Programmer's Manual

Digital Gamma Finder (DGF)

DGF-4C Rev. F

Version 4.04, September 2009

XIA LLC

31057 Genstar Road
Hayward, CA 94544 USA

Phone: (510) 401-5760; Fax: (510) 401-5761
http://www.xia.com

Disclaimer

1 Overview

This manual is divided into three major sections. The first section is a description of the DGF-4C C Driver which is currently used in the DGF-4C Viewer. Advanced users can build their own user interface using the user accessible functions in the driver. The second section is a reference guide to program the DGF-4C modules via the C Driver. This will be interesting to those users who want to integrate the DGF-4C modules into their own data acquisition system. The third section describes those user accessible variables that control the functions of the DGF-4C modules. Advanced and curious users can use this section to better understand the operation of the DGF-4C. Additionally, this manual also includes instructions on how to write User DSP code (Appendix A) and to control DGF modules using CAMAC commands (Appendix B).

The scope of this document is all DGF-4C modules with serial numbers 1400 through 1499.

2 DGF-4C C Driver

The DGF-4C C Driver consists of a group of C functions which can be used to configure DGF modules, make MCA or list mode runs and retrieve data from DGF modules. These functions can be compiled as a WaveMetrics Igor XOP file which is currently used by the DGF-4C Viewer, a dynamic link library (DLL) or static library to be used in customized user interfaces or applications. In order to better illustrate the usage of these functions, an overview of the operation of DGF is given below and the usage of these functions is mentioned wherever appropriate.

At first the DGF-4C C Driver needs to be initialized. This is a process in which the names of system configuration files and variable names are downloaded to the driver. The function **C_Dgf4c_Hand_Down_Names** is used to achieve this.

The second step is to boot the DGF modules. It involves downloading all FPGA configurations and booting the digital signal processor (DSP). It concludes with downloading all DSP parameters (the instrument settings) and commanding the DSP to program the FPGAs and the on-board digital to analog converters (DAC). All this has been encapsulated in a single function **C_Dgf4c_Boot_System**.

Now, the instrument is ready for data acquisition. The function used for this purpose is **C_Dgf4c_Acquire_Data**. By setting different run types, it can be used to start, end or poll a data acquisition run (list mode run, MCA run, or special task runs like acquiring ADC traces). It can also be used to retrieve list mode or histogram data from the DGF modules.

After checking the quality of a MCA spectrum, a DGF user may decide to change one or more settings like energy filter rise time or flat top. The function used to change DGF settings is **C_Dgf4c_User_Par_IO**. This function converts a user parameter like energy filter rise time in μs into a number understood by the DGF hardware or vice versa.

Another function, **C_Dgf4c_Buffer_IO**, is used to read data from DSP's internal memory to the host or write data from the host into the internal memory. This is useful for diagnosing DGF modules by looking at their internal memory values. The other usage of this function is to read, save, copy or extract DGF's configurations though its settings files.

In a multi-module DGF system, it is essential for the host to know which module and which channel it is communicating to. The function **C_Dgf4c_Set_Current_ModChan** is used to set the current module and channel.

The detailed description of each function is given below.

C_Dgf4c_Hand_Down_Names

Syntax

```
long C_Dgf4c_Hand_Down_Names (
  char *Names[],       // An array containing the names to be downloaded
  char *Name);         // A string indicating the type of names (file or
                       // variable names) to be downloaded
```

Description

Use this function to download file or variable names from host user interface to the DGF-4C C driver. The driver needs these file names so that it can read the DGF hardware configurations from the files stored in the host computer and download these configurations to the DGF. The variable names are used by the driver to obtain the indices of DSP variables when the driver converts user variable values into DSP variable values or vice versa.

Parameter description

Names is a two dimensional string array containing either the file names or the variable names. It can be one of the following four sets of names whose selection depends on the other parameter *Name*:

1. **All_Files** is a string array which has (MAX_NUMBER_OF_MODULES+5) elements. Currently MAX_NUMBER_OF_MODULES is defined as 24. The first five elements of All_Files are the name of system FPGA file, DSP code binary file, DSP I/O parameter values file, DSP code I/O variable names file, and DSP code memory variable names file. The remaining elements are FIPPI file names for each module. All file names should contain the complete path name. Note: all modules use the same system FPGA and DSP codes, but could use different FIPPI files. An example of All_Files is given in Table 3.2.

2. **Module_Global_Names** is a string array containing global variable names which are applicable to all modules, e.g. number of modules in the crate, the CAMAC controller type, the SCSI number, and the CAMAC crate ID, etc. Module_Global_Names currently can hold 64 names. If less than 64 names are needed (which is the current case), the remaining names should be defined as empty strings. A detailed description of Module_Global_Names is given in Table 3.6.

3. **Global_Data_Names** is a string array containing global variable names which are applicable to each individual module, e.g. module number, module CSR, coincidence pattern, and run type, etc. Global_Data_Names can currently hold 64 names. If less than 64 names are needed (which is the current case), the remaining names should be defined as empty strings. A detailed description of Global_Data_Names is given in Table 3.6.

4. **User_Var_Names** is a string array containing variable names which are applicable to individual channels of individual modules, e.g. channel CSR, filter rise time, filter flat top, voltage gain, and DC offset, etc. User _Var_Names currently can hold 64 names. If less than 64 names are needed (which is the current case), the remaining names should be defined as empty strings. A detailed description of User_Var_Names is given in Table 3.6.

Name is a string variable used to select which set of names to be handed down. It can be one of the following four choices: "ALL_FILES", "MODULE_GLOBAL_NAMES", "GLOBAL_DATA_NAMES", or "USER_VAR_NAMES".

Return values

Value	Description	Error Handling
0	Success	None
-1	Invalid name	Check the second parameter *Name*

Usage example

```
// download module global names; define Module_Global_Names first
C_Dgf4c_Hand_Down_Names(Module_Global_Names, "MODULE_GLOBAL_NAMES");

// download global data names; define Global_Data_Names first
C_Dgf4c_Hand_Down_Names(Global_Data_Names, "GLOBAL_DATA_NAMES");

// download user variable names; define User_Var_Names first
C_Dgf4c_Hand_Down_Names(User_Var_Names, "USER_VAR_NAMES");

// download file names; define All_Files first
C_Dgf4c_Hand_Down_Names(All_Files, "ALL_FILES");
```

C_Dgf4c_Boot_System

Syntax

```
long C_Dgf4c_Boot_System (
  long Boot_Pattern);    // The DGF boot pattern
```

Description

Use this function to boot all DGF modules in the system. Before booting the modules, it initializes the CAMAC communication port, and if CAMAC Master is going to be used, loads the CAMAC station number register.

Parameter description

Boot_Pattern is a bit mask used to control the boot pattern of DGF modules:

Bit 0: Boot system FPGA
Bit 1: Boot FIPPI
Bit 2: Boot DSP
Bit 3: Load DSP parameters
Bit 4: Apply DSP parameters (call Set_DACs and Program_FIPPI)

Under most of the circumstances, all the above tasks should be executed to initialize the DGF modules, i.e. the *Boot_Pattern* should be 0x1F.

Return values

Value	Description	Error Handling
0	Success	None
<0	Boot failed	Check the error message reported by the driver

Usage example

```
// boot DGF-4C modules
ret = C_Dgf4c_Boot_System(0x1F);
if(ret < 0)
// error handling
```

C_Dgf4c_User_Par_IO

Syntax

```
long C_Dgf4c_User_Par_IO (
  double *User_Par_Values,     // A double precision array containing the
                               // user parameters to be transferred
  char *User_Par_Name,         // A string variable which designates
                               // which type of user parameters to be
                               // transferred
  long direction);             // Transfer direction (read or write)
```

Description

Use this function to transfer user parameters between the user interface, driver and DSP's I/O memory. Some of these parameters are applicable to all DGF modules in the system, like CAMAC Controller ID or SCSI Bus number. Other parameters are applicable to either a DGF module (independent of its four channels), e.g. coincidence pattern, or any of the four channels in a DGF module, e.g. energy filter settings. For those parameters which need to be transferred to or from DSP's internal memory (other parameters such as number of modules are only used by the driver), this function calls another function **UA_PAR_IO** which first converts these parameters into numbers that are recognized by both the DSP and the driver then performs the transfer.

Parameter description

User_Par_Values is a double precision array containing the parameters to be transferred. Depending on another input parameter *User_Par_Name*, different *User_Par_Values* array should be used. Totally three *User_Par_Values* arrays should be defined and all of them are one-dimensional arrays. The corresponding relationship between *User_Par_Values* and *User_Par_Name* is listed in Table 2.1.

Table 2.1: The Combination of User_Par_Name and User_Par_Values.

User_Par_Name	User_Par_Values		
	Name	Size	Data Type
MODULE_GLOBAL_NAMES	Module_Global_Values	64	Double precision
GLOBAL_DATA_NAMES	Global_Data_Values	64×24	Double precision
SYNCH_WAIT or IN_SYNCH			
Element of array User_Var_Names	User_Values	64×24×4	Double precision

The way to fill the *User_Values* array is to fill the channel first then the module. First 64 values are stored in the array for channel 0, and then repeat this for other three channels. At that time, 64×4 values have been filled for module 1. Then repeat this for the remaining

modules. For the *Global_Data_Values* array, first store 64 values for module 1, and then repeat this for other modules.

User_Par_Name is a string variable which selects what type of parameters to be transferred. It can be one of the following choices:

User_Par_Name	**Function**	**Notes**
"MODULE_GLOBAL_NAMES"	transfer parameters applicable to all modules	r/w
"GLOBAL_DATA_NAMES"	transfer all module parameters applicable to current modules	r/w
"SYNCH_WAIT"	broadcast SYNCH_WAIT to all modules.	w
"IN_SYNCH"	broadcast IN_SYNCH to all modules.	w
"FILTERRANGE"	Transfer FILTERRANGE and apply updates to all channels of current module	w
< Element of array **User_Var_Names>**	transfer channel parameters for current channel of current module	r/w read updates all channel input parameters
"MCA_RUN"	Read out run statistics for current module	r
"TAU"	Find decay time for current channel	r
"BLCUT"	Find BLcut value for current channel	r
<any other value>	Read all channel input parameters for current channel of current module	r

direction indicates the transfer direction of parameters:
0 - download (write) parameters from the user interface to the driver;
1 - upload (read) parameters from the driver to the user interface.

Return values

Value	Description	Error Handling
0	Success	None
<0	Transfer failed	Check the error message reported by the driver

Usage example

```
// set global data variable MODULE_CSRA to 0x2400
Global_Data_Values[Find_Xact_Global_DATA_Match("MODULE_CSRA")]=0x2400;
// download MODULE_CSRA to DSP
C_Dgf4c_User_Par_IO(Global_Data_Values, " GLOBAL_DATA_NAMES", 0);
// set user variable ENERGY_RISETIME to 6.0 µs
User_Var_Values[Find_Xact_User_Match ("ENERGY_RISETIME")]=6.0;
```

```
// download ENERGY_RISETIME to DSP
C_Dgf4c_User_Par_IO(User_Var_Values, "ENERGY_RISETIME", 0);
```

C_Dgf4c_Acquire_Data

Syntax

```
long C_Dgf4c_Acquire_Data (
  long Run_Type,              // Data acquisition run type
  unsigned int *User_data,    // An unsigned 32-bit integer array
                              // containing the data to be transferred
  char *file_name);           // Name of the file used to store list
                              // mode or histogram data
```

Description

Use this function to acquire ADC traces, MCA spectrum, or list mode data. The string variable *file_name* needs to be specified when stopping a MCA run or list mode run in order to save the data into a file, or when calling those special list mode runs to retrieve list mode data from a saved list mode data file. In all other cases, *file_name* can be specified as an empty string. The unsigned 32-bit integer array *User_data* is only used for acquiring ADC traces (control task 0x4), reading out list mode data or MCA spectrum. In all other cases, *User_data* can be any unsigned integer array with arbitrary size. Make sure that *User_data* has the correct size and data type before reading out ADC traces, list mode data, or MCA spectrum.

Parameter description

Run_Type is a 16-bit word whose lower 12-bit specifies the type of either data run or control task run and upper 4-bit specifies actions (start\stop\poll) as described below.

Lower 12-bit:

0x100,0x101,0x102,0x103	list mode runs
0x200,0x201,0x202,0x203	fast list mode runs
0x301	MCA run
0x1 - 0x15	control task runs
0x3	adjust offsets
0x4	acquire ADC traces

Upper 4-bit:

0x1000	start new run
0x2000	resume run
0x3000	stop run and automatically store spectrum data
0x4000	poll
0x500x	list mode special runs
0x5000	Parse list mode data file
0x5001	Locate list mode traces
0x5002	Read list mode traces

0x5003	Read list mode energies
0x5004	Read list mode event PSA values
0x6000	stop list mode run during repeated runs
0x7000	manually read spectrum from module
0x8000	manually read spectrum from a MCA file
0x9000	stop any data run

file_name is a string variable which specifies the name of the output file. It needs to have the complete file path.

Return values

Value	Description	Error Handling
0*	Success	None
<0	Data acquisition failed	Check the error message reported by the driver

***NOTE:** when polling the status of a data acquisition run (Run_Type = 0x4000), the return value of C_Dgf4c_Acquire_Data will depend on the run type:

Data run (list mode or MCA run):	0	run is still in progress
	1	run has finished
Control task runs:	0	run has finished
	1	run is still in progress

Usage example

```
// start a new list mode run
C_Dgf4c_Acquire_Data(0x1100, dummy, " ");

// wait until the run has ended
while( ! C_Dgf4c_Acquire_Data(0x4100, dummy, " ") ) {;}

// stop run and save list mode run data
C_Dgf4c_Acquire_Data(0x6100, dummy, file_name_1);

// store energy histogram
C_Dgf4c_Acquire_Data(0x3100, dummy, file_name_2);
```

C_Dgf4c_Set_Current_ModChan

Syntax

```
long C_Dgf4c_Set_Current_ModChan (
  unsigned short Module,       // Module number to be set
  unsigned short Channel);     // Channel number to be set
```

Description

Use this function to set the current module number and channel number.

