Programmer's Manual

Digital Gamma Finder (DGF)

PIXIE-4

Version 1.61, May 2008

XIA LLC

31057 Genstar Road
Hayward, CA 94544 USA

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

Disclaimer

Table of Contents

1 Overview

This manual is divided into three major sections. The first section is a description of the PIXIE-4 application program interface (API). Advanced users can build their own user interface using these API functions. The second section is a reference guide to program the PIXIE-4 modules using the PIXIE-4 API. This will be interesting to those users who want to integrate the PIXIE-4 modules into their own data acquisition system. The third section describes those user accessible variables that control the functions of the PIXIE-4 modules. Those advanced and curious users can use this section to better understand the operation of the PIXIE-4. Additionally, this manual also includes instructions on how to write User DSP code.

2 PIXIE-4 API

The PIXIE-4 API consists of a set of C functions for building various coincidence data acquisition applications. It can be used to configure Pixie-4 modules, make MCA or list mode runs and retrieve data from the Pixie modules. The API can be compiled as a WaveMetrics Igor XOP file which is currently used by the Pixie-4 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 Pixie-4 is given below and the usage of these functions is mentioned wherever appropriate.

At first the PIXIE-4 API needs to be initialized. This is a process in which the names of system configuration files and variables are downloaded to the API. The function **Pixie_Hand_Down_Names** is used to achieve this.

The second step is to boot the Pixie modules. It involves initializing each PXI slot where a Pixie module is installed, 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 (DACs). All this has been encapsulated in a single function **Pixie_Boot_System**.

Now, the instrument is ready for data acquisition. The function used for this purpose is **Pixie_Acquire_Data**. By setting different run types, it can be used to start, stop 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 Pixie modules.

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

Another function, **Pixie_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 Pixie

modules by looking at their internal memory values. The other usage of this function is to read, save, copy or extract Pixie's configurations though its settings files.

In a multi-module Pixie-4 system, it is essential for the host to know which module or channel it is communicating to. The function **Pixie_Set_Current_ModChan** is used to set the current module and channel.

The detailed description of each function is given below.

Pixie_Hand_Down_Names

Syntax

```
S32 Pixie_Hand_Down_Names (
  U8 *Names[],       // An array containing the names to be downloaded
  U8 *Name);         // A string indicating the type of names (file or
                     // variable names) to be downloaded
```

Description

Use this function to download the file or variable names from the host user interface to the Pixie-4 API. The API needs these file names so that it can read the Pixie hardware configurations from the files stored in the host computer and download these configurations to the Pixie. The variable names are used by the API to obtain the indices of the DSP variables when the API 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. The API will know which type of names is being downloaded by checking the other parameter *Name*:

1. **ALL_FILES**: This indicates we are downloading boot files names. In this case, *Names* is a string array which has N_BOOT_FILES elements. Currently N_BOOT_FILES is defined as 7. The elements of *Names* are the names of communication FPGA files (Revision B and Revision C, respectively), signal processing FPGA file, DSP executable code binary file, DSP I/O parameter values file, DSP code I/O variable names file, and DSP code memory variable names file. All file names should contain the complete path name.

2. **SYSTEM**: This indicates we are downloading System_Parameter_Names. System_Parameter_Names are those global variables that are applicable to all modules in a Pixie system, e.g. number of Pixie modules in the chassis, etc. System_Parameter_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 System_Parameter_Names is given in Table 3.5.

3. **MODULE**: This indicates we are downloading Module_Parameter_Names. Module_Parameter_Names are those global variables that are applicable to each individual module, e.g. module number, module CSR, coincidence pattern, and run type, etc. Module_Parameter_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 Module_Parameter_Names is given in Table 3.5.

4. **CHANNEL**: This indicates we are downloading Channel_Parameter_Names. Channel_Parameter_Names are those global variables that are applicable to individual channels of the Pixie modules, e.g. channel CSR, filter rise time, filter flat top, voltage gain, and DC offset, etc. Channel_Parameter_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 Channel_Parameter_Names is given in Table 3.5.

Return values

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

Usage example

```
S32 retval;

// download system parameter names; define System_Parameter_Names first
retval = Pixie_Hand_Down_Names(System_Parameter_Names, "SYSTEM");
if(retval < 0)
{
// error handling
}

// download module parameter names; define Module_Parameter_Names first
retval = Pixie_Hand_Down_Names(Module_Parameter_Names, "MODULE");
if(retval < 0)
{
// error handling
}

// download channel parameter names; define Channel_Parameter_Names
// first
retval = Pixie_Hand_Down_Names(Channel_Parameter_Names, "CHANNEL");
if(retval < 0)
{
// error handling
}

// download boot file names; define All_Files first
retval = Pixie_Hand_Down_Names(All_Files, "ALL_FILES");
if(retval < 0)
{
// error handling
}
```

Pixie_Boot_System

Syntax

```
S32 Pixie_Boot_System (
  U16 Boot_Pattern);   // The Pixie-4 boot pattern
```

Description

Use this function to boot all Pixie modules in the system. Before booting the modules, it scans all PXI crate slots and finds the address for each slot where a Pixie module is installed.

Parameter description

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

Bit 0: Boot communication FPGA
Bit 1: Boot signal processing FPGA
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 Pixie modules, i.e. the *Boot_Pattern* should be 0x1F.

Return values

Value	Description	Error Handling
0	Success	None
-1	Unable to scan crate slots	Check PXI slot map
-2	Unable to read communication FPGA configuration (Rev. B)	Check comFPGA file
-3	Unable to read communication FPGA configuration (Rev. C)	Check comFPGA file
-4	Unable to read signal processing FPGA configuration	Check SPFPGA file
-5	Unable to read DSP executable code	Check DSP code file
-6	Unable to read DSP parameter values	Check DSP parameter file
-7	Unable to initialize DSP parameter names	Check DSP .var file
-8	Failed to boot all modules present in the system	Check Pixie modules

Usage example

```
S32 intval;

retval = Pixie_Boot_System(0x1F);
if(ret < 0)
{
// error handling
}
```

Pixie_User_Par_IO

Syntax

```
S32 Pixie_User_Par_IO (
  double *User_Par_Values,     // A double precision array containing the
                               // user parameters to be transferred
  U8 *User_Par_Name,           // A string variable indicating which user
                               // parameter is being transferred
  U8 *User_Par_Type,           // A string variable indicating which type
                               // of user parameters is being transferred
  U16 Direction,               // I/O direction (read or write)
  U8 ModNum,                   // Number of the module to work on
  U8 ChaNum);                  // Channel number of the Pixie module
```

