas Instruments, "Real–Time Data Exchange", SPRY012, Feb.1998.

[6] Nasser Kehtarnavaz and Namjin Kim, "Digital Signal Processing System – Level Design Using LabVIEW", Elsevier, 2005.

[7] Rulph Chassaing, Digital "Signal Processing and Applications with the C6713 and C6416 DSK", John Wiley & Sons, 2005.

[8] Sem M. Kuo, Bob H.Lee and Wenshun Tian, "Real-Time Digital Signal Processing", John Wiley & Sons, 2006.

[9] Istvan A.Szabo and Lajos Harasztosi, "Ways to use LabVIEW to aid DSP education", Proc. of the EDERS-2006, 2nd European DSP Education & Research Symposium, Munich, Germany, 4 April, 2006.

[10] E. Zigouris, D. Petropoulos, M. Kristalli and M. Hatzigiorgaki., "An integrated low-cost laboratory enviroment for digital image processing applications", Proc. of the ICSES'04, International Conference on Signals and Electronics Systems, pp.569-572, Poznan, Poland, 13-15 September, 2004.