Parameter description

Module is an unsigned 16-bit integer which specifies the current module to be set. Module should be in the range of 1 to 23.

Channel is an unsigned 16-bit integer which specifies the current channel to be set. Channel should be in the range of 0 to 3.

Return values

Value	Description	Error Handling
0	Success	None
<0	Failed to set module or channel number	Check the error message reported by the driver

Usage example

```
// Set current module to 1 and current channel to 3
C_Dgf4c_Set_Current_ModChan(1, 3);
```

C_Dgf4c_Buffer_IO

Syntax

```
long C_Dgf4c_Buffer_IO (
  unsigned short *Values,    // An unsigned 16-bit integer array
                             // containing the data to be transferred
  unsigned short type,       // Data transfer type
  unsigned short direction,  // Data transfer direction
  char *file_name);          // File name
```

Description

Use this function to: 1) download or upload DSP parameters between the user interface and the DGF modules; 2) save DSP parameters into a settings file or load DSP parameters from a settings file and applies to all modules present in the system; 3) copy parameters from one module to others or extracts parameters from a settings file and applies to selected modules.

Parameter description

Values is an unsigned 16-bit integer array used for data transfer between the user interface and DGF modules. *type* specifies the I/O type. *direction* indicates the data flow direction. The string variable *file_name* contains the name of settings files. Different combinations of the three parameters - *Values*, *type*, *direction* – designate different I/O operations as listed in Table 2.2.

Table 2.2: Different I/O operations using function C_Dgf4c_Buffer_IO.

Type	Direction	Values	I/O Operation
0	0	DSP I/O variable values	Write DSP I/O variable values to modules
	1		Read DSP I/O variable values from modules
1	0*	Values to be written	Write to certain locations of the data memory
	1	All DSP variable values	Read all DSP variable values from modules
2	0	N/A**	Save current settings in all modules to a file
	1		Read settings from a file and apply to all modules
3	0	Values[0] – source module number; Values[1] – source channel number; Values[2] – copy/extract pattern bit mask; Values[3], Values[4], … - destination channel pattern	Extract settings from a file and apply to selected modules
	1		Copy settings from a source module to destination modules
4	N/A***	Values[0] – address; Values[1] – length	Specify the location and number of words to be written into the data memory

*Special care should be taken for this I/O operation since mistakenly writing to some locations of the data memory will cause the system to crash. The Type 4 I/O operation should be called first to specify the location and the number of words to be written before calling this one. If necessary, please contact XIA for assistance.
**Any unsigned 16-bit integer array could be used here.
***Direction can be either 0 or 1 and it has no effect on the operation.

Return values

Value	Description	Error Handling
0	Success	None
<0	I/O operation failed	Check the error message reported by the driver

Usage example

```
// Download DSP parameters to the current DGF module; DSP_Values is a
// pointer pointing to the DSP parameters; no need to specify file name
// here.
C_Dgf4c_Buffer_IO(DSP_Values, 0, 0, "");

// Read DSP memory values from the current DGF module; Memory_Values is
// a pointer pointing to the memory block; no need to specify file name
// here.
C_Dgf4c_Buffer_IO(Memory_Values, 1, 1, "");
```

Options for Compiling DGF-4C C Driver

DGF-4C C Driver can be compiled as either a WaveMetrics Igor XOP file which is currently used in the DGF-4C Viewer, or a standalone C-Library. The latter option can be used by advanced users to integrate DGF modules into their own data acquisition systems.

The following table summarizes the required files for these two options.

Table 2.3: Two options for compiling the DGF-4C C Driver.

Compilation Option	Required Files		
	C source files	C header files	Library files
All options	boot.c, camac.c, camacdll.c, CC32.c, Communication.c, dgf4c_c.c, utilities.c	boot.h, Camacdll.h, globals.h, sharedfiles.h, utilities.h, Libcc32.h, vpcic32d.h, Winaspi.h xia_common.h, xia_usb2_api.h, xia_usb2_cb.h	pcicc32_ni.lib, pcicc32_ni.dll, wnaspi32.dll, USB2Dll.lib
Additional for standalone C-Library			
Additional for Igor XOP	dgf4c_iface.c, dgf4c_igor.c, Dgf4cWinCustom.rc	dgf4c_iface.h, IgorXOP.h, VCExtraIncludes.h, Xop.h, XOPResources.h, XOPStandardHeaders.h, XOPSupport.h, XOPSupportWin.h, XOPWinMacSupport.h	XOPSupport x86.lib, IGOR.lib

3 Control DGF-4C Modules via DGF-4C C Driver

3.1 Initialization

DGF-4C modules sitting in a CAMAC crate can be initialized using those functions described in Section 2. As an example, we assume two DGF-4C modules sit in slot 3 and 11, respectively. The CAMAC controller sits in slot 24 functioning as a master controller. Users are also encouraged to read the sample code shipped with the C Driver.

3.1.1 Initialize global variables

As discussed in Section 2, we assume that three global variable arrays have been defined: Module_Global_Values, Global_Data_Values and User_Values. For these three global variable arrays, we also need to define three global name arrays: Module_Global_Names, Global_Data_Names and User_Var_Names, respectively. Table 3.5 lists the names contained in each of these name arrays. The order of placing these names into the array is not important since the C Driver uses search functions to locate each name at run time.

Additionally, a string array All_Files containing the file names for the initialization is also needed. Table 3.1 lists the file names needed to initialize two DGF-4C modules.

Table 3.1: File Names in All_Files.

All_Files	File Name	Note
All_Files[0]	C:\XIA\DGF4C\Firmware\dgf4c.bin	System FPGA configurations
All_Files[1]	C:\XIA\DGF4C\DSP\DGFcodeF.bin	DSP code
All_Files[2]	C:\XIA\DGF4C\Configuration\test.itx	Settings file
All_Files[3]	C:\XIA\DGF4C\DSP\DGFcodeF.var	File of DSP I/O variable names
All_Files[4]	C:\XIA\DGF4C\DSP\DGFcodeF.lst	File of DSP memory variable names
All_Files[5]	C:\XIA\DGF4C\Firmware\fdgf4c4E.bin	FIPPI configuration for Module 1 (Rev. E)
All_Files[6]	C:\XIA\DGF4C\Firmware\Sysdgfrevf.bin	FIPPI configuration for Module 2 (Rev. F)

The global variable array, Module_Global_Values, also needs to be initialized before C Driver functions can be called to start the initialization. Table 3.2 lists those global variables.

Table 3.2: Initialization of Module_Global_Values.

Module_Global_Names	Module_Global_Values	Note
NUMBER_MODULES	2	The total number of DGF-4C modules
CONTROLLER_ID	0	0: J73A, 1: CC32, 2: offline
SCSI_BUS	0	Usually 0 or 1; could be 0 to 7
CRATE_ID	1	The crate number on the front panel dial of the controller
CAMAC_MASTER	1	1: enable, 0: disable; use master controller

FAST_CAMAC	0	1: enable, 0: disable; use level-1 fast CAMAC transfer
LAM_ENABLE	0	1: enable, 0: disable; use LAM interrupt
AUTO_PROCESSLMDATA	0	0: Do not process LM data
SLOT_WAVE[0]	24	CAMAC master controller
SLOT_WAVE[1]	3	Module 1 sits in slot 3
SLOT_WAVE[2]	11	Module 2 sits in slot 11

3.1.2 Boot DGF-4C modules

The boot procedure for DGF-4C modules includes the following steps. First, all the global parameter names should be downloaded by calling function C_Dgf4c_Hand_Down_Names. Then function C_Dgf4c_User_Par_IO should be called to initialize the global variable array Module_Global_Values. After that, function C_Dgf4c_Hand_Down_Names should be called again to download the file name array All_Files. Finally, function C_Dgf4c_Boot_System should be called to boot the modules. The following code is an example showing how to boot the DGF-4C modules using the C Driver functions.

Table 3.3: An Example Code Illustrating How to Boot DGF-4C Modules.

```
// download module global names
C_Dgf4c_Hand_Down_Names(Module_Global_Names, "MODULE_GLOBAL_NAMES");
// download global data names
C_Dgf4c_Hand_Down_Names(Global_Data_Names, "GLOBAL_DATA_NAMES");
// download user variable names
C_Dgf4c_Hand_Down_Names(User_Var_Names, "USER_VAR_NAMES");
// initialize module global values
C_Dgf4c_User_Par_IO(Module_Global_Values, "MODULE_GLOBAL_VALUES", 0);
// download file names
C_Dgf4c_Hand_Down_Names(All_Files, "ALL_FILES");
// boot DGF-4C modules
C_Dgf4c_Boot_System(0x1F);
// set current module and channel number
C_Dgf4c_Set_Current_ModChan(1,0);
```

3.2 Setting DSP variables

The host computer communicates with the DSP by setting and reading a set of variables called DSP I/O variables. These variables, totally 416 unsigned 16-bit integers, sit in the first 416 words of the data memory. The first 256 words, which store input variables, are both readable and writeable, while the remaining 160 words, which store pointers to various data buffers and run summary data, are only readable. The exact location of any particular variable in the DSP code will vary from one code version to another. To facilitate writing robust user code, we provide a reference table of variable names and addresses with each

DSP code version. Included with your software distribution is a file called DGFcodeF.var. It contains a two-column list of variable names and their respective addresses. Thus you can write your code such that it addresses the DSP variables by name, rather than by fixed location.

It should come as no surprise that many of the DSP variables have meaningful values and ranges depending on the values of other variables. A complete description of all interdependencies can be found in Section 4. All of these interdependencies have been taken care of by the DGF-4C C Driver. So instead of directly setting DSP variables, users only need to set the values of those global variables defined in Table 3.5. The C Driver will then convert these values into corresponding DSP variable values and download them into the DSP data memory. On the other hand, if users want to read out the data memory, the C Driver will first convert these DSP values into the global variable values. The code shown in Table 3.5 is an example of setting DSP variables through the C Driver. Table 3.5 gives a complete description of all the global variables being used by the DGF-4C C Driver.

```
// set global data variable MODULE_CSRA to 0x2400
Global_Data_Values[Find_Xact_Global_DATA_Match("MODULE_CSRA")]=0x2400;
// download MODULE_CSRA to DSP
C_Dgf4c_User_Par_IO(Global_Data_Values, " GLOBAL_DATA_NAMES", 0);
// set user variable ENERGY_RISETIME to 6.0 µs
User_Var_Values[Find_Xact_User_Match ("ENERGY_RISETIME")]=6.0;
// download ENERGY_RISETIME to DSP
C_Dgf4c_User_Par_IO(User_Var_Values, "ENERGY_RISETIME", 0);
```

Table 3.4: An Example Code to Illustrating How to Set DSP Variables.

Table 3.5: Descriptions of Global Variables in DGF-4C.

Module_Global_Names	**I/O Type**	**Unit**	**Corresponding DSP Variables**	**Legal Range of Variable Values**
NUMBER_MODULES	Read/Write	N/A	N/A	[1, Max # of Modules)
CONTROLLER_ID	Read/Write	N/A	N/A	Check Controller
SCSI_BUS	Read/Write	N/A	N/A	Check SCSI bus
CRATE_ID	Read/Write	N/A	N/A	Check Crate
CAMAC_MASTER	Read/Write	N/A	N/A	0 or 1
FAST_CAMAC	Read/Write	N/A	N/A	0 or 1
LAM_ENABLE	Read/Write	N/A	N/A	0 or 1
C_LIBRARY_RELEASE	Read only	N/A	N/A	N/A
C_LIBRARY_BUILD	Read only	N/A	N/A	N/A
AUTO_PROCESSLMDATA	Read/Write	N/A	N/A	0 or 1
SLOT_WAVE	Read/Write	N/A	N/A	N/A
Global_Data_Names	**I/O Type**	**Unit**	**Corresponding DSP Variables**	**Legal Range of Variable Values**
MODULE_NUMBER	Read only	N/A	MODNUM	[1, Max # of Modules)