Description

Use this function to transfer user parameters between the user interface, the API and DSP's I/O memory. Some of these parameters (`User_Par_Type = "SYSTEM"`) are applicable to all Pixie modules in the system, like the total number of Pixie modules in the system. Other parameters (`User_Par_Type = "MODULE"`) are applicable to a whole Pixie module (independent of its four channels), e.g. coincidence pattern, Module CSRA, etc. The final set of parameters (`User_Par_Type = "CHANNEL"`) are applicable to each individual channel in a Pixie module, e.g. energy filter settings or voltage gain, etc. 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 API), this function will call another function **UA_PAR_IO** which first converts these parameters into numbers that are recognized by both the DSP and the API 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_Type*, 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_Type* is shown in Table 2.1.

Table 2.1: The Combination of User_Par_Name and User_Par_Values.

User_Par_Type	User_Par_Values		
	Name	Size	Data Type
SYSTEM	System_Parameter_Values	64	Double precision
MODULE	Module_Parameter_Values	64×7	Double precision
CHANNEL	Channel_Parameter_Values	64×7×4	Double precision

The way to fill the *Channel_Parameter_Values* array is to fill the channel first then the module. For instance, first 64 values are stored in the array for channel 0, and then repeat this for other three channels. After that, 64×4 values have been filled for module 0. Then repeat this for the remaining modules. For the *Module_Parameter_Values* array, first store 64 values for module 0, and then repeat this for the other modules.

User_Par_Name is the name of the variable which is to be transferred. It is one element of either System_Parameter_Names, or Module_Parameter_Names, or Channel_Parameter_Names.

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

ModNum is the number of the Pixie module being communicated to.

ChanNum is the channel number of the Pixie module being communicated to.

Return values

Value	Description	Error Handling
0	Success	None
-1	Null pointer for User_Par_Values	Check User_Par_Values
-2	Invalid user parameter name	Check User_Par_Name
-3	Invalid user parameter type	Check User_Par_Type
-4	Invalid I/O direction	Check direction
-5	Invalid Pixie module number	Check ModNum
-6	Invalid Pixie channel number	Check ChanNum

Usage example

```
U16 direction, modnum, channum;
S32 retval;

direction = 0;   // download
modnum = 0;      // Module #0
channum = 1;     // Channel #1

// set module parameter COINCIDENCE_PATTERN to 0xFFFF
Module_Parameter_Values[Coincidence_Pattern_Index]=0xFFFF;
// download COINCIDENCE_PATTERN to the DSP
retval = Pixie_User_Par_IO(Module_Parameter_Values,
          "COINCIDENCE_PATTERN", "MODULE", direction, modnum, channum);
if(retval < 0)
{
// error handling
}

// set channel parameter ENERGY_RISETIME to 6.0 µs
```

```
Channel_Parameter_Values[ENERGY_RISETIME_Index]=6.0;
// download ENERGY_RISETIME to DSP
retval = Pixie_User_Par_IO(Channel_Parameter_Values, "ENERGY_RISETIME",
                           "CHANNEL", direction, modnum, channum);
if(retval < 0)
{
// error handling
}
```

Pixie_Acquire_Data

Syntax

```
S32 Pixie_Acquire_Data (
  U16 Run_Type,      // Data acquisition run type
  U32 *User_data,    // An unsigned 32-bit integer array containing the
                     // data to be transferred
  U8 *file_name,     // Name of the file used to store list mode or MCA
                     // histogram data
  U8 ModNum);        // The number of the Pixie module
```

Description

Use this function to acquire ADC traces, MCA histogram, 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

Upper 4-bit:

0x0000	start a control task run
0x1000	start a new data run
0x2000	resume a data run
0x3000	stop a data run
0x4000	poll run status
0x5000	read histogram data and save it to a file
0x6000	read list mode buffer data and save it to a file
0x7000	offline list mode data parse routines
0x7001	parse list mode data file
0x7002	locate traces

0x7003	read traces
0x7004	read energies
0x7005	read PSA values
0x7006	read extended PSA values
0x7007	locate events
0x7008	read event
0x8000	manually read MCA histogram from a MCA file
0x9000	external memory (EM) I/O
0x9001	read histogram memory section of EM
0x9002	write to histogram memory section of EM
0x9003	read list mode memory section of EM
0x9004	write to list mode memory section of EM

User_data has the following format for the run types listed below:

0x4: Get ADC traces

Length must be ADCTraceLen*NumberOfChannels, i.e. 8192 * 4 = 32k.
All array elements are return values.
the Nth 8k of data are the ADC trace of channel N.

0x7001: Parse list mode data file

Length must be 2* MaxNumModules, i.e. 16 or 34
All array elements are return values.
User_data[i] = NumEvents of module i
User_data[i+MaxNumModules] = TotalTraces of module I

0x7002: Locate Traces of all events

Length must be (TotalTraces of ModNum)*3*NumberOfChannels
All array elements are return values.
User_data[i*3n] = Location of channel n's trace in file for event i (word number)
User_data[i*3n+1] = length of channel n's trace
User_data[i*3n+2] = energy for channel n

0x7003: Read Traces of one event

Length must be (NumberOfChannels*2+combined tracelength of channels)
First (NumberOfChannels*2) elements are input values:
User_data[2n] = Location of channel n's data in file for selected event (word number)
User_data[2n+1] = length of channel n's trace

The remaining array elements are return values.
User_data[8 …] = Trace data of channel 0 followed by channels 1,2, and 3.