MODULE_CSRA	Read/Write	N/A	MODCSRA	N/A
MODULE_CSRB	Read/Write	N/A	MODCSRB	N/A
MODULE_FORMAT	Read/Write	N/A	MODFORMAT	N/A
MAX_EVENTS	Read/Write	N/A	MAXEVENTS	N/A
COINCIDENCE_PATTERN	Read/Write	N/A	COINCPATTERN	[0, 65535]
ACTUAL_COINCIDENCE_WAIT	Read/Write	cyc	COINCWAIT	[0, 16383]
MIN_COINCIDENCE_WAIT	Read only	cyc	COINCWAIT	[0, 16383]
SYNCH_WAIT	Read/Write	N/A	SYNCHWAIT	0 or 1
IN_SYNCH	Read/Write	N/A	INSYNCH	0 or 1
RUN_TYPE	Write only	N/A	RUNTASK	0x100, … 0x301
BUFFER_HEAD_LENGTH	Read only	N/A	BUFHEADLEN	6
EVENT_HEAD_LENGTH	Read only	N/A	EVENTHEADLEN	3
CHANNEL_HEAD_LENGTH	Read only	N/A	CHANHEADLEN	9, 4, 2
OUTPUT_BUFFER_LENGTH	Read only	N/A	LOUTBUFFER	8192
NUMBER_EVENTS	Read only	N/A	NUMEVENTSA, NUMEVENTSB	N/A
RUN_TIME	Read only	s	RUNTIMEA, RUNTIMEB, RUNTIMEC	N/A
FILTERRANGE	Read/Write	N/A	FILTERRANGE	1, 2, 3, 4, 5, 6
DBLBUF_CSR	Read/Write	N/A	DBLBUFCSR	0..15
DECIMATION	Read only	N/A	DECIMATION	1, 2, 3, 4, 5, 6
TOTAL_TIME	Read only	s	TOTALTIMEA, TOTALTIMEB, TOTALTIMEC	N/A
MODULE_CRSC	Read/Write	N/A	MODCSRC	N/A
User_Var_Names	**I/O Type**	**Unit**	**Corresponding DSP Variables**	**Legal Range of Variable Values**
CHANNEL_CSRA	Read/Write	N/A	CHANCSRA	N/A
CHANNEL_CSRB	Read/Write	N/A	CHANCSRB	N/A
ENERGY_RISETIME	Read/Write	µs	SLOWLENGTH	Depends on decimation
ENERGY_FLATTOP	Read/Write	µs	SLOWGAP	Depends on decimation
TRIGGER_RISETIME	Read/Write	µs	FASTLENGTH	[0.025, 0.775]
TRIGGER_FLATTOP	Read/Write	µs	FASTGAP	[0, 0.75]
TRIGGER_THRESHOLD	Read/Write	N/A	FASTTHRESH	[0, 4095/FASTLENGTH]
VGAIN	Read/Write	V/V	GAINDAC	(0, 16]
VOFFSET	Read/Write	V	TRACKDAC	(-3, 3)
TRACE_LENGTH	Read/Write	cyc	TRACELENGTH	[0, 100]
TRACE_DELAY	Read/Write	cyc	TRIGGERDELAY	[0, 100)
PSA_START	Read/Write	cyc	PSAOFFSET	(0, 100)
PSA_END	Read/Write	cyc	PSALENGTH	(0, 100)
EMIN	Read/Write	N/A	ENERGYLOW	[0, 32768)
BINFACTOR	Read/Write	N/A	LOG2EBIN	1, 2, 3, 4, 5, 6
TAU	Read/Write	µs	PREAMPTAUA, PREAMPTAUB	N/A
BLCUT	Read/Write	N/A	BLCUT	N/A
XDT	Read/Write	µs	XWAIT	>= 0.075
BASELINE_PERCENT	Read/Write	%	BASELINEPERCENT	(0, 100)
CFD_THRESHOLD	Read/Write	%	CFDTHR	(0, 100)
MULTIPLICITY_PULSE_WIDTH	Read/Write	cyc	FTPWIDTH	[1, 65535]

INTEGRATOR	Read/Write	N/A	INTEGRATOR	N/A
CHANNEL_CSRC	Read/Write	N/A	CHANCSRC	N/A
GATE_WINDOW	Read/Write	cyc	GATEWINDOW	N/A
GATE_DELAY	Read/Write	cyc	GATEDELAY	N/A
BLAVG	Read/Write	N/A	LOG2BWEIGHT	N/A
LIVE_TIME	Read only	s	LIVETIMEA, LIVETIMEB, LIVETIMEC	N/A
INPUT_COUNT_RATE	Read only	cps	FASTPEAKSA, FASTPEAKSB, LIVETIMEA, LIVETIMEB, LIVETIMEC, FTDTA, FTDTB, FTDTC	N/A
FAST_PEAKS	Read only	N/A	FASTPEAKSA, FASTPEAKSB,	N/A
OUTPUT_COUNT_RATE	Read only	cps	NOUTA, NOUTB, LIVETIMEA, LIVETIMEB, LIVETIMEC,	N/A
NOUT	Read only	N/A	NOUTA, NOUTB	N/A
GATE_RATE	Read only	cps	GCOUNTA, GCOUNTB, LIVETIMEA, LIVETIMEB, LIVETIMEC,	N/A
GATE_COUNTS	Read only	N/A	GCOUNTA, GCOUNTB	N/A
FTDT	Read only	N/A	FTDTA, FTDTB, FTDTC	N/A
SFDT	Read only	N/A	SFDTA, SFDTB, SFDTC	N/A
GDT	Read only	N/A	GDTA, GDTB, GDTC	N/A

3.3 Access spectrum memory or list mode data

3.3.1 Access spectrum memory

The MCA spectrum memory is fixed to 32K words (32 bits per word) per channel, residing in the external memory.. The Spectrum memory is accessible after a MCA run, or a list mode run if histogramming energy is requested. The following code in Table 3.6 is an example of how to start a MCA run and read out the MCA spectrum after the run is finished.

Table 3.6: Accessing the spectrum memory.

```
// start a MCA run; dummy is an unsigned 32-bit integer array of any size
C_Dgf4c_Acquire_Data(0x1301, dummy, " ");
// wait until run has ended
while( ! C_Dgf4c_Acquire_Data(0x4301, dummy, " ") ) {;}
// stop run and save MCA spectrum to a file
C_Dgf4c_Acquire_Data(0x3301, dummy, file_name);
// read out the MCA spectrum and put it to array User_data
C_Dgf4c_Acquire_Data(0x7301, User_data, " ");
```

3.3.2 Access list mode data

In a list mode run setup, you can do any number of runs in a row. The first run would be started as a NEW run. This clears all histograms in memory. Once the I/O buffer is full and has been read out, you can RESUME running. This keeps the histogram memory intact and you can accumulate spectra over many runs. The example code shown in Table 3.7 illustrates this.

Table 3.7: Command sequence for multiple list mode runs in a row.

```
C_Dgf4c_Acquire_Data(0x1100, dummy, " ");       // start a new list mode run
k = 1;     // initialize counter
do{
    while( ! C_Dgf4c_Acquire_Data(0x4100, dummy, " ") ) {;}  // wait until run has ended
    C_Dgf4c_Acquire_Data(0x6100, dummy, file_name_1);       // stop run and save list mode run data
    k ++;
    if(k > Nruns)
        break;
    C_Dgf4c_Acquire_Data(0x2100, dummy, " "));  // issue ResumeRun command
}while(1);
C_Dgf4c_Acquire_Data(0x3100, dummy, file_name_2);       // store energy histogram
```

The list mode data in the I/O buffer can be written in a number of formats. User code should access three DSP variables BUFHEADLEN, EVENTHEADLEN, and CHANHEADLEN in

the settings file of a particular run to navigate through the data set. To facilitate the access of list mode data after a run is finished, the DGF-4C C Driver provides several utility routines to parse the list mode data saved in the output file and read out the waveform, energy, or PSA values of each event. The code in Table 3.8 shows how to read waveforms from a list mode file.

Table 3.8: An Example Code Showing How to Access List Mode Data.

```
C_Dgf4c_Acquire_Data(0x1100, dummy, " ");   // start a new list mode run
while( ! C_Dgf4c_Acquire_Data(0x4100, dummy, " ") ) {;}  // wait until run has ended
C_Dgf4c_Acquire_Data(0x6100, dummy, file_name_1);  // store list mode data in a file
C_Dgf4c_Acquire_Data(0x3100, dummy, file_name_2);  // store energy histogram in a file
C_Dgf4c_Acquire_Data(0x5100, listmodewave, file_name_1);  // parse list mode file
totaltraces = 0;
for(i=0; i<2; i++)
    totaltraces += listmodewave[i+24];   // sum the total number of traces for the two modules
traceposlen = (long)malloc(totaltraces*3*4);  // allocate memory to hold position and length information
C_Dgf4c_Acquire_Data(0x5001, traceposlen, file_name_1);      // locate traces
Trace0 = (unsigned short)malloc(traceposlen[1]+2);     // allocate memory to hold the first trace
Trace0[0] = traceposlen[0];     // position of the first trace
Trace0[1] = traceposlen[1];     // length of the first trace
C_Dgf4c_Acquire_Data(0x5002, Trace0, file_name_1);   // read out the first trace and put it into trace0
```

4 User Accessible Variables

User parameters are stored in the data memory space of the on-board DSP. The organization is that of a linear memory with 16-bit words. Subsequent memory locations are indicated by increasing addresses. The data memory space, as seen by the host computer, starts at 0x4000.

There are two sets of user-accessible parameters. 256 words in data memory are used to store input parameters. These can and must be set properly by the user application. A second set of 160 words is used for results furnished by the DGF-4C module. These should not be overwritten.

As of this writing the start address for the input parameter block is InParAddr=0x4000 and for the output parameter block it is OutParAddr=0x4100, i.e. the two blocks are contiguous in memory space. We provide an ASCII file named DGFcodeE.var which contains in a 2-column format the offset and name of every user accessible variable. We suggest that user code use this information to create a name→address lookup table, rather than relying on the parameters retaining their address offsets with respect to the start address.

The input parameter block is partitioned into 5 subunits. The first contains 64 data that pertain to the DGF-4C as a whole. It is followed by four blocks of 48 words, which describe the settings of the four channels.

Below we describe the module and channel parameters in turn. Where appropriate, we show how a variable can be viewed using the DGF-4C Viewer.

The DGF-4C C Driver function used to write or read these parameters is C_Dgf4c_User_Par_IO, and the corresponding DSP parameters to the user-defined global variables are listed in Table 3.6.

4.1 Module input parameters

MODNUM: Logical number of the module. This number will be written into the header of the list mode buffer to aid offline event reconstruction**.** It is normally set by the C library during the boot process matching the order entered in the SLOT_WAVE.

Igor controls: Slot Wave.

C global: SLOT_WAVE

ControlTask to apply change: none

MODCSRA: The Module Control and Status Register A

Bit 0: reserved

Bit 1: If set, DSP acquires 32 data buffers in each list mode run and stores the data in external memory. If not set, only one buffer is acquired and the data is kept in local memory. **Must be set/cleared for all modules in the system.** If set, clear bit 0 of DBLBUFCSR

Bit 2: If cleared, connect module to auxiliary bus and share triggers with other modules. If set, do not connect and share triggers only within the module.

Bit 3: If set, compute sum of channel energies for events with more hits in more than one channel and put into addback spectrum

Bit 4: If set, spectra for individual channels contain only events with a single hit. Only effective if bit 3 is set also.

Bit 5 -9: reserved

Bit 10: If set, module terminates the event trigger line of the auxiliary bus.

Bit 11,12: reserved

Bit 13: If set, module terminates the fast trigger line of the auxiliary bus.

Bit 14-15: reserved

Igor controls: checkboxes in the MODULE REGISTER panel.

C global: MODULE_CSRA

ControlTask to apply change: 5.

MODCSRB: The Module Control and Status Register B

Bit 0: Execute user code routines programmed in user.dsp.

Bits 1-15: Reserved for user code.

Igor controls: none.

C global: MODULE_CSRB.

ControlTask to apply change: none

MODCSRC: The Module Control and Status Register C

Bits 0-15: Reserved

Igor controls: none.

C global: MODULE_CSRC.

ControlTask to apply change: none

MODFORMAT: List mode data format descriptor. Currently it is not in use**.**

Igor controls: none.

C global: MODULE_FORMAT

ControlTask to apply change: none

RUNTASK: This variable tells the DGF-4C what kind of run to start in response to a run start request. Six run tasks are currently supported.

RunTask	Mode	Trace Capture	CHANHEADLEN
0	Slow control run	N/A	N/A
256 (0x100)	Standard list mode	Yes	9
257 (0x101)	Compressed list mode	Yes	9
258 (0x102)	Compressed list mode	Yes	4
259 (0x103)	Compressed list mode	Yes	2
769 (0x301)	MCA mode	No	N/A

RunTask 0 is used to request slow control tasks. These include programming the trigger/filter FPGAs, setting the DACs in the system, transfers to/from the external memory, and calibration tasks.

RunTask 256 (0x100) requests a standard list mode run. In this run type all bells and whistles are available. The scope of event processing includes computing energies to 16-bit accuracy, and performing pulse shape analyses for improved energy resolution and better time of arrival measurements. Nine words of results, including time of arrival, energy, XIA pulse shape analysis, user pulse shape analysis, etc. are written into the I/O buffer for each channel. Level-1 buffer is not used in this RunTask.

RunTask 257 (0x101) requests a compressed list mode run. Both Level-1 buffer and I/O buffer are used in this RunTask, but no traces are written into the I/O buffer. Nine words of results, including time of arrival, energy, XIA pulse shape analysis, user pulse shape analysis, etc. are written into the I/O buffer for each channel.

RunTask 258 (0x102) requests a compressed list mode run. The only difference between RunTask 258 and 257 is that in RunTask 258, only four words of results (time of arrival, energy, XIA pulse shape analysis, user pulse shape analysis) are written into the I/O buffer for each channel.

RunTask 259 (0x103) requests a compressed list mode run. The only difference between RunTask 259 and 257 is that in RunTask 259, only two words of results (time of arrival and energy) are written into the I/O buffer for each channel.

RunTasks 512-515 are no longer supported

RunTask 769 (0x301) requests a MCA run. The raw data stream is always sent to the level-1 buffer, independent of MODCSRA. The data-gathering interrupt routine fills that buffer with raw data, while the event processing routine removes events after processing. If the interrupt routine finds the level-1 buffer to be full, it will ignore events until there is room again in the buffer. The run will not abort due to buffer-full condition. This run type does not write data to the I/O buffer.

Igor controls: *Run Type* in the *Run* tab

C global: RUN_TYPE.

ControlTask to apply change: none

CONTROLTASK: Use this variable to select a control task. Consult the control tasks section of this manual for detailed information. The control task will be launched when you issue a run start command with RUNTASK=0.

See section 4.6 for a list of acceptable values

Igor controls: none.

C global: none.