0x7004: Read Energies of all events

Length must be (NumEvents of ModNum * NumberOfChannels)
All array elements are return values.
User_data[i*4+n] = energy of channel n for event i

0x7005: Read PSA values of all events

Length must be (NumEvents of ModNum*2 * NumberOfChannels)
All array elements are return values.
User_data[i*2n] = XIAPSA word of channel n for event i
User_data[i*2n+1] = UserPSA word of channel n for event i

0x7006: Read extended PSA values of all events

Length must be (NumEvents of ModNum*8 * NumberOfChannels)
All array elements are return values.
User_data[i*8n] = timestamp word of channel n for event i
User_data[i*8n+1] = energy word of channel n for event i
User_data[i*8n+2] = XIAPSA word of channel n for event i
User_data[i*8n+3] = UserPSA word of channel n for event i
User_data[i*8n+4] = Unused1 word of channel n for event i
User_data[i*8n+5] = Unused2 word of channel n for event i
User_data[i*8n+6] = Unused3 word of channel n for event i
User_data[i*8n+7] = RealTimeHi word of channel n for event i

0x7007: Locate all events

Length must be (NumEvents of ModNum)*3
All array elements are return values.
User_data[i*3] = Location of event i in file (word number)
User_data[i*3+1] = Location of buffer header start for event i in file
User_data[i*3+2] = Length of event i (event header, channel header, traces)

0x7008: Read one event

Length must be (length of selected event) + 7 +36
(this is longer than actually used, but ensures enough room for channel headers in all runtypes)
First 3 elements are input values:
User_data[0] = Location of selected event in file (word number)
User_data[1] = Location of buffer header start for selected event in file
User_data[2] = Length of selected event

The remaining array elements are return values.
User_data[3 …6] are the tracelengths of channel 0-3
User_data[7 …6+BHL] contain the buffer header corresponding to the selected event
User_data[7+BHL .. 6+BHL+EHL] contain the event header
User_data[7+BHL+ELH .. 6+BHL+EHL+4*CHL] are the channel headers for channel 0-3; always 9 words per channel header, but in compressed runtypes some entries are be invalid
User_data[7+BHL+EHL+4*CHL …] contain the traces of channel 0-3, followed by some undefined values (use tracelength to parse traces)

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

ModNum is the number of the module addressed, counting from 0 to (number of modules - 1). If *ModNum* == (number of modules), all modules are addressed in a for loop, however this option is not valid for all RunTypes.

Return values

Return values depend on the run type:

Run type = 0x0000

Value	Description	Error Handling

0	Success	None
-0x1	Invalid Pixie module number	Check ModNum
-0x2	Failure to adjust offsets	Reboot the module
-0x3	Failure to acquire ADC traces	Reboot the module
-0x4	Failure to start the control task run	Reboot the module

Run type = 0x1000

Value	Description	Error Handling
0x10	Success	None
-0x11	Invalid Pixie module number	Check ModNum
-0x12	Failure to start the data run	Reboot the module

Run type = 0x2000

Value	Description	Error Handling
0x20	Success	None
-0x21	Invalid Pixie module number	Check ModNum
-0x22	Failure to resume the data run	Reboot the module

Run type = 0x3000

Value	Description	Error Handling
0x30	Success	None
-0x31	Invalid Pixie module number	Check ModNum
-0x32	Failure to end the run	Reboot the module

Run type = 0x4000

Value	Description	Error Handling
0	No run is in progress	N/A
1	Run is in progress	N/A
CSR value	When run type = 0x40FF	N/A
-0x41	Invalid Pixie module number	Check ModNum

Run type = 0x5000

Value	Description	Error Handling
0x50	Success	None
-0x51	Failure to save histogram data to a file	Check the file name

Run type = 0x6000

Value	Description	Error Handling
0x60	Success	None
-0x61	Failure to save list mode data to a file	Check the file name

Run type = 0x7000

Value	Description	Error Handling

0x70	Success	None
-0x71	Failure to parse the list mode data file	Check list mode data file
-0x72	Failure to locate list mode traces	Check list mode data file
-0x73	Failure to read list mode traces	Check list mode data file
-0x74	Failure to read event energies	Check list mode data file
-0x75	Failure to read PSA values	Check list mode data file
-0x76	Failure to read extended PSA values	Check list mode data file
-0x77	Failure to locate events	Check list mode data file
-0x78	Failure to read events	Check list mode data file
-0x79	Invalid list mode parse analysis request	Check run type

Run type = 0x8000

Value	Description	Error Handling
0x80	Success	None
-0x81	Failure to read out MCA spectrum from the file	Check the MCA data file

Run type = 0x9000

Value	Description	Error Handling
0x90	Success	None
-0x91	Failure to read out MCA section of external memory	Reboot the module
-0x92	Failure to write to MCA section of external memory	Reboot the module
-0x93	Failure to read out LM section of external memory	Reboot the module
-0x94	Failure to write to LM section of external memory	Reboot the module
-0x95	Invalid external memory I/O request	Check the run type

Usage example

```
S32 retval;
U16 RunType;
U32 dummy[2];
U8 ModNum;

RunType = 0x1100;      // start a new list mode run
ModNum = 0;
retval = Pixie_Acquire_Data(RunType, dummy, " ", ModNum);
if(retval != 0x10)
{
// Error handling
}

// wait until the run has ended
RunType = 0x4100;
while( ! Pixie_Acquire_Data(RunType, dummy, " ", ModNum) ) {;}

// Read out the list mode data from all Pixie modules and save to a file
RunType = 0x6100;
retval = Pixie_Acquire_Data(RunType, dummy,
```

```
                “C:\XIA\Pixie4\PulseShape\Listdata0001.bin”, ModNum);
if(retval != 0x60)
{
// Error handling
}

// Read out the histogram data from all Pixie modules and save to a file
RunType = 0x5100;
retval = Pixie_Acquire_Data(RunType, dummy,
                “C:\XIA\Pixie4\MCA\Histdata0001.bin”, ModNum);
if(retval != 0x50)
{
// Error handling
}
```

Pixie_Set_Current_ModChan

Syntax

```
S32 Pixie_Set_Current_ModChan (
  U8 Module,       // Module number to be set
  U8 Channel);     // Channel number to be set
```

Description

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

Parameter description

Module specifies the current module to be set. Module should be in the range of 0 to MAX_NUMBER_OF_MODULES (currently MAX_NUMBER_OF_MODULES is set to 7).

Channel specifies the current channel to be set. Channel should be in the range of 0 to NUMBER_OF_CHANNELS - 1 (currently NUMBER_OF_CHANNELS is set to 4).

Return values

Value	Description	Error Handling
0	Success	None
-1	Invalid module number	Check Module
-2	Invalid channel number	Check Channel

Usage example

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

Pixie_Buffer_IO

Syntax

```
S32 Pixie_Buffer_IO (
  U16 *Values,     // An unsigned 16-bit integer array containing the
                   // data to be transferred
  U8 type,         // Data transfer type
  U8 direction,    // Data transfer direction
  U8 *file_name,   // File name
  U8 ModNum);      // Module number
```

Description

Use this function to:

1) Download or upload DSP parameters between the user interface and the Pixie 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 the selected modules.

Parameter description

Values is an unsigned 16-bit integer array used for data transfer between the user interface and Pixie 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 Pixie_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 in the system
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
-1	Failure to set DACs after writing DSP parameters	Reboot the module
-2	Failure to program Fippi after writing DSP parameters	Reboot the module
-3	Failure to set DACs after loading DSP parameters	Reboot the module
-4	Failure to program Fippi after loading DSP parameters	Reboot the module
-5	Can't open settings file for loading	Check the file name
-6	Can't open settings file for reading	Check the file name
-7	Can't open settings file to extract settings	Check the file name
-8	Failure to set DACs after copying or extracting settings	Reboot the module
-9	Failure to program Fippi after copying or extracting settings	Reboot the module
-10	Invalid module number	Check ModNum
-11	Invalid I/O direction	Check direction
-12	Invalid I/O type	Check type

Usage example

```
S32 retval;
U8 type, direction, modnum;

modnum = 0;        // Module number

// Download DSP parameters to the current Pixie module; DSP_Values is a
// pointer pointing to the DSP parameters; no need to specify file name
// here.
direction = 0;     // Write
type = 0;          // DSP I/O values
retval = Pixie_Buffer_IO(DSP_Values, type, direction, "", modnum);
if(retval < 0)
{
     // Error handling
}

// Read DSP memory values from the current PIXIE module; Memory_Values
// is a pointer pointing to the memory block; no need to specify file
// name Here.
direction = 1;     // Read
type = 1;          // DSP memory values
retval = Pixie_Buffer_IO(Memory_Values, type, direction, "", modnum);
if(retval < 0)
```

```
{
// Error handling
}
```

Options for Compiling PIXIE-4 API

Pixie-4 API can be compiled as either a WaveMetrics Igor XOP file which is currently used by the Pixie-4 Viewer, a dynamic link library (DLL) or static library. The two latter options can be used by advanced users to integrate Pixie modules into their own data acquisition systems.

The following table summarizes the required files for these options.

Table 2.3: Options for compiling the PIXIE-4 C Driver.

Compilation Option	Required Files		
	C source files	C header files	Library files
a dynamic link library (DLL) or static library	Boot.c, eeprom.c, pixie_c.c, utilities.c	boot.h, defs.h, globals.h, sharedfiles.h, utilities.h, PciApi.h, PciRegs.h, Plx.h, PlxApi.h, PlxDefinitionsCheck.h, PlxError.h, PlxTypes.h, Reg9054.h	PlxApi.lib, PlxApi.dll
Igor XOP	Boot.c, eeprom.c, pixie_c.c, utilities.c, pixie4_iface.c, pixie4_igor.c, PixieWinCustom.rc	boot.h, defs.h, globals.h, pixie4_iface.h, sharedfiles.h, utilities.h, PciApi.h, PciRegs.h, Plx.h, PlxApi.h, PlxDefinitionsCheck.h, PlxError.h, PlxTypes.h, Reg9054.h	PlxApi.lib, PlxApi.dll

The Igor XOP option also needs the following files in the Igor XOP Library provided by WaveMetrics.

IgorXOP.h, VCExtraIncludes.h, Xop.h, XOPResources.h, XOPStandardHeaders.h, XOPSupport.h, XOPSupportWin.h, XOPWinMacSupport.h, XOPSupport x86.lib, and IGOR.lib.

3 Control PIXIE-4 Modules via CompactPCI

3.1 Initializing

We describe here how to initialize PIXIE-4 modules in a PXI chassis using the functions described in Section 2. As an example, we assume two PIXIE-4 modules – resided in slot #3 and #4, respectively. Users are also encouraged to read the sample C code shipped with the API.

3.1.1 Initialize Global Variables

As discussed in Section 2, we assume that three global variable arrays have been defined: System_Parameter_Values, Module_Parameter_Values and Channel_Parameter_Values. For these three global variable arrays, we also need to define three global name arrays: System_Prameter_Names, Module_Parameter_Names and Channel_Parameter_Names, respectively. Table 3.1 lists the names contained in each of these name arrays. The order of placing these names into the name array is not important since the API uses search functions to locate each name at run time.

Table 3.1: Contents of Global Name Arrays.

Array	Names
System_Parameter_Names	NUMBER_MODULES, OFFLINE_ANALYSIS, C_LIBRARY_RELEASE, C_LIBRARY_BUILD, SLOT_WAVE
Module_Parameter_Names	MODULE_NUMBER, MODULE_CSRA, MODULE_CSRB, MODULE_FORMAT, MAX_EVENTS, COINCIDENCE_PATTERN, ACTUAL_COINCIDENCE_WAIT, MIN_COINCIDENCE_WAIT, SYNCH_WAIT, IN_SYNCH, RUN_TYPE, FILTER_RANGE, BUFFER_HEAD_LENGTH, EVENT_HEAD_LENGTH, CHANNEL_HEAD_LENGTH, OUTPUT_BUFFER_LENGTH, NUMBER_EVENTS, RUN_TIME, BOARD_VERSION, SERIAL_NUMBER
Channel_Parameter_Names	CHANNEL_CSRA, CHANNEL_CSRB, ENERGY_RISETIME, ENERGY_FLATTOP, TRIGGER_RISETIME, TRIGGER_FLATTOP, TRIGGER_THRESHOLD, VGAIN, VOFFSET, TRACE_LENGTH, TRACE_DELAY, PSA_START, PSA_END, EMIN, BINFACTOR, TAU, BLCUT, XDT, BASELINE_PERCENT, CFD_THRESHOLD, INTEGRATOR, LIVE_TIME, INPUT_COUNT_RATE, FAST_PEAKS

Additionally, a string array All_Files containing the file names for the initialization is also needed. Table 3.2 lists the file names needed to initialize the PIXIE-4 modules.

Table 3.2: File Names in All_Files.