ControlTask to apply change: none

MAXEVENTS: The module ends its run when this number of events has been acquired. In the DGF Viewer, the maximum value for MAXEVENTS is automatically calculated and applied when a run mode is chosen from the run type pulldown menu. The calculation is based on the RUN_TYPE and the TRACELENGTH, using BUFHEADLEN, EVENTHEADLEN, and CHANHEADLEN given by the RUN_TYPE and the length of the output buffer (8K words):

$$\text{Event length} = \text{EVENTHEADLEN} + \Sigma_i\,(\text{CHANHEADLEN} + \text{TRACELENGTH}_i)$$

$$\text{MAXEVENTS}_{max} = (8\text{K} - \text{BUFHEADLEN})/\,\text{Event length}$$

Set MAXEVENTS = 0 if you want to switch off this feature, e.g., when logging spectra or when there is no need to enforce a fixed number of events. In particular, if 4 channels are "good" and MAXEVENTS is computed accordingly, but the majority of events have only a single channel with a pulse, the buffer is only filled up to about 1/4 when MAXEVENTS is reached. So in this case it would be more efficient to disable the MAXEVENT limit by setting it to zero. The parameter is ignored in an MCA mode run.

Igor controls: *Maximum no. of events* in the *Run* tab

C global: MAX_EVENTS. The C library enforces an upper limit for MAXEVENTS.

ControlTask to apply change: none

COINCPATTERN: The user can request that certain coincidence/anticoincidence patterns are required for the event to be accepted. With four channels there are 16 different hit patterns, and each can be individually selected or marked for rejection by setting the appropriate bit in the COINCPATTERN mask.

Consider the 4-bit hit pattern 1010. The two 1's indicate that channel 3 (MSB) and channel 1 have reported a hit. Channels 2 and 0 did not. The 4-bit word reads as 10(decimal). If this hit pattern qualifies as an acceptable event, set bit 10 in the COINCPATTERN to 1. The 16 bit in COINCPATTERN cover all combinations. Setting COINCPATTERN to 0xFFFF causes the DGF-4C to accept any hit pattern as valid.

Igor controls: Checkboxes in the COINCIDENCE PATTERN panel

C global: COINCIDENCE_PATTERN.

ControlTask to apply change: 5 (in the future)

COINCWAIT: Duration of the coincidence time window in clock ticks (12.5 ns each)

Normally, this parameter is used to define a time window from the first event validation until the final hit pattern is latched for the coincidence test. This accommodates delays between channels due to cabling or the physics of the experiment.

In addition, since the coincidence test is applied only after validation, a minimum coincidence window is required if validation requires different amounts of time due to differences in the energy filter settings. For example, a channel with the energy filter rise time set to 6 µs will start the coincidence window 2 µs before a channel with a filter rise time of 8 µs, and thus simultaneous events in the second channel will be lost unless the coincidence window is at least 2 µs. The following formula should be used to determine the minimum COINCWAIT:

$$\text{COINCWAIT} = \max(\text{PEAKSEP}*2^{\text{FILTERRANGE}})_{\text{ch0-ch3}} - \min(\text{PEAKSEP}*2^{\text{FILTERRANGE}})_{\text{ch0-ch3}}$$

Only channels marked as "good" in the CHANCSRA need to be included in this computation.

When energy filter rise time or flat top are changed from Igor, the C library computes the minimum COINCWAIT and sets the variable to at least this minimum. The C library will never reduce the current value, since it can not distinguish if it has been increased by the user for experimental reasons. The minimum value is reported back to the user for reference. The Igor control shows the wait time in ns, the DSP variable and the C global in clock cycles (12.5ns)

Constraints**:** COINCWAIT >= 1
COINCWAIT <= 2^14 -1

Igor controls: *Actual Coinc window* in the COINCIDENCE PATTERN *panel*

C global: ACTUAL_COINCIDENCE_WAIT.

ControlTask to apply change: 5

SYNCHWAIT: Controls run start behavior. When set to 0 the module simply starts or resumes a run in response to the corresponding request. When set to 1, the module will use the Busy/Sync loop to delay run start until all modules are ready, and stop runs when the first module fills its buffer.

Igor controls: Checkboxes in the *Run* tab

C global: SYNCH_WAIT

ControlTask to apply change: none

INSYNCH: InSynch is an input/output variable. It is used to clear the DGF-4C on-board clock at the start of the data acquisition. When INSYNCH is 1, no particular action is taken. If this variable is 0 and SYNCHWAIT =1, then all system timers are cleared at the beginning of the next data acquisition run (RUNTASK>=0x100). After run start, INSYNCH is automatically set to 1. In this way, all clocks in a multi-module system are synchronized to within ~3 clock cycles.

Igor controls: Checkboxes in the *Run* tab

C global: IN_SYNCH

ControlTask to apply change: none

HOSTIO: A 4 word data block that is used to specify command options.

Igor controls: none

C global: none

ControlTask to apply change: none

RESUME: Set this variable to 1 to resume a data run; otherwise, set it to 0. Set to 2 before stopping a list mode run prematurely.

Igor controls: none

C global: none

ControlTask to apply change: none

FILTERRANGE: The energy filter range downloaded from the host to the DSP. It sets the number of ADC samples (2^FILTERRANGE) to be averaged before entering the filtering logic. The currently supported filer range in the signal processing FPGA includes 1, 2, 3, 4, 5 and 6.

Igor controls: Filter Range in the *Settings* tab (in the future)

C global: FILTERRANGE.

ControlTask to apply change: 5

MODULEPATTERN: reserved

NNSHAREPATTERN: reserved

CHANNUM: The chosen channel number. May be modified internally for tasks looping over all 4 channels, or to pass on current channel to user code. Should be set by host before starting controltask 4 and 6 to indicate which channel to operate on. (Previously HOSTIO was used in controltask 4). We recommend to always change CHANNUM when changing the channel that is addressed in the user interface.

Igor controls: none

C global none

ControlTask to apply change: none

DBLBUFCSR: A register containing several bits to control the double buffer (ping pong) mode to read out external memory. In the future, these control bits may be moved to the CSR register in the System FPGA.

Bit 0: Enable double buffer: If this bit is set, transfer list mode data to external memory in double buffer mode. **Must be set/cleared for all modules in the system.** If set, clear bit 1 of MODCSRA. Set by host, read by DSP.

Bit 1: Host read: Host sets this bit after reading a block from external memory to indicate DSP can write into it again. Set by host, read and cleared by DSP.

Bit 2: reserved

Bit 3: Read_128K_first: If run halted because host did not read fast enough and both blocks in external memory are filled, DSP will set this bit to indicate host to first read from block 1 (staring at address128K), else (if zero) host should first read from block 2. Set by DSP, read by host. Cleared by DSP at runstart or resume

Igor controls: Radio buttons in the *Run* tab (in the future)

C global DBLBUFCSR

ControlTask to apply change: 5

U00: Many unused, but reserved, data blocks have names of the structure Unn. Those unused data blocks which reside in the block of input parameters for each channel are called UNUSEDA and UNUSEDB.

XDATLENGTH: Length of a data block to be downloaded from the host for debugging of PSA. Use XDATLENGTH=0 as the default value for normal operation.

USERIN: A block of 16 input variables used by user-written DSP code.

4.2 Channel input variables

All channel-0 variables end with "0", channel-1 variables end with "1", etc. In the following explanations the numerical suffix has been removed. Thus, e.g., CHANCSRA0 becomes CHANCSRA, etc.

CHANCSRA: The control and status register bits switch on/off various aspects of the DGF-4C operation.

Bit 0: Respond to group triggers only.
Set this bit if you want to control the waveform acquisition for non-triggering channels by a triggering master channel. For this option to work properly choose one channel as the master and have its Trigger_Enable bit set. All dependent channels should have their Trigger_Enable bit cleared. Set bit 0 in

all slave channels. You should also set it the master channel to ensure equal time of arrivals for the fast trigger signal, which is used to halt the FIFOs.

Note: To distribute group triggers between modules, bit 2 in the variable MODCSRA has to be set as well.

Bit 1: reserved

Bit 2: Good channel.
Only channels marked as good will contribute to spectra and list mode data.

Bit 3: Read always
Channels marked as such will contribute to list mode data, even if they did not report a hit. This is most useful when acquiring induced signal waveforms on spectator electrodes, i.e., electrodes that did not collect any net charge, but only saw a transient induced signal.

Bit 4: Enable trigger.
Set this bit for channels that are supposed to contribute to an event trigger.

Bit 5: Trigger positive.
Set this bit to trigger on a positive slope; clear it for triggering on a negative slope. The trigger/filter FPGA can only handle positive signals. The Pixie handles negative signals by inverting them immediately after entering the FPGA.

Bit 6: GFLT required.
Set this bit if you want to validate events based on a global external signal input through the GFLT connector. When the bit is cleared, the GFLT signal is ignored. When set, the event is accepted only if validated by GFLT. To be validated, the GFLT ignal must be a logic 0 at time PEAKSEP * 2^(FILTERRANGE) after the rising edge of a pulse. Polarity can be reversed in CHANCSRC

Bit 7: Histogram energies.
Set this bit to histogram energies from this channel in the on-board MCA memory.

Bit 8: Reserved.
Set to 0.

Bit 9: If set, allow negative number as the result of the pulse height computation. This may be useful in list mode runs to return a rough measure of an inverted pulse. Due to the binary representation of negative numbers, the pulse height will be histogrammed into bin 64K-abs(pulse height) of the spectrum. This option is ignored in MCA runs.

Bit 10: Compute constant fraction timing.
This pulse shape analysis computes the time of arrival for the signal from the recorded waveform. The result is stated in units of $1/256^{th}$ of a sampling period (12.5 ns). Time zero is the start of the waveform.

Bit 11: Reserved.
(Enabled multiplicity contribution in DGF-4C)

Bit 12: GATE required
Set this bit if you want to validate or veto events based on the VETO signal. When set, the event is accepted only if validated by VETO. To be validated, the VETO input must have a rising edge within a time window (defined by GATEWINDOW and GATEDELAY) around the rising edge of a detector pulse. When the bit is cleared, the VETO input is not used for event validation, but its status is reported in the list mode output data. Polarities of the edge starting the Window and the status required for accepting events can be selected in CHANCSRC.

Bit 13: If set, use the local trigger to latch the time stamp even in group trigger mode, else use the distributed group trigger.

Bit 14: Estimate energy if channel not hit.
If set, the DSP reads out energy filter values and computes the pulse height for a channel that is not hit, for example when "read always" in group trigger mode. If not set, the energy will be reported as zero if the channel is not "hit"

Bit 15: Reserved.

Igor controls: Checkboxes in the CHANNEL REGISTER panel

C global CHANNEL_CSRA

ControlTask to apply change: 5

CHANCSRB: Control and status register B. (for user code)

Bit 0: If set, call user written DSP code.

Bit 1: If set, all words in the channel header except Ndata, TrigTime and Energy will be overwritten with the contents of URETVAL. Depending on the run type, this allows for 6, 2 or 0 user return values in the channel header.

Bit2..15: reserved. Set to 0.
Bits 2 and 3 are used in MPI custom code.

Igor controls: none

C global CHANNEL_CSRB

ControlTask to apply change: none

CHANCSRC: Control and status register C.

Bit 0: GFLT polarity.
Controls polarity of GFLT to be considered present for accepting events, i.e. GFLT must be zero to record events instead of 1.

Bit 1: GATE acceptance polarity.
Controls polarity of GATE to be considered present for accepting events, i.e. the GATE status latched at the time of the rising edge of the detector pulse must be zero to record events instead of 1.

Bit 2: Use GFLT for GATE.
If set, use GFLT input for fast validation of signal rising edge of pulse instead of the VETO input.

Bit 3: Disable pileup inspection.
If set, pulses are accepted even if a pileup was detected.

Bit 4: Disable out-of-range rejection
If set, pulses are accepted even if the ADC input goes out of range. This can be used for detectors with occasional very large pulses. The energy filter essentially saturates for the time the signal is out of range, which means the reported energy is a measure of how long the signal is out of range and thus a coarse measure of the energy (assuming exponential decay)

Bit 5: Invert pileup inspection
If set, only accept events with pileup. May be useful to capture double pulse events.

Bit 6: Pause pileup inspection
If set, disable pileup inspection for 32 clock cycles (426 ns). May be useful for systems where the detector output shows significant ringing that causes two or more triggers on the same pulse (especially those with higher amplitude), to avoid these events to be rejected as piled up.

Bit 7: Gate edge polarity inverted
If set, the GATE Window counter is started at the falling edge of the GATE signal instead of on the rising edge.

Note: Only one of bits 3, 5, and 6 is meant to be set at the same time, but this is not enforced.

Igor controls: Checkboxes in the CHANNEL REGISTER panel

C global CHANNEL_CSRC

ControlTask to apply change: 5

GAINDAC: Reserved and not supported.

TRACKDAC: This DAC determines the DC-offset voltage. The offset in volts displayed in Igor and contained in the C global VOFFSET can be calculated using the following formula:

Offset [V] = 2.5V * ((32768 - TRACKDAC) / 32768)

Constraints**:** TRACKDAC >= 0
TRACKDAC <= 65535

Igor controls: *Offset [V]* in the *Calibrate* tab

C global VOFFSET

ControlTask to apply change: 0

SGA: The index of the relay combinations of the switchable gain amplifier. For a given value of SGA, the analog SGA gain is G = (1+Rf/Rg)/2 with

Rf = 2150 - 120*((SGA & 0x1)>0) - 270*((SGA &0x2)>0) - 560*((SGA &0x4)>0)

Rg = 1320 - 100*((SGA & 0x10)>0) - 300*((SGA & 0x20)>0)- 820*((SGA & 0x40)>0)

The C library computes the closest SGA setting for a given voltage gain VGAIN and adjusts the parameter DIGGAIN to compensate differences between VGAIN and the gain from SGA up to ±10%.