All_Files	File Name	Note
All_Files[0]	C:\XIA\PIXIE4\Firmware\syspixie_revB.bin	Communication FPGA configurations (Rev. B)
All_Files[1]	C:\XIA\PIXIE4\Firmware\syspixie_revC.bin	Communication FPGA configurations (Rev. C)
All_Files[2]	C:\XIA\PIXIE4\Firmware\pixie.bin	Signal processing FPGA configurations
All_Files[3]	C:\XIA\PIXIE4\DSP\PXIcode.bin	DSP executable binary code
All_Files[4]	C:\XIA\PIXIE4\Configuration\default.set	Settings file
All_Files[5]	C:\XIA\PIXIE4\DSP\PXIcode.var	File of DSP I/O variable names
All_Files[6]	C:\XIA\PIXIE4\DSP\PXIcode.lst	File of DSP memory variable names

The global variable array, System_Parameter_Values, also needs to be initialized before the API functions are called to start the initialization. Table 3.3 lists those global variables.

Table 3.3: Initialization of Module_Global_Values.

Module_Global_Names	Module_Global_Values	Note
NUMBER_MODULES	2	The total number of PIXIE-4 modules
SLOT_WAVE[0]	3	Module 0 sits in slot 3
SLOT_WAVE[1]	4	Module 1 sits in slot 4

3.1.2 Boot PIXIE Modules

The boot procedure for PIXIE-4 modules includes the following steps. First, all the global parameter names and boot file names should be downloaded by calling **Pixie_Hand_Down_Names**. Then function **Pixie_User_Par_IO** should be called to initialize the global value array System_Parameter_Values. Finally, function **Pixie_Boot_System** should be called to boot the modules. The following code is an example showing how to boot the PIXIE-4 modules using the API functions.

An Example Code Illustrating How to Boot PIXIE-4 Modules

```
S32 retval;
U8 direction, modnum, channum;

// initialize system parameter values
System_Parameter_Values[NUMBER_MODULES_Index] = 2;
System_Parameter_Values[OFFLINE_ANALYSIS_Index] = 0;
System_Parameter_Values[SLOT_WAVE_Index] = 3;
System_Parameter_Values[SLOT_WAVE_Index+1] = 4;

retval = Pixie_Hand_Down_Names(System_Parameter_Names, "SYSTEM");
if( retval < 0 )
{
  // Error handling
}

retval = Pixie_Hand_Down_Names(Module_Parameter_Names, "MODULE");
if( retval < 0 )
```

```
{
  // Error handling
}

retval = Pixie_Hand_Down_Names(Channel_Parameter_Names, "CHANNEL");
if( retval < 0 )
{
  // Error handling
}

retval = Pixie_Hand_Down_Names(All_Files, "ALL_FILES");
if( retval < 0 )
{
  // Error handling
}

direction = 0;   // download
modnum = 0;      // Module #0
channum = 0;     // Channel #0

retval = Pixie_User_Par_IO(System_Parameter_Values, "NUMBER_MODULES",
           "SYSTEM", direction, modnum, channum);
if( retval < 0 )
{
  // Error handling
}

retval = Pixie_User_Par_IO(System_Parameter_Values,
           "OFFLINE_ANALYSIS", "SYSTEM", direction, modnum, channum);
if( retval < 0 )
{
  // Error handling
}

retval = Pixie_User_Par_IO(System_Parameter_Values, "SLOT_WAVE",
             "SYSTEM", direction, modnum, channum);
if( retval < 0 )
{
  // Error handling
}

// boot PIXIE-4 modules
retval = Pixie_Boot_System(0x1F);
if( retval < 0 )
{
  // Error handling
}

// set current module and channel number
Pixie_Set_Current_ModChan(0, 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 PXIcode.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 PIXIE-4 API. So instead of directly setting DSP variables, users only need to set the values of those global variables defined in Table 3.1. The API 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 API will first convert these DSP values into the global variable values. The code shown below is an example of setting DSP variables through the API.

An Example Code Illustrating How to Set DSP Variables through the API

```
S32 retval;
U8 direction, modnum, channum;

direction = 0;   // download
modnum = 0;      // Module #0
channum = 0;     // Channel #0

// set COINCIDENCE_PATTERN to 0xFFFF
Module_Parameter_Values[COINCIDENCE_PATTERN_Index] = 0xFFFF;

// download COINCIDENCE_PATTERN to the DSP
retval = Pixie_User_Par_IO(Module_Parameter_Values,
        "COINCIDENCE_PATTERN", "MODULE", direction, modnum, channum);
if( retval < 0 )
{
  // Error handling
}

// set ENERGY_RISETIME to 6.0 µs
Channel_Parameter_Values[ENERGY_RISETIME_Index] = 6.0;

// download ENERGY_RISETIME to the DSP
retval = Pixie_User_Par_IO(Channel_Parameter_Values,
```

```
                "ENERGY_RISETIME", "CHANNEL", direction, modnum, channum);
if( retval < 0 )
{
  // Error handling
}
```

Table 3.5 gives a complete description of all the global variables being used by the PIXIE-4 API.

Table 3.5: Descriptions of Global Variables in PIXIE-4.

System_Parameter_Names	**I/O Type**	**Unit**	**Corresponding DSP Variables**
NUMBER_MODULES	Read/Write	N/A	N/A
OFFLINE_ANALYSIS	Read/Write	N/A	N/A
C_LIBRARY_RELEASE	Read only	N/A	N/A
C_LIBRARY_BUILD	Read only	N/A	N/A
SLOT_WAVE	Read/Write	N/A	N/A
Module_Parameter_Names	**I/O Type**	**Unit**	**Corresponding DSP Variables**
MODULE_NUMBER	Read only	N/A	MODNUM
MODULE_CSRA	Read/Write	N/A	MODCSRA
MODULE_CSRB	Read/Write	N/A	MODCSRB
MODULE_FORMAT	Read/Write	N/A	MODFORMAT
MAX_EVENTS	Read/Write	N/A	MAXEVENTS
COINCIDENCE_PATTERN	Read/Write	N/A	COINCPATTERN
ACTUAL_COINCIDENCE_WAIT	Read/Write	N/A	COINCWAIT
MIN_COINCIDENCE_WAIT	Read only	N/A	COINCWAIT
SYNCH_WAIT	Read/Write	N/A	SYNCHWAIT
IN_SYNCH	Read/Write	N/A	INSYNCH
RUN_TYPE	Write only	N/A	RUNTASK
FILTER_RANGE	Read/Write	N/A	FILTERRANGE
BUFFER_HEAD_LENGTH	Read only	N/A	BUFHEADLEN
EVENT_HEAD_LENGTH	Read only	N/A	EVENTHEADLEN
CHANNEL_HEAD_LENGTH	Read only	N/A	CHANHEADLEN
OUTPUT_BUFFER_LENGTH	Read only	N/A	LOUTBUFFER
NUMBER_EVENTS	Read only	N/A	NUMEVENTSA, NUMEVENTSB
RUN_TIME	Read only	s	RUNTIMEA, RUNTIMEB, RUNTIMEC
BOARD_VERSION	Read only	N/A	N/A
SERIAL_NUMBER	Read only	N/A	N/A
Channel_Parameter_Names	**I/O Type**	**Unit**	**Corresponding DSP Variables**
CHANNEL_CSRA	Read/Write	N/A	CHANCSRA
CHANNEL_CSRB	Read/Write	N/A	CHANCSRB
ENERGY_RISETIME	Read/Write	μs	SLOWLENGTH
ENERGY_FLATTOP	Read/Write	μs	SLOWGAP
TRIGGER_RISETIME	Read/Write	μs	FASTLENGTH
TRIGGER_FLATTOP	Read/Write	μs	FASTGAP
TRIGGER_THRESHOLD	Read/Write	N/A	FASTTHRESH
VGAIN	Read/Write	V/V	GAINDAC

VOFFSET	Read/Write	V	TRACKDAC
TRACE_LENGTH	Read/Write	µs	TRACELENGTH
TRACE_DELAY	Read/Write	µs	TRIGGERDELAY
PSA_START	Read/Write	µs	PSAOFFSET
PSA_END	Read/Write	µs	PSALENGTH
EMIN	Read/Write	N/A	ENERGYLOW
BINFACTOR	Read/Write	N/A	LOG2EBIN
TAU	Read/Write	µs	PREAMPTAUA, PREAMPTAUB
BLCUT	Read/Write	N/A	BLCUT
XDT	Read/Write	N/A	XWAIT
BASELINE_PERCENT	Read/Write	N/A	BASELINEPERCENT
CFD_THRESHOLD	Read/Write	N/A	CFDTHR
INTEGRATOR	Read/Write	N/A	FTPWIDTH
LIVE_TIME	Read only	s	LIVETIMEA, LIVETIMEB, LIVETIMEC
INPUT_COUNT_RATE	Read only	cps	FASTPEAKSA, FASTPEAKSB, FASTPEAKSC, LIVETIMEA, LIVETIMEB, LIVETIMEC
FAST_PEAKS	Read only	N/A	FASTPEAKSA, FASTPEAKSB, FASTPEAKSC

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. Therefore, the starting address of the MCA spectrum in the external memory for Channel #0, 1, 2 and 3 will be 0x00000000, 0x000080000, 0x00010000, 0x00018000, respectively. The reading-out of the spectrum memory to the host is through the PCI burst read at rates over 100 Mbytes/s. The spectrum memory is accessible even when a data acquisition run is in progress. The following code is an example of how to start a MCA run and read out the MCA spectrum after the run is finished.

An Example Code Illustrating How to Access MCA Spectrum Memory

```
S32 retval;
U8 direction, modnum, channum;
U32 User_Data[131072]; // an array for holding the MCA spectrum data of
                       // 4 channels

direction = 0;   // download
modnum = 0;      // Module #0
channum = 0;     // Channel #0

// start a MCA run
retval = Pixie_Acquire_Data(0x1301, User_Data, " ", modnum);
if( retval < 0 )
{
  // Error handling
```

```
}

// wait for 30 seconds
Sleep(30000);

// stop the MCA run
retval = Pixie_Acquire_Data(0x3301, User_Data, " ", modnum);
if( retval < 0 )
{
  // Error handling
}

// save MCA spectrum to a file
retval = Pixie_Acquire_Data(0x5301, User_Data,
            "C:\\XIA\\Pixie4\\MCA\Data0001.bin", modnum);
if( retval < 0 )
{
  // Error handling
}

// read out the MCA spectrum and put it to array User_data
retval = Pixie_Acquire_Data(0x9001, User_data, " ", modnum);
if( retval < 0 )
{
  // Error handling
}
```

Note that in clover addback mode, the spectrum length is fixed to 16K for each channel plus 16K of addback spectrum. Therefore, the starting address of the MCA spectrum in the external memory for Channel #0, 1, 2 and 3 will be 0x00000000, 0x000040000, 0x00008000, 0x00010000, respectively, for the addback spectrum it is 0x00018000.

3.3.2 Access list mode data

The list mode data in the linear output data buffer can be written in a number of formats. User code should access the three variables BUFHEADLEN, EVENTHEADLEN, and CHANHEADLEN in the configuration file of a particular run to navigate through the data set.

There are two data buffers to choose from: the DSP's local I/O buffer (8K 16-bit words), and section of the external memory (128K 32-bit words). The target data buffer is selected by setting bit 1 in the variable MODCSRA.

If the external buffer is chosen to hold the output data, the local buffer is transferred to the external memory when it has been filled. Then the run resumes automatically, without interference from the host, until 32 local buffers have been transferred. The data can then be read from external memory in a fast block read starting from location 0x00020000.

If the local buffer is chosen, the run stops when the local buffer is filled. The data has to be read out from local memory.

With any data buffer, you can do any number of runs in a row. The first run would be started as a NEW run. This clears all histograms and run statistics in the memory. Once the data has been read out, you can RESUME running. Each RESUME run will acquire another either 32 or 1 8K buffers of data, depending on which buffer has been chosen. In a RESUME run the histogram memory is kept intact and you can accumulate spectra over many runs. The example code shown below illustrates this.

An Example Code Illustrating How to Access List Mode Data

```
S32 retval;
U8 direction, modnum, channum;
U32 User_Data[131072]; // an array for holding the MCA spectrum data of
                       // 4 channels
U16 k, Nruns;
char *DataFile = {"C:\\XIA\\PIXIE4\\PulseShape\\Data.bin"};

direction = 0;   // download
modnum = 0;      // Module #0
channum = 0;     // Channel #0
Nruns = 10;      // 10 repeated list mode runs
k = 0;           // initialize counter

// start a general list mode run
retval = Pixie_Acquire_Data(0x1100, User_Data, " ", modnum);
if( retval < 0 )
{
  // Error handling
}

do
{
  // wait until run has ended
  while( ! Pixie_Acquire_Data(0x4100, User_Data, " ", modnum) ) {;}