Constraints**:** SGA >= 0
SGA <= 127

Igor controls: *Gain [V/V]* in the *Calibrate* tab

C global VGAIN

ControlTask to apply change: 0

DIGGAIN: The digital gain factor for compensating the difference between the user-desired voltage gain and the SGA gain. This is computed by the C library and limited to 10% in the following way:

DG = voltage gain / SGA gain - 1.0;
DIGGAIN = 65535*DG if DG >0
= 65536 + 65535*DG if DG <0

Constraints**:** DIGGAIN >= 0 for positive DG
DIGGAIN <= 6554
. DIGGAIN >= 64K - 6554 for negative DG
DIGGAIN <= 64K

Other values between 6554 and 64K-6554 are possible, but may lead to binning errors or other undesirable effects

Igor controls: *Gain [V/V]* in the *Calibrate* tab

C global VGAIN

ControlTask to apply change: none

UNUSEDA0 or UNUSEDA1: Reserved.

SLOWLENGTH: The rise time of the energy filter (also called peaking time) depends on SLOWLENGTH:

Energy Filter Rise Time = SLOWLENGTH * 2^FILTERRANGE * 12.5 ns

Constraints**:** SLOWLENGTH >= 2
SLOWLENGTH + SLOWGAP <= 127

Igor controls: *Energy Filter Rise Time* in the *Settings* tab

C global: ENERGY_RISETIME

ControlTask to apply change: 5

SLOWGAP: The flat top of the energy filter (also called gap time) depends on SLOWGAP:

Energy Filter Flat Top = SLOWGAP * 2^ FILTERRANGE * 12.5 ns.

Constraints**:** SLOWGAP >= 3
SLOWLENGTH + SLOWGAP <= 127

Igor controls: *Energy Filter Flat Top* in the *Settings* tab

C global: ENERGY_FLATTOP

ControlTask to apply change: 5

FASTLENGTH: The rise time of the trigger filter depends on FASTLENGTH

Trigger Filter Rise Time = FASTLENGTH * 12.5 ns.

Constraints**:** FASTLENGTH >= 2
FASTLENGTH + FASTGAP <= 63

The value for the C global variable TRIGGER_RISETIME (in clock cycles, equal to FASTLENGTH) is computed by Igor from the Trigger Filter Rise Time value entered by the user. Limits are applied in the C library

Igor controls: *Trigger Filter Rise Time* in the *Settings* tab

C global: TRIGGER_RISETIME

ControlTask to apply change: 5

FASTGAP: The flat top of the trigger filter depends on FASTGAP**:**

Trigger Filter Flat Top = FASTGAP * 12.5 ns.

Constraints**:** FASTLENGTH >= 0
FASTLENGTH + FASTGAP <= 63

The value for the C global variable TRIGGER_FLATTOP (in clock cycles, equal to FASTGAP) is computed by Igor from the Trigger Filter Rise Time value entered by the user. Limits are applied in the C library

Igor controls: T*rigger Filter Flat Top* in the *Settings* tab

C global: TRIGGER_FLATTOP

ControlTask to apply change: 5

PEAKSAMPLE: This variable determines at what time the value from the energy filter will be sampled. Note that the following formulae depend on the filter range:

FILTERRANGE = 0: PEAKSAMPLE = max(0, SLOWLENGTH + SLOWGAP – 7)
FILTERRANGE = 1: PEAKSAMPLE = max(2, SLOWLENGTH + SLOWGAP – 4)
FILTERRANGE = 2: PEAKSAMPLE = SLOWLENGTH + SLOWGAP – 2
FILTERRANGE >= 3: PEAKSAMPLE = SLOWLENGTH + SLOWGAP – 1

If the sampling point is chosen poorly, the resulting spectrum will show energy resolutions of 10% and wider rather than the expected fraction of a percent. For some parameter combinations PEAKSAMPLE needs to be

varied by one or two units in either direction, due to the pipelined architecture of the trigger/filter FPGA.

Igor controls: none

C global: none

ControlTask to apply change: 5

PEAKSEP: This value governs the minimum time separation between two pulses. Two pulses that arrive within a time span shorter than determined by PEAKSEP will be rejected as piled up.

The recommended value is: PEAKSEP = PEAKSAMPLE +5

Constraints: If PEAKSEP >128, PEAKSEP = PEAKSAMPLE +1
0 < PEAKSEP - PEAKSAMPLE < 7

Igor controls: none

C global: none

ControlTask to apply change: 5

FASTTHRESH: This is the trigger threshold used by the trigger/filter FPGA. It is compared to the output of the fast filter; if the filter output is greater or equal to FASTTHRESH, a trigger is issued. For a pulse with a given height, the trigger filter output scales with the trigger filter rise time FASTLENGTH, i.e.

filter output = FASTLENGTH * pulse amplitude/4 * form factor

where ”pulse amplitude” is the amplitude in ADC units (as displayed in the oscilloscope) and form factor describes the effect of the shape of the pulse during FASTLENGTH. For a square pulse, form factor = 1; for a slow rising or fast decaying pulse, form factor will be less than 1 because the amplitude is not constant during FASTLENGTH.

The threshold value TRIGGER_THRESHOLD set in the *Settings* tab in Igor is scaled in the C library for FASTLENGTH so that a given value of TRIGGER_THRESHOLD causes triggering at the same pulse amplitude independent of FASTLENGTH:

FASTTHRESH = TRIGGER_THRESHOLD * FASTLENGTH

Constraints**:** FASTTHRESH >= 0
FASTTHRESH <= 4095
the lower 3 bits are ignored

Igor controls: *Threshold*, *Rise Time* in the *Settings* tab

C global: TRIGGER_THRESHOLD, TRIGGER_RISETIME

ControlTask to apply change: 5

MINWIDTH: Unused.

MAXWIDTH: This value aids the pile up inspection. MAXWIDTH is the maximum duration, in clock cycles (12.5 ns), which the output from the fast filter may spend over threshold. Pulses longer than that will be rejected as piled up. The recommended setting is zero to disable this feature. Otherwise, a possible starting value is

MAXWIDTH = FASTLENGTH + FASTGAP+ SignalRiseTime / 12.5 ns.

Constraints**:** MAXWIDTH >= 0
MAXWIDTH <= 255

Once the other parameters have been optimized with MAXWIDTH =0, one can use the MAXWIDTH cut to improve the pile up rejection at high count rates. MAXWIDTH should be tuned by observing the main energy peak in the spectrum for fixed time intervals. Once the MAXWIDTH cut is too tight there will be a loss of efficiency in the main peak. Setting MAXWIDTH to such a value that the efficiency loss in the main peak is acceptable will give the best overall performance in terms of efficiency and pile up rejection.

Igor controls: none

C global: none

ControlTask to apply change: 5

PAFLENGTH: Obsolete, but preserved for backwards compatibility

A FIFO control variable that needs to be written into the trigger/ filter FPGA. Using the programmable almost-full register we can time the waveform capturing thus that by the time the DSP is triggered at the end of the pile up inspection period the data of interest have percolated through to the begin of the FIFO and are available for read out without delay.

The acquired waveform will start rising from the baseline at a time delay after the beginning of the trace. This delay is a quantity that the user will want to set. In the DGF Viewer it is called Trace_Delay (measured in microseconds) and is available through the *Settings* tab. Igor converts the value entered in the DGF Viewer into clock cycles before passing it to the C library global. Limits are applied in the C library. The recommended setting for PAFLENGTH is:

PAFLENGTH = TRIGGERDELAY + Trace Delay/12.5ns

Constraints**:** PAFLENGTH >= 0
PAFLENGTH <= 4092

Note that PAFLENGTH should be adjusted only in multiples of 4, as the hardware ignores the lower two bits of this value.

Igor controls: *Trace Delay* in the *Settings* tab and energy filter settings

C global: TRACE_DELAY

ControlTask to apply change: 5

TRIGGERDELAY: Obsolete, but preserved for backwards compatibility

This is a partner variable to PAFLENGTH. For *all* filter ranges,

TRIGGERDELAY = (PEAKSEP -1)*2^(FILTERRANGE)

Note that TRIGGERDELAY should be adjusted only in multiples of 4, as the hardware ignores the lower two bits of this value. For MCA runs without taking traces, (trace length=0), TRIGGERDELAY should be 2.

Igor controls: *Trace Delay* in the *Settings* tab and energy filter settings

C global: none

ControlTask to apply change: 5

RESETDELAY: This variable sets the timeout to restart processing in the trigger/filter FPGA automatically after it captured an event, but has not received event validation. In most circumstances, event capture constitutes validation and this timeout does not apply. However, if a channel is not trigger enabled or responds to group triggers only, a local event by itself is not valid, so the channel waits for RESETDELAY clock cycles to receive external validation. In other words, RESETDELAY is the window for trigger disabled channels to be included in an event *later* triggered by another channel

Constraints**:** RESETDELAY >= 0
RESETDELAY <= 255

The default value is 29 and should normally not be changed by the user.

Igor controls: none

C global: none

ControlTask to apply change: 5

TRACELENGTH: This variable determines the length of captured waveforms in list mode runs (in clock cycles). Igor converts the value in microseconds entered in the DGF Viewer into clock cycles before passing it to the C library global. Limits are applied in the C library.

TRACELENGTH = Trace Length/12.5ns

Constraints**:** TRACELENGTH >= 0
TRACELENGTH <= 1024 (but see below)

Generally, if TRACELENGTH > 1024, the size of the FIFO memory in the trigger/filter FPGA, the C library will force the TRACELENGTH to be equal to 1024.

In a debug/test mode, if direct download to the DSP sets a value greater than 1024, the DSP will not read waveforms from FIFO memory, but instead from the ADC register. Since the DSP event readout occurs after the pileup inspection, the waveform will in this case only contain data from after the pulse. The time between samples is set by XWAIT

Igor controls: *Trace Length* in the *Settings* tab

C global: TRACE_LENGTH

ControlTask to apply change: 5

USERDELAY: This variable specifies the number of pre-trigger samples in the captured waveform. Igor converts the value entered in the DGF Viewer into clock cycles before passing it to the C library global. Limits are applied in the C library.

USERDELAY = Trace Delay/12.5 ns

Constraints**:** USERDELAY >= 0
USERDELAY <= TRACELENGTH

Replaces TRIGGERDELAY and PAFLENGTH. The current recommended handling for backwards compatibility is to compute TRIGGERDELAY, PAFLENGTH and USERDELAY and download them all to the module. When reading back settings from the module, TRIGGERDELAY and PAFLENGTH are used to compute Trace Delay.

Igor controls: *Trace Delay* in the *Waveform* tab

C global: TRACE_DELAY

ControlTask to apply change: 5

FTPWIDTH: Length of the pulse generated for this channel on the multiplicity output in clock cycles. Both Igor value and global variable are in units of clock cycles.

Constraints**:** FTPWIDTH >= 0
FTPWIDTH <= 255

Igor controls: *Pulse Width* in the CHANNEL REGISTER panel

C global: MULTIPLICITY_PULSE_WIDTH

ControlTask to apply change: 5

GATEDELAY, GATEWINDOW: These variables set the coincidence window for the Gate signal to reject events. At the rising edge of the Gate signal, and internal Gate status bit goes high for the duration of GATEWINDOW. A GATEDELAY after a fast trigger the status bit is latched into GATEBIT. GATEBIT can be used to reject events in the FPGA, and it is reported in the hit pattern in the list mode data stream for offline processing if no online rejection is desirable.
Igor converts the values entered in the DGF Viewer into clock cycles before passing it to the C library globals.

GATEWINDOW = Gate Window /12.5ns

GATEDELAY = Gate Dealy /12.5ns

Constraints: GATEWINDOW >= 1
GATEWINDOW <= 255
. GATEDELAY >= 1

GATEDELAY <= 255
GATEDELAY < PEAKSEP*2^FILTERRANGE
(for online rejection only)

Igor controls: *Gate Delay, Gate Window* in the CHANNEL REGISTER panel

C global: GATE_WINDOW, GATE_DELAY

ControlTask to apply change: 5

XWAIT: Extra wait states. XWAIT is used when acquiring untriggered traces in a control run with ControlTask = 4, e.g. the traces in the Igor oscilloscope display. The time ΔT between data points in the output buffer is

ΔT = XWAIT *12.5ns

If XWAIT > 12, a filter is implemented during the acquisition to return each data point as the average over (XWAIT-3)/5 samples.

Constraints**:** XWAIT >= 4
XWAIT <= 65533
If XWAIT >12, it has to be in the form XWAIT = 13+ N*5

In a test/debug mode, this parameter also controls how many extra clock cycles the DSP waits when reading extra long waveforms in real time from the ADC register rather than out of the FIFO memory. This occurs when acquiring data in list mode and asking for trace lengths longer than FIFOlength (1024), which is possible if the C library's tests are bypassed. The time between recorded samples is then ΔT = (3+XWAIT)*12.5ns.

Igor controls: *dT* in the OSCILLOSCOPE

C global: XDT

ControlTask to apply change: none

ENERGYLOW: Start energy histogram at ENERGYLOW. This value is subtracted from the computed pulse height before binning into the MCA spectrum. Only applies to list mode runs.