  // read out the list mode data and save it to a file
  retval = Pixie_Acquire_Data(0x6100, User_Data, DataFile, modnum);
  if( retval < 0 )
  {
    // Error handling
  }

  k ++;
  if(k > Nruns)
  {
    break;
  }

  // issue RESUME RUN command
  retval = Pixie_Acquire_Data(0x2100, User_Data, " ", modnum);
  if( retval < 0 )
  {
    // Error handling
  }
```

```
}while(1);

// read out the MCA spectrum and put it to array User_Data
retval = Pixie_Acquire_Data(0x9001, User_Data, " ", modnum);
if( retval < 0 )
{
  // Error handling
}
```

To process the list mode data after it is saved to a file, the PIXIE-4 API provides several utility routines to parse the list mode data and read out the waveform, energy of each individual trace or PSA values. The code below shows how to read waveforms from a list mode file.

An Example Code Illustrating How to Parse List Mode Data

```
S32 retval;
U8 direction, modnum, channum, i;
U32 List_Data[2* MAX_NUMBER_OF_MODULES]; // list mode trace information
char *DataFile = {"C:\\XIA\\PIXIE4\\PulseShape\\Data.bin"};
U32 totaltraces;  // total number of traces in the list mode data file
U32 *traceposlen; // point to positions of the traces in the file
U16 *Trace0;      // point to the first trace in the file

direction = 0;  // download
modnum = 0;     // Module #0
channum = 0;    // Channel #0

// start a general list mode run
retval = Pixie_Acquire_Data(0x1100, List_Data, " ", modnum);
if( retval < 0 )
{
  // Error handling
}

// wait until run has ended
while( ! Pixie_Acquire_Data(0x4100, List_Data, " ", modnum) ) {;}

// read out the list mode data and save it to a file
retval = Pixie_Acquire_Data(0x6100, List_Data, DataFile, modnum);
if( retval < 0 )
{
  // Error handling
}

// parse list mode file
retval = Pixie_Acquire_Data(0x7001, List_Data, DataFile, modnum);
if( retval < 0 )
{
  // Error handling
}

totaltraces = 0;
```

```
for(i=0; i<MAX_NUMBER_OF_MODULES; i++)
{
  // sum the total number of traces for all modules
  totaltraces += List_Data[i+ MAX_NUMBER_OF_MODULES];
}

// allocate memory to hold the starting address, trace length, and
// energy of each trace (therefore, 3 32-bit words are needed for each
// trace.)

traceposlen = (U32)malloc(totaltraces*3*NUMBER_OF_CHANNELS);
if(traceposlen == NULL)
{
  // Error handling
}

// locate traces in the data file
retval = Pixie_Acquire_Data(0x7002, traceposlen, DataFile, modnum);
if( retval < 0 )
{
  // Error handling
}

// allocate memory to hold the first trace; 2 extra 16-bit words for
// notifying the API the trace position and length information
Trace0 = (U16)malloc(traceposlen[1]+2);
if(Trace0 == NULL)
{
  // Error handling
}

Trace0[0] = traceposlen[0];  // position of the first trace
Trace0[1] = traceposlen[1];  // length of the first trace