Constraints**:** ENERGYLOW >= 0
ENERGYLOW <= 65535

Igor controls: Em*in* in the *Calibrate* tab

C global: EMIN

ControlTask to apply change: none

LOG2EBIN: This variable controls the binning of the histogram. Energy values are always calculated to 16 bits precision. The energy value corresponds to 4 times the 14-bit ADC amplitude. The DGF modules, however, do not have enough histogram memory available to record 64k spectra, nor would this always be desirable. The user is therefore free to choose a lower cutoff for the spectrum (ENERGYLOW) and control the binning by downshifting the computed

energy. The following formula describes which MCA bin a value of Energy will contribute:

MCAbin = (Energy- ENERGYLOW) * 2^LOG2EBIN

As can be seen, Log2Ebin should be a negative number to achieve the correct behaviour. At run start the DSP program ensures that Log2Ebin is indeed negative by replacing the stored value by -abs(Log2Ebin). The C global BINFACTOR contains the absolute value of LOG2EBIN

Constraints**:** LOG2EBIN >= 65520 (equiv. -16)
LOG2EBIN <= 65535 (equiv. -1)
and additionally LOG2EBIN = 0 is allowed

The histogramming routine of the DSP takes care of spectrum overflows and underflows.

Igor controls: *Binning Factor* in the *Calibrate* tab

C global: BINFACTOR

ControlTask to apply change: none

CFDTHR: This sets the threshold of the software constant fraction discriminator. The threshold fraction CFD_THRESHOLD (f) is encoded as round(f*65536), with 0<f<1.

Constraints**:** CFDTHR >= 0
CFDTHR <= 65535

Igor controls: *Threshold* in the CHANNEL REGISTER panel

C global: CFD_THRESHOLD

ControlTask to apply change: none

PSAOFFSET, PSALENGTH: When recording traces and requiring any pulse shape analysis by the DSP, these two parameters govern the range over which the analysis will be applied. The analysis begins at a point PSAOFFSET sampling clock ticks into the trace, and is applied over a piece of the trace with a total length of PSALENGTH clock ticks.
Igor converts the values entered in the DGF Viewer into clock cycles before passing it to the C library globals.

PSAOFFSET = PSA Start / 12.5ns

PSALENGTH = (PSA End - PSA Start) / 12.5ns

Constraints**:** PSALENGTH >= 0
PSALENGTH <= TRACELENGTH – PSAOFFSET
PSAOFFSET >= 0
PSAOFFSET <= TRACELENGTH

Igor controls: *PSA Start, PSA End* in the *Settings* tab

C global: PSA_START, PSA_END.

ControlTask to apply change: none

INTEGRATOR: This variable controls the energy reconstruction in the DSP.

INTEGRATOR = 0: normal trapezoidal filtering

INTEGRATOR = 1: use gap sum only; good for scintillator signals

INTEGRATOR = 2: ignore gap sum; pulse height=leading sum – trailing sum; good for step-like pulses.

INTEGRATOR = 3,4,5: same as 1, but multiply energy by 2, 4, or 8

Igor controls: *Integrator* in the CHANNEL REGISTER panel

C global: INTEGRATOR.

ControlTask to apply change: none

BLCUT: This variable sets the cutoff value for baselines in baseline measurements. If BLCUT is not set to zero, the DSP checks continuously each baseline value to see if it is outside of the limit set by BLCUT. If the baseline value is within the limit, it will be used to calculate the average baseline value. Otherwise, it will be discarded. To reduce processing time, set BLCUT to zero to not check baselines.

ControlTask 6 can be used to measure baselines. The host computer can then histogram these baseline values and determine the appropriate value for BLCUT for each channel according to the standard deviation of the averaged baseline value. This is done automatically in the C library every time the decay time or the energy filter rise time or flat top is changed, setting BLCUT to 4 times the standard deviation.

The value of BLCUT depends on decay time, gain, filter settings, and the noise from the detector. Automatically computed values below 15 are suspicious and 15 is used instead. Values have to be measured as above and can not be derived easily from first principles.

Constraints: BLCUT >= 0
BLCUT <= 65535

Igor controls: *none*

C global: BLCUT.

ControlTask to apply change: none.

BASELINEPERCENT: This variable sets the target DC-offset level for automatic adjustment (ControlTask 3) in terms of the percentage of the ADC range.

Constraints**:** BASELINEPERCENT >= 0
BASELINEPERCENT <= 100

Igor controls: *Baseline (%)* in the OSCILLOSCOPE

C global: BASELINE_PERCENT.

ControlTask to apply change: none

XAVG: Only used in Controltask 4 for reading untriggered traces. XAVG stores the weight in the geometric-weight averaging scheme to remove higher frequency signal and noise components. The value is calculated as follows:

For a given dT (in µs), calculate the integer intdt = dT/0.0125
If intdt > 13, XAVG = floor(65536/((intdt-3)/5))
If intdt < 13, XAVG = 65535.

Igor controls: *dT* in the OSCILLOSCOPE tab

C global: XDT

ControlTask to apply change: none

UNUSEDB0 or UNUSEDB1: Reserved.

CFDREG: Reserved for FPGA-based constant fraction discriminator.

LOG2BWEIGHT: The DGF-4C module measures baselines continuously and effectively extracts DC-offsets from these measurements. The DC-offset value is needed to apply a correction to the computed energies. To reduce the noise contribution from this correction baseline samples are averaged in a geometric weight scheme. The averaging depends on LOG2BWEIGHT:

$DC\_avg_{new} = DC\_avg_{old} + (DC - DC\_avg_{old}) * 2^{LOG2BWEIGHT}$

DC is the latest measurement and DC_avg is the average that is continuously being updated. At the beginning, and at the resuming, of a run, DC_avg is seeded with the first available DC measurement.

The DSP ensures that LOG2BWEIGHT will be negative. Larger (absolute) numbers mean the previous baseline measurements contribute more. The noise contribution from the DC-offset correction falls with increased averaging. The standard deviation of DC_avg falls in proportion to sqrt(2^LOG2BWEIGHT).

When using a BLCUT value from a noise measurement (cf control task 6) the DGF-4C will internally adjust the effective Log2Bweight for best energy resolution, up to the maximum value given by LOG2BWEIGHT. Hence, the LOG2BWEIGHT setting should be chosen at low count rates (dead time < 10%). Best energy resolutions are typically obtained at values of -3 to -4 (in 16bit signed integer format), and this parameter does not need to be adjusted afterwards.

The C global variable BLAVG stores the absolute value of LOG2BWEIGHT.

Constraints**:** LOG2BWEIGHT >= 65520 (equiv. -16)
LOG2BWEIGHT <= 65535 (equiv. =-1)
and additionally LOG2BWEIGHT = 0

Igor controls: *none*

C global: BLAVG.

ControlTask to apply change: none.

PREAMPTAUA, PREAMPTAUB: High word and low word of the preamplifier's exponential decay time. The two variables are used to store the value with higher precision. The time τ is measured in μs. The two words are computed as follows.

PREAMPTAUA = floor(τ)
PREAMPTAUB = 65536 * (τ - PreampTauA)

To recover τ use: τ = PREAMPTAUA + PREAMPTAUB / 65536

Constraints**:** TAU >= 1/65536 μs
TAU <= 65535 μs

Igor controls: *Tau* in the *Calibrate* tab

C global: TAU.

ControlTask to apply change: none.

This ends the block of channel input data. Note that there are four equivalent blocks of input channel data, one for each DGF-4C input channel.

4.3 Module output parameters

We now show the output variables, again beginning with module variables and continuing afterwards with the channel variables. The output data block begins at the address 0x4100. Note, however, that this address could change. The output data block comprises of 160 words; 1 block of 32 is reserved for module data; 4 blocks of 32 words each hold channel data.

DECIMATION: This variable is a copy of the input parameter FILTERRANGE. It is copied as an output parameter for backwards compatibility

REALTIMEA, REALTIMEB, REALTIMEC: The 48-bit real time clock. A,B,C are the high, middle and low word, respectively. The clock is zeroed on power up, and in response to a synch request at run start (INSYNCH = 0, SYNCHWAIT = 1). Compute the real time (since boot or synchronization) using the following formula:

RealTime =(RealTimeA * 64K^2 + RealTimeB * 64K + RealTimeC) * 12.5ns

RUNTIMEA, RUNTIMEB, RUNTIMEC: The 48-bit run time clock. A,B,C words are as for the RealTime clock. This time counter is active only while a data

acquisition run is in progress. Comparing the run time with the total time allows judging the overhead due to data readout. Compute the run time using the following formula:

RunTime =(RUNTimeA * 64K^2 + RUNTimeB * 64K + RUNTimeC) * 12.5ns

TOTALTIMEA, TOTALTIMEB, TOTALTIMEC: A third 48-bit clock to track the total time an acquisition was requested by the host. RUNTIME excludes the time waiting for host readout, TOTALTIME is the closest to the true lab time passed since the most recent "new run" command (the first spill in a series). A,B,C words are as for the RealTime clock. Compute the total time using the following formula:

TotalTime =(TOTALTIMEA * 64K^2 + TOTALTIMEB * 64K + TOTALTIMEC) * 12.5ns

GSLTTIMEA, GSLTTIMEB, GSLTTIMEC: Unused.

NUMEVENTSA, NUMEVENTSB: Number of valid events serviced by the DSP. Again the high word carries the suffix A and the low word the suffix B.

DSPERROR: This variable reports error conditions:

= 0 (NOERROR), no error

= 1 (RUNTYPEERROR), unsupported RunType

= 2 (RAMPDACERROR), Baseline measurement failed

= 3 (EMERROR), writing to external memory failed

SYNCHDONE: This variable can be set to 1 to force the DSP out of an infinite loop caused by a malfunctioning synchronization loop, when a run start request was issued with SYNCHWAIT=1.

TEMPERATURE: reserved.

BUFHEADLEN: At the beginning of each run the DSP writes a buffer header to the list mode data buffer. BUFHEADLEN is the length of that header. Currently, BUFHEADLEN is 6, but this value should not be hardcoded, it should be read from the DSP to ensure upgrade compatibility.

EVENTHEADLEN: For each event in the list mode buffer, or the level-1 buffer, there is an event header containing time and hit pattern information. EVENTHEADLEN is the length of that header. Currently, EVENTHEADLEN is 3, but this value should not be hardcoded, it should be read from the DSP to ensure upgrade compatibility.

CHANHEADLEN: For each channel that has been read, there is a channel header containing energy and auxiliary information. CHANHEADLEN is the length of this header. CHANHEADLEN varies between 2 and 9 words depending on the run type (see RUNTASK).

The event and channel header lengths plus the requested trace lengths determine the maximum logically possible event size. The maximum event size is the sum of EVENTHEADLEN and the CHANHEADLENs plus the TraceLengths for all channels marked as good, i.e. which have bit 2 in the CHANCSRA set.

Example: With all four channels marked as good and required trace lengths of 1000 (i.e. 13.3µs) the maximum event size will be

MaxEventSize = EVENTHEADLEN + 4*(CHANHEADLEN + 1000)
= 4039

In the last line typical values for EVENTHEADLEN (3) and CHANHEADLEN (9) were substituted. BUFHEADLEN equals 6. Thus there is room for at least 2 events in the list mode data buffer, which is 8192 words long. But there is only room for 1 event in the level-1 buffer used in compressed RUNTASKs 0x101-103, which contains only 4060 words.

EMWORDS, EMWORDS2: Each of these variables are two-word arrays (high word first) counting the number of 16 bit words written to external memory.

USEROUT: 16 words of user output data, which may be used by user written DSP code.

AOUTBUFFER, LOUTBUFFER: Address and number of words of the list mode data buffer. The addresses are generated by the assembler/linker when creating the executable. On power up the DSP code makes these values accessible to the user. Note that the addresses may change with every new compilation. Therefore your code should never assume to find any given buffer at a fixed address.

HARDWAREID: ID of the hardware version. Read from a PROM on the DGF-4C module.

HARDVARIANT: Variant of the hardware

FIFOLENGTH: Length of the onboard FIFOs, measured in storage locations.

FIPPIID: ID of the FiPPI FPGA configuration

FIPPIVARIANT: Variant of the FiPPI FPGA configuration

INTRFCID: ID of the system FPGA configuration

INTRFCVARIANT: Variant of the system FPGA configuration

DSPRELEASE: DSP software release number

DSPBUILD: DSP software build number

4.4 Channel output parameters

The following channel variables contain run statistics. Again the variable names carry the channel number as a suffix. For example the LIVETIME words for channel 2 are LIVETIMEA2, LIVETIMEB2, LIVETIMEC2. Channel numbers run from 0 to 3.

LIVETIMEA, LIVETIMEB, LIVETIMEC: Total live time as measured by the trigger/filter FPGA of that channel. See the user manual for a description of the measured time. Convert the three LiveTime words into a live time using the formula:

LiveTime = (LIVETIMEA *64K^2 + LIVETIMEB *64K + LIVETIMEC) * 16*12.5ns

FASTPEAKSA, FASTPEAKSB: The number of events detected by the fast filter is: FAST_PEAKS = FASTPEAKSA*65536 + FASTPEAKSB

FTDTA, FTDTB, FTDTC: Fast Trigger dead time is the time the fast trigger output was above threshold and thus not ready to detect further triggers, as measured by the trigger/filter FPGA. See the user manual for a description of the measured time. Convert the three words into a time using the formula (note missing factor 16):

FTDT = (FTDTA *64K^2 + FTDTB *64K + FTDTC) *12.5ns

SFDTA, SFDTB, SFDTC: Slow Filter Dead Time is the time the associated with each pulse that prohibited acquisition of a second pulse, for example due to pileup inspection or DSP readout. See the user manual for a description of the measured time. Convert the three words into a time using the formula:

SFDT = (SFDTA *64K^2 + SFDTB *64K + SFDTC) *16 * 12.5ns

GCOUNTA, GCOUNTB: The number of gate pulses for this channel (high, low) GCOUNT = GCOUNTA *65536 + GCOUNTB

NOUTA, NOUTB: The number of output counts in this channel (high, low) NOUT = NOUTA *65536 + NOUTB

GDTA, GDTB, GDTC: Gate Dead Time is the time during which a channel was gated. See the user manual for a description of the measured time. Convert the three words into a time using the formula:

GDT = (GDTA *64K^2 + GDTB *64K + GDTC) *16 * 12.5ns

ICR: ICR is an averaged measure of the current input count rate. It is updated if a run is in progress or not. The averaging is implemented such that at every update,

$Average_{new} = (Average_{old} + Number\ fast\ triggers\ in\ update\ period)/2$

The value reported in the variable ICR is equal to $2*Average_{new}$. Updates occur every 32*64K clock cycles. Thus to compute the rate in counts/s, the value in ICR has to be divided by 32*64K * 12.5ns. The reported value is precise to about 50 counts/s, with a maximum count rate of about one million counts/s

OORF: OORF is an averaged measure of the fraction of time the channel is out of range. It is updated if a run is in progress or not. The averaging is implemented such that at every update,

$Average_{new} = (Average_{old} + Time\ out\ of\ range/64)/2$

The value reported in the variable OORF is equal to $2*Average_{new}$. Updates occur every 32*64K clock cycles. Thus to compute the out of range fraction in percent, the value in OORF has to be multiplied by (100% / 64K).

4.5 ADC data

The revision-F DGF-4C modules employ 14-bit waveform digitizing ADCs, operating at 40MSPS. Hence, the natural units are 25ns for a time step. The original waveform data are 14-bit unsigned numbers ranging from 0 to 16383. Derived quantities, however, are reported by the DGF to higher than 14-bit precision:

- Energy values are all reported as unsigned 16-bit numbers, and a pulse step covering the full range of the ADC would be reported as having amplitude of 65535. That is, an LSB of an energy value corresponds to $1/4^{th}$ of an original ADC step

- Waveform data are reported as untriggered traces in the Oscilloscope of the DGF4C-Viewer as 14-bit numbers (cf control task 4). The triggered traces in the list mode trace display of the DGF4C-Viewer acquired in regular data acquisition are shown as 16-bit numbers so that the pulse height matches the computed energy.

5 Control Tasks

The DSP can execute a number of control tasks, which are necessary to control hardware blocks that are not directly accessible from the host computer. The most prominent tasks are those to set the DACs, program the trigger/filter FPGAs and read the histogram memory. The following is a list of control tasks that will be of interest to the programmer.

To start a control task, set RUNTASK=0 and choose a CONTROLTASK value from the list below. Then start a run by setting bit 0 in the control and status register (CSR).

Control tasks respond within a few hundred nanoseconds by setting the RUNACTIVE bit (#13) in the CSR. The host can poll the CSR and watch for the RUNACTIVE bit to be deasserted. All control tasks indicate task completion by clearing this bit.

Execution times vary considerably from task to task, ranging from under a microsecond to 10 seconds. Hence, polling the CSR is the most effective way to check for completion of a control task.

Control Task 0: **SetDACs**
Write the gaindac and trackdac values of all channels into the respective DACs. Also program the SumDAC. Reprogramming the DACs is required to make effective changes in the values of the variables GAINDAC{0…3}, TRACKDAC{0…3} and SUMDAC.

Control Task 1: **Connect inputs**
Close the input relay to connect the DGF electronics to the input connector.

Control Task 2: **Disconnect inputs**
Open the input relay to disconnect the DGF electronics from the input connector.

Control Task 3: **Ramp offset DAC**
This is used for calibrating the offset DAC. For each channel the offset DAC is incremented in 2048 equal-size steps. At each DAC setting the DC-offset is determined and written into the I/O buffer. At the end of the task the I/O buffer holds the following data. Its 8192 words are divided up equally amongst the four channels. Data for channel 0 occupy the lowest 2048 words, followed by data for channel 1, etc. The first entry for each channel's data block is for a DAC value of 0, the last entry is for a DAC value of 65504. In between entries the DAC value is incremented in steps of 32.

An examination of the results will reveal a linearly rising or falling response of the ADC to the DAC increments. The slope depends on

the trigger polarity setting, i.e., bit 5 of the channel control and status register A (ChanCSRA). For very low and very big DAC values the ADC will be driven out of range and an unpredictable, but constant response is seen. From the sloped parts a user program can find the DAC value that is necessary for a desired ADC offset. It is recommended, that for unipolar signals an ADC offset of 400 units is chosen. For bipolar signals, like the induced waveforms from a segmented detector, the ADC offset would be 2048 units, i.e., midway between 0 and 4095.

Note that for both revision-D and revision-E modules, ADC waveforms are reported as 14-bit numbers, ranging from 0 to 16383. Hence, the DC-offsets should be adjusted to produce readings of 1600 and 8192 counts, respectively, for unipolar and bipolar signals.

A user program would use the result from the calibration task to find, set and program the correct offset DAC values.

Since the offset measurement has to take the preamplifier offset into account, this measurement must be made with the preamplifier connected to the DGF-4C input. The control task makes 16 measurements at each DAC step and uses the last computed DC-offset value to enter into the I/O buffer. Due to electronic noise, it may occasionally happen that none of the sixteen attempts at a base line measurement is successful, in which case a zero is returned. The user software must be able to cope with an occasional deviation from the expected straight line.

On exit, the task restores the offset DAC values to the values they had on entry.

ControlTask 4: **Untriggered Traces**

This task provides ADC values measured on all four channels and gives the user an idea of what the noise and the DC-levels in the system are. This function samples 8192 ADC words for the channel specified in CHANNUM. The XWAIT variable determines the time between successive ADC samples (samples are XWAIT * 12.5ns apart). In the DGF-4C Viewer XWAIT can be adjusted through the dT variable in the Oscilloscope panel. The results are written to the 8192 words long I/O buffer. Use this function to check if the offset adjustment was successful.

From the DGF-4C Viewer this function is available through the Oscilloscope Panel. Hit the Refresh button to start four consecutive runs with ControlTask 4 in the selected module, one for each channel.

ControlTask 5: **ProgramFiPPI**

Write all relevant data to the FiPPI control registers.

ControlTask 6: **Measure Baselines**

This routine is used to collect baseline values. Currently, DSP collects six words, B0L, B0H, B1L, B1H, time stamp, and ADC value, for each baseline. 1365 baselines are collected until the 8192-word I/O buffer is almost completely filled. The host computer can then read the I/O buffer and calculate the baseline according to the formula:

$B1 = (B1L+B1H*65536)/2^{(DECIMATION+8)}$
$B0 = (B0L+B0H*65536)/ 2^{(DECIMATION+8)}$
$TAU = PreampTauA+PreampTauB/65536$
$Baseline = B1-B0*e^{(-0.025*(SlowLength+SlowGap)*2\text{^}DECIMATION/TAU)}$

Baseline values can then be statistically analyzed to determine the standard deviation associated with the averaged baseline value and to set the BLCUT.
BLCUT should be about 3 times the standard deviation. Baseline values can also be plotted against time stamp or ADC value to explore the detector performance. BLCUT should be set to zero while running ControlTask 6.

ControlTask 22: Test EM write

This routine is used to write a test pattern from the DSP into the external memory (testing list mode data transfers). The data written is as follows:

Word (16bit)	Value	Notes
0	8002	Works as buffer length
1	MODNUM	Can be used to identify a module by writing MODNUM through CAMAC and reading the EM through USB.
2	0xAAAA	
3	0x5555	
4	0xCCCC	
5	0x3333	
6	0x1111	
7	0xEEEE	
8	0x8888	
9	0x7777	
10-8001	Repeat above words 2-9 for 999 times	
8002-8104	103, MODNUM, 25x (0x8888, 0x8888, 0x7777,0x7777), 0x8888 (testing odd sized buffer transfers)	
8105-8207	103, MODNUM, 25x (0xCCCC, 0xCCCC, 0x3333,0x3333), 0xCCCC (testing odd sized buffer transfers)	

ControlTask 25: clear external memory

This routine is used to clear the external memory.

ControlTask 26: Test histogramming

This routine is used to write a test pattern to the external memory by incrementing bin N for N times, for bins 0..4K. The result is a "spectrum" in channel 0 that forms a line with Ncounts = bin number for bins 0..4K. This procedure may take several seconds to complete.

ControlTask 7..21, 23-25, 26-127: reserved

ControlTask >128: reserved for tasks performed by C library, not DSP

6 Appendix A — User supplied DSP code

6.1 Introduction

It is possible for users to enhance the capabilities of the DGF-4C by adding their own DSP code. XIA provides an interface on the DSP level and has built support for this into the DGF-4C Viewer. The following sections describe the interfaces and support features.

6.2 The development environment

For the DSP code development, XIA uses and recommends version 5 or 6 of the assembler and linker distributed by Analog Devices. Both versions are in use at XIA and work fine.

It may be inconvenient, but is unavoidable to program the ADSP-2181 on board processor in assembler rather than in a higher level programming language like C. We found that code generated by the C-compiler is bloated and consequently runs very slow. As the main piece of the code could not be written in C at all, we did not burden our design by trying to be compatible with the C-compiler. Hence, using the C-compiler is currently not an option.

With the general software distribution we provide working executables and support files. To support user DSP programming we provide files containing pre-assembled forms of XIA's DSP code, together with a source code file that has templates for the user functions. The user templates have to be converted by the assembler and the whole project is brought together by the linker. XIA provides a link and a make file to assist the process.

In the DGF-4C Viewer we provide diagnostic tools to aid code developing and a data interface to exchange data between the host and the user code. The DGF-4C Viewer can, at any time, examine the complete memory content of the DSP and call any variable from any code section by name. A particularly useful added feature is the capability to download data in native format into the DSP and pretend that they were just acquired. The event processing routine, which calls the user code, is then activated and processes the data. This in-situ code testing allows the most control in the debugging process and is more powerful than having to rely on real signal sources.

6.3 Interfacing user code to XIA's DSP code

When the DSP is booted it launches a general initialization routine to reach a known, and useful, state. As part of this process a routine called UserBegin is executed. It is used to communicate addresses and lengths of buffers, local to the user code, to the host. The host finds this information in the USEROUT[16] buffer described in the main section of this document. The calling of **UserBegin** is not maskable. All other functions that are part of the user interface will be called only if bit 0 of MODCSRB is set at the time.

When a run starts, the DSP executes a run start initialization during which it will call **UserRunInit**. It may be used to prepare data for the event procesing routines.

When events are processed by the DSP code it may call user code in two different instances. Events are processed one channel at the time. For each channel with data, **UserChannel** is called at the end of the processing, but before the energy is histogrammed. UserChannel has access to the energy, the acquired wave form (the trace) and is permitted one return value. This is the routine in which custom pulse shape analysis will be performed.

After the entire event, consisting of data from one to four channels, has been processed the function **UserEvent** may be called. It may be used in applications in which data have to be correlated across channels.

At the end of a run the closing routine may call **UserRunFinish**, typically for updating statistics and similar run end tasks.

The above mentioned routines are described below, including the interface variables and the permissible use of resources.

6.4 The interface

The interface consists of five routines and a number of global variables. Data exchange with the host computer is achieved via two data arrays that are part of the I/O parameter blocks visible to the host. The total amount of memory available to the user comprises 2048 instructions and 1100 data words.

Interface DSP routines:

UserBegin:
This routine is called after rebooting the DSP. Its purpose is to establish values for variables that need to be known before the first run may start. Address pointers to data buffers established by the user are an example. The host will need know where to write essential data to before starting a run.

Since the DSP program comes up in a default state after rebooting UserBegin will always be called. This is different for the routines listed below, which will only be called if for at least one channel bit 0 of ChannelCSRB has been set.

UserRunInit:
This function is called at each run start, for new runs as well as for resumed runs. The purpose is to precompute often needed variables and pointers here and make them available to the routines that are being called on an event-by-event basis. The variables in question would be those that depend on settings that may change in between runs.

UserChannel:

This function is called for every event and every DGF-4C channel for which data are reported and for which bit 0 of the channel CSR_B (ChannelCSRB variable) has been set. It is called after all regular event processing for this channel has finished, but before the energy has been histogrammed.

UserEvent:
This function is called after all event processing for this particular event has finished. It may be used as an event finish routine, or for purposes where the event as a whole is to be examined.

UserRunFinish:
This routine is called after the run has ended, but before the host computer is notified of that fact. Its purpose is to update run summary information.

Global variables:

UserIn[16]	16 words of input data, also visible to host
UserOut[16]	16 words of output data, also visible to host
Uglobals[32]	32 words to pass global variables from the user code to the main code. The use of these variables is controlled by the main code
UretVal[6]	User output data to be incorporated into list mode data

The return value for UserChannel for list mode data is UretVal. It is an array of 6 words. If bit 1 of ChanCSRB is 0, only the first word is incorporated into the output data stream by the main code. (See Tables 4.2 to 4.6 in the user manual for the output data structure.) If the bit is 1, up to six values are incorporated, overwriting the XIA PSA value, the USER PSA value, the GSLT time, and the reserved word in the channel header. If the run type compresses the standard nine channel header words, the number of user return values is reduced accordingly (i.e only 2 words are available in RunTask 0x102, and no words in RunTask 0x103).

When entering UserChannel a number of global variables have been set by the DSP. These are listed in the file “INTRFACE.INC” as “externals:

Register usage:
For register usage restrictions, see the file “INTRFACE.INC”.

6.5 Debugging tools

Besides the debugging tools that are accessible through the DGF-4C Viewer, it is also possible to download data into the DGF data buffers and call the event processing routine. This allows for an in-situ test of the newly written code and allows exploring the valid parameter space systematically or through a Monte Carlo from the host computer. For this to work the module has to halt the background activity of continuous base line measuring. Next, data have to be downloaded and the event processing started. When done the host can read the results from the known address.

The process is fairly simple. The host writes the length of the data block that is to be downloaded into the variable XDATLENGTH. Then the data are written to the linear I/O buffer, the address and length of which are given in the variables AOUTBUFFER and LOUTBUFFER. Next the user starts a data run, and reads the results after the run has ended.

7 Appendix B — Control DGF-4C modules using CAMAC commands

Although DGF users are recommended to use to DGF-4C C Driver to program their DGF modules, CAMAC commands are the other alternative to control the modules.

7.1 CAMAC interface

The CAMAC interface through which the host communicates with the DGF-4C is implemented in its own FPGA. The configuration of this gate array is stored in a PROM, which is placed in the only DIP-8 IC-socket on the DGF-4C board. The interface conforms to the regular CAMAC standard, as well as the newer Level-1 fast CAMAC with a cycle time of 400 ns per read operation. The interface moves 16-bit data words at a time. The upper 8 bits of the read and write bus are ignored.

7.2 Initialization

The booting process for DGF-4C modules goes through the following steps:

1. A "Config" or "Interface" FPGA is configured at power up from PROM
2. The host computer configures the System FPGA and trigger/filter FPGAs through the "Config FPGA"
3. The host computer boots the DSP through the System FPGA
4. The host computer sets DSP variables through the System FPGA
5. The host computer starts a control task run to apply parameters to the FPGAs ("ProgramFippi")

FPGA configuration files will be found in the DGF4C\FirmWare directory. These are byte-oriented binary files. They should be written using block transfer mode, one byte at a time. The DGF-4C expects to see the data in the lower 8 bits of the CAMAC data way write bus.

Table 7.1 shows the CAMAC commands issued for steps 1-3 above. This sequence assumes the version register can now be read before the System FPGA is configured; still to be tested. (Previously a read before all System FPGAs in the crate were configured might have locked up the CAMAC bus). Reading the version register allows the software to automatically select the appropriate file to download. However, if the version register can not be read before downloading the System FPGA, *the user has to specify* version D/E or F for each board, and the hardware version has to be read only after the SYSTEm FPGA is configured.

Table 7.1: Boot procedure for revision-F DGFs.

Action	CAMAC command	Data	Notes
Read hardware version	Read_Version, F(1)A(13)		Result's lower 4 bits contain revision number (D=3,E=4,F=5)

Configure System FPGA	Write_ICSR, F(17)A(8)	0x1	Writing this bit resets FPGA
	Wait at least 50ms	--	
	Write_SysFPGA, F(17)A(10)	Configuration data (166980 bytes)	Use configuration file according to revision found above
Configure Trigger/ Filter FPGA	Write_ICSR, F(17)A(8)	0xF0	Writing these bits resets FPGAs
	Wait at least 50ms	--	
	Write_FipFPGA, F(17)A(9)	Configuration data (166980 bytes)	According to revision found above
Confirm FPGA Downloads	Read_ICSR, F(1)A(8)		If result = 0, all downloads were successful
Boot DSP	Write_CSR, F(17)A(0)	0x10	
	Wait 50ms	--	
	Write_TSAR, F(17)A(1)	1	Set DSP memory address to 1
	Write_Memory, F(16)A(0)	DSPcode[2],DSPcode[3], …,DSPcode[N]	Automatic memory increment
	Write_TSAR, F(17)A(1)	0	Set DSP memory address to 0
	Write_Memory, F(16)A(0)	DSPcode[0],DSPcode[1]	Writing to DSP address 0 reboots DSP

After initialization, the switchbus registers have to be set in order to properly terminate trigger signals. These registers are programmed as part of the "ProgramFippi" control task (step 5 above)

7.3 CAMAC commands

Below follows a list of CAMAC commands to be used by programmers. This list is not exhaustive. The modules also respond to other commands not listed here. Those commands, however, are for XIA use only. Therefore, you must make sure that the modules are not addressed with commands other than those shown in Table 7.2.

Table 7.2: List of CAMAC commands for revision-D and revision-E DGFs. Refer to subsection 7.4 concerning peculiarities of the fast level-1 CAMAC reads.

Command, F,A code	Action	Notes
Write_CSR, F(17)A(0)	Write to CSR	
Read_CSR, F(1)A(0)	Read CSR	
Write_ICSR, F(17)A(8)	Write to ICSR	
Read_ICSR, F(1) A(8)	Read ICSR	
Write_TSAR, F(17)A(1)	Write to TSAR	
Read_TSAR, F(1)A(1)	Read TSAR	Disabled. Normally no need for host reads.
Write_WrdCnt, F(1)A(2)	Write to word count register	Disabled. Normally no need for host writes.
Read_WrdCnt, F(17)A(2)	Read word count register	
Write_Data, F(16)A(0)	Write data to DSP memory	
Read_Data, F(0)A(0)	Read data from DSP memory	
Read_Data_fast, F(5)A(0)	Level-1 fast CAMAC DSP data read	May not be fully tested in initial release of DGF F firmware/software.
Read_Version_Sys, F(1)A(5)	Read version of System FPGA	
Read_Version_Conf, F(1)A(13)	Read version of Config FPGA	
Write_FipFPGA, F(17)A(9)	Write configuration data for Fippi	
Write_SysFPGA, F(17)A(10)	Write configuration data for System	

7.4 Using level-1 fast CAMAC data reads

Fast CAMAC reads are implemented for data reads from DSP memory, and can be applied to reading list mode and histogram data. It is important to note that some CAMAC controllers do not deassert the N-line at the end of the fast CAMAC data transfer. In such a case the red front panel LED will remain lit after the transfer. In fact the controller may deassert the N-line only after the next regular CAMAC command has been completed. Therefore, you have to issue a dummy command to the module, say Read_CSR, to make the controller deassert the N-line.

7.5 Accessing DSP memory

Accessing DSP memory is a two-step process. First you have to write the start address for the data transfer into TSAR. Then you begin a block transfer. The data transfer between the interface FPGA and the DSP is via a DMA channel and does not interrupt the running DSP program, though it may slow it down. Writing to the TSAR transfers the TSAR content to a DMA address register in the DSP. With each read or write the address register in the DSP is

incremented. The TSAR in the interface FPGA, however, remains unaltered. Therefore, if you want to read from the same memory location twice, you have to write the TSAR again.

Secondly, data and program memory of the DSP are organized into different banks with different word length. Data memory is 16-bit wide, and you read one location with each CAMAC cycle. The program memory is 24-bit wide. The DSP uses a 16-bit data bus and has to transfer the 24-bit words in two CAMAC cycles. For writes, you have to write the higher 16 bits of the 24-bit word first, followed by a second write in which the lower 8 bits carry the lower portion of the 24-bit word. For reads, you will receive the higher 16 bits in the first word, and the lower 8 bits of the 24-bit word in the lower 8 bits of the second read.

Setting individual DSP variables in general requires a very good understanding of how the DGF-4C works. However, you may want to be able to change some of the settings using your own host computer. The best strategy is to create an image of the first 256 data words of the DSP memory in your host computer and then download the whole set. Since your change of variables may require a reprogramming of the DGF DACs or the trigger/filter FPGAs you should call the relevant DSP routines. You may also want to make sure you stopped any run in progress. Assuming that the 256 DSP variable values are stored in an array called DSPvalues, the sequence of operations in pseudocode, using the CAMAC commands defined above, is given in the following block.

```
cmd=Read_CSR;
cmd=CLRBIT(0,cmd);              // clear bit 0
Write_CSR(cmd);                 // stop run without overwriting other bits
while(Read_CSR & 0x2000) {;}    // Wait for end of run
Write_TSAR(0x4000);             // start of data memory
Write_Data(DSPvalues,256);      // write 256 words

Address=GetAddress("RUNTASK");
Write_TSAR(Address);
Write_Data(0,1);                // write 1 word, setting RUNTASK to 0

Address=GetAddress("CONTROLTASK");
Write_TSAR(Address);
Write_Data(0,1);                // write 1 word, setting CONTROLTASK to 0

cmd=Read_CSR;
cmd=SETBIT(0,cmd);              // set bit 0 of cmd to 1
Write_CSR(cmd);                 // start run without overwriting other bits
while(Read_CSR & 0x2000) {;}    // wait until DACs are reprogrammed

Address=GetAddress("CONTROLTASK");
Write_TSAR(Address);
Write_Data(5,1);                // write 1 word, setting CONTROLTASK to 5

cmd=Read_CSR;
cmd=SETBIT(0,cmd);              // set bit 0 of Ret to 1
Write_CSR(cmd);                 // start run without overwriting other bits
while(Read_CSR & 0x2000) {;}    // wait until trigger/filter FPGAs are reprogrammed
```

7.6 Data acquisition runs and data buffering

When the DSP receives an event interrupt, it responds by gathering the requested data from the RTPUs. We minimize the dead time associated with the interrupt routine by writing the intermediate data to a buffer and deferring the event-related computations. Those are performed by an event processing routine, which is executed in regular, not interrupt, mode. A global write and a global read pointer control writing to and reading from the data buffer. The interrupt routine updates the write pointer, while the event processing routine increments the read pointer. The read pointer is incremented to point to the next event only after the event computations are finished.

You can do any number of list mode runs in a row. The first run would be started with variable RESUME set to 1. This clears all histograms and run statistcs. Once this run has ended and data has been read out you resume running with variable RESUME set to 0. This keeps the histogram memory intact and you can accumulate spectra over many runs. RESUME is automatically set to 0 at the end of the run by the DSP. The following pseudocode illustrates this.

```
Address=GetAddress("RESUME");
Write_TSAR(Address);
Write_Data(0,1);                    // write 1 word, setting RESUME to 1

cmd=Read_CSR();    // read value of interface CSR
cmd=setbit(0, cmd);  // set bit 0 to 1, ie start a new run
Write_CSR(cmd);     // issue StartRun command

for(k=1; k<Nruns; k++)
{
  while(Read_CSR() AND 0x2000) {; }  // wait until run has ended, use bit-wise AND
  Read data and save to file
  Write_CSR(cmd);  // issue StartRun command (RESUME set to 0 by DSP)
}
```

To stop a run before it finished by itself (filling the data buffer), set RESUME=2 before writing to the CSR with bit 0 cleared.

When a sequence of runs is requested the controller overhead can be kept to a minimum in the following way: Start the runs with the LAM interrupt enabled (bit 4 of the CSR set), even if LAMs are not serviced by the controller. At the end of each run, the DSP writes the buffer start address into its own DMA address register, sets the LAMstate bit (bit 14 of the CSR), and writes the number of data words (NumData) available into the word count register of the CAMAC interface. (NumData is also stored in the first word of the output buffer).

To see if the run is finished, the user code should poll the modules, and check if the LAMstate bit is set. If so, reading the word count register tells the number of words available, and clears the register—it can't be read twice. The read also clears the LAMstate bit.

8 Appendix C — USB interface

The USB interface is used only to read out the external memory of the DGF Rev. F. Booting of the module, setting of parameters, starting/polling/stopping runs use the CAMAC interface with the same commands as previous revisions. Reading out MCA data or list mode data (in "32 buffer per spill" mode) from the external memory uses the USB interface. Single buffer list mode data (from DSP memory) is read out through CAMAC as before.

8.1 Drivers

For a (Windows) PC to communicate with the USB interface of a DGF Rev. F, it needs two driver files:

1. xia_usb2.inf contains the setup information to pick the correct driver file. It links devices with XIA's Vendor ID to the driver file provided by Cypress
2. CyUsb.sys is the system driver file provided by Cypress.

When Windows recognizes a DGF Rev. F plugged into a USB port, it should be pointed to these drivers (located in the "drivers" folder of the XIA software distribution). When drivers are installed correctly, the DGF will appear in Window's device manager as "XIA DGF-4C Spectrometer (Rev. F)"

8.2 DLL functions

Cypress provides a Windows development kit for USB interface development. XIA used this kit to generate a dll library which provides USB I/O functions to the main XIA C library. This dll ("USBdll.dll") has to be copied to the Windows/System32 folder. The DLL defines 4 USB I/O functions, but only 2 (open and read) are used in the C library:

1. xia_usb2_open (int dev, HANDLE *h)
   Opens the device with the specified number (dev) and returns a valid HANDLE to the device or NULL if the device could not be opened.

2. xia_usb2_close (HANDLE h)
   Closes a device handle (h) previously opened via. xia_usb2_open().

3. xia_usb2_read (HANDLE h, unsigned long addr, unsigned long n_bytes, byte_t *buf);
   Reads the specified number of bytes from the specified address into the buffer *buf.

4. xia_usb2_write (HANDLE h, unsigned long addr, unsigned long n_bytes, byte_t *buf);
   Writes the specified number of bytes from the buffer *buf to the specified address.

The USB "address space" is defined as follows

0x0 – beginning of MCA memory (4 blocks of 32K words)
0x20000 - beginning of List mode memory (128K words)
0x10000000 - address of EEPROM storing serial number

While the addresses specify locations of 32bit words in the DGF's external memory, the USB DLL functions require the number of *bytes* as the argument of how much data to read. Any address from 0x0 to 0x3FFFF can be addressed (though rarely necessary), but the EEPROM address is only a "code word" to read the specified number of bytes from the

EEPROM memory, starting EEPROM memory address 0. The EEPROM can hold 16K bytes, currently only the first 2 are used to store the module's serial number.

Thus typical operations are the following:

a) read from address 0x0 128K words (= 512K bytes) to read all spectra,
b) read from address 0x2000 two times the number of words specified in DSP variables EMwords (= number of 16 bit list mode data words acquired), i.e. (2 x EMwords) bytes
c) in double buffer mode, read from address 0x20000 [or 0x30000] two times the number of words specified in DSP variables EMwords [or EMwords2]
d) read from address 0x10000000 2 bytes to obtain the serial number.

The USB chip has an internal FIFO for 512 bytes. It is therefore most efficient to read data in multiples of 512 bytes; reading fewer bytes takes about the same time (or more?) that 512.