// read out the first trace and put it into trace0
retval = Pixie_Acquire_Data(0x7003, trace0, DataFile, modnum);
if( retval < 0 )
{
  // Error handling
}
```

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 PIXIE-4 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 PXIcode.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 PIXIE-4 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 PIXIE-Viewer.

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.

MODCSRA: The Module Control and Status Register A

Bit 0: If set, timestamps are latched by local triggers, not by the (last) group trigger. This preserves trigger timing information for runs without waveform acquisition

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: Bits 2 and 15 control trigger distribution over the backplane. If neither bit 2 or bit 15 are set, triggers are distributed only between channels of this module. Otherwise, triggers are distributed as follows:

Bit 2	Bit 15	Function
0	0	Triggers are distributed within module only, no connection to backplane
1	0	Module shares triggers using bussed wire-OR line. In systems with less than 8 modules and no PXI bridge boundaries, all modules sharing trigger should be set this way
0	1	Module receives triggers from master trigger lines, but uses neighboring lines to distribute triggers from right to left. In systems with more than 7 modules and/orPXI bridge boundaries, all modules except the leftmost should be set this way
1	1	Module puts own triggers and triggers received from right neighbor on the master trigger lines and responds to triggers on master trigger line. In systems with more than 7 modules and/orPXI bridge boundaries, the leftmost module should be set this way

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: If set, use signal on front panel input "DSP-OUT" (between channel 1 and 2) and distribute on backplane to all modules as Veto signal (GFLT). Note that only one module may enable this options to avoid a conflict on the backplane.

Bit 6 -8: reserved

Bit 9: If set, module writes the value of NNSHAREPATTERN to its left neighbor during ControlTask 5, using a PXI neighbor line. The left neighbor should be a PXI PDM. The values specifies which coincidence test is applied in the PDM.

Bit 10, 11: reserved

Bit 12: If set, the module will drive low the TOKEN backplane line (used to distribute the result of the global coincidence test) if its local coincidence test fails. This way a module can inhibit all other module from acquiring data.

Bit 13: If set, the module will send out its hit pattern to slot 2 using the PXI STAR trigger line for each event.
This option must not be enabled in slot 2, because slot 2 can not send signals to itself. The line is instead used for chassis clock distribution and therefore should be left alone.

Bit 14: If set, the front panel input "DSP out" is connected as an input to the "Status" line on the backplane. The Status line is set up as a wire-OR, so more than one module can enable this option.

Bit 15: Controls sharing of triggers over backplane. See bit 2.

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.

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

RUNTASK: This variable tells the Pixie-4 what kind of run to start in response to a run start request. Nine 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**
512 (0x200)	**Standard fast list mode**	**No**	**9**
513 (0x201)	**Compressed fast list mode**	**No**	**9**
514 (0x202)	**Compressed fast list mode**	**No**	**4**
515 (0x203)	**Compressed fast list mode**	**No**	**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.

RunTask 512 (0x200) employs the same internal data format as RunTask 256, but omits buffer-full checks and trace capture. The run is stopped when the required number of events (MaxEvents) has been acquired. This run type uses the shortest possible interrupt routine for raw data gathering. Hence, it allows for the shortest time between two logged events. For best results the channel variables PAFLength and TriggerDelay should be set to 1 for all channels involved. Level-1 buffer is not used in this run type. 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 513 (0x201) requests a compressed fast list mode run without trace capture. Both Level-1 buffer and I/O buffer are used in this RunTask. 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 514 (0x202) requests a compressed fast list mode run. The only difference between RunTask 514 and 513 is that in RunTask 514, 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 515 (0x203) requests a compressed fast list mode run. The only difference between RunTask 515 and 513 is that in RunTask 515, only two words of results (time of arrival and energy) are written into the I/O buffer for each channel.

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. The module variable MAXEVENTS should

be set to zero, to avoid early run termination due to a MAXEVENTS-exceeded condition.

The RunTask can be chosen as the run type in the Run tab of the PIXIE-4 Viewer.

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.

MAXEVENTS: The module ends its run when this number of events has been acquired. In PIXIE-4 Viewer, MAXEVENTS is automatically calculated when a run mode is chosen from the run type pulldown menu. The calculation is based on the trace lengths set by the user. Set MaxEvents=0 if you want to switch off this feature, e.g., when logging spectra (done automatically in an MCA mode run).

COINCPATTERN: When a PIXIE-4 is operated on its own, the user can request that certain coincidence/anticoincidence patterns are found 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 Pixie-4 to accept any hit pattern as valid.

In the PIXIE-4 Viewer this variable can be set in the Coincidence Pattern Edit Panel reachable through the Settings tab by clicking on Edit next to the Coinc. Pattern entry.

COINCWAIT: Duration of the coincidence time window in clock ticks (each clock tick spans 13.3 ns). For this feature to work, bit no. 1 of the ChannelCSRA of the involved channels should be cleared. This ensures that the DSP can at the end of the coincidence window suppress further hits reporting by late channels.

In the PIXIE-4 Viewer this bit is set or cleared in line 1 of the Channel CSRA Edit Panel. The line has the title "Measure individual live time". Make sure it is unchecked, so the DSP globally controls FPGA triggering and live time measurements.

When acquiring long waveforms it may be necessary to delay DSP data reading to ensure that the FIFOs will contain valid data. Secondly, when using a filter range of 6 in the FPGA, the minimum value for COINCWAIT is

larger than 1 in all circumstances. Use the following formula to determine COINCWAIT:

COINCWAIT = $\text{Max}(\text{PeakSep}*2\text{^FilterRange})_{\text{ch0-ch3}} - \text{Min}(\text{PeakSep}*2\text{^FilterRange})_{\text{ch0-ch3}}$

Choose COINCWAIT big enough such that the requirements of all channels in the module are met.

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, one or multi-modules will run synchronously through the backplane. This kind of set up in connection with SyncWait=1 will ensure that the last module ready to actually begin data taking will start the run in all modules. And the first module to end the run will stop the run in all modules. This way it never happens that a multi-Pixie system is only partially active.

INSYNCH: InSynch is an input/output variable. It is used in multi-Pixie systems in which the modules are driven by a common clock. When InSynch is 1, the module assumes it is in synch with the other modules and no particular action is taken at run start. If this variable is 0, then all system timers are cleared at the beginning of the next data acquisition run (RunTask>0). The timers are reset when the entire system actually starts the run. After run start, InSynch is automatically set to 1.

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

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.

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.

MODULEPATTERN: To determine if an event is acceptable according to local or global coincidence tests, the DSP computes the quantity (MODULEPATTERN AND (HITPATTERN AND 0x0FFF). If nonzero, the event is accepted.

HITPATTERN bits 4..7 contain the following status information:
4: Logic level of FRONT panel input
5: Result of LOCAL coincidence test
6: Logic level of backplane STATUS line
7: Result of GLOBAL coincidence test (TOKEN backplane line)
Logic levels are captured at the time the coincidence window closes.

Consequently, to accept events based only on the local coincidence test, bit 5 of MODULEPATTERN must be 1, and all others zero. To accept events based only on the global coincidence test, bit 7 of MODULEPATTERN must be 1, and all others zero. To accept events based on both tests (either test passed => accept), set bits 5 and 7 of MODULEPATTERN to one, others to zero.
Other values of MODULEPATTERN can in principle be used, but are not tested and/or supported at this time

NNSHAREPATTERN: 16 bit user defined control word for PXI-PDM. If enabled (MODCSRA), the Pixie-4 module writes this word to its left neighbor using a PXI left neighbor line. The PDM uses this word to make a coincidence accept/reject decision based on the hit pattern from all modules

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.

MODCSRC: The Module Control and Status Register C

Bits 0-15: Reserved for user code.

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

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. 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 PIXIE-4 operation, see the Channel CSRA Edit Panel reachable through the Settings tab of the PIXIE-4 Viewer. In general, setting the bit activates the option in question.

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: Measure individual live time.
Keep this bit cleared when operating with master and slave channels, or when making coincidence measurements using single modules. Set this bit when measuring independent spectra, i.e., when list mode data are not required.

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 or veto events based on a global signal. GFLT is distributed over a PXI bussed line, affecting all channels with this bit set. When the bit is cleared, the GFLT input is ignored. When set, the event is accepted only if validated. To be validated, the GFLT input must be a logic 0 no later than an energy filter rise time after the signal arrival, and must remain at logic 0 level until a rise time + flat top after signal arrival. Polarity can be reversed in CHANCSRC (to be implemented)

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: Reserved.

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 (13.3 ns). Time zero is the start of the waveform.

Bit 11: Reserved.

Bit 12: GATE required (to be implemented)
Set this bit if you want to validate or veto events based on a individual signal. GATE is distributed over a PXI left neighbor line, for example from a PDM in the slot to the left of the Pixie-4. Each channel has its own line. When the bit is cleared, the GATE input is ignored. When set, the event is accepted only if validated. To be validated, the GFLT input must be a logic 0 within the time window defined in GATEWINDOW after the rising edge of the pulse, Polarity can be reversed in CHANCSRC

Bit 13: Reserved.

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.

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

The following two data words are used to set the on-board DACs for this channel. Once a new variable has been written to DSP memory the DACs have to be reprogrammed by starting a run with RunTask=0 and ControlTask=0.

GAINDAC: Reserved and not supported.

TRACKDAC: This DAC determines the DC-offset voltage. The offset can be calculated using the following formula:

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

SGA: The index of the relay combinations of the switchable gain amplifier. For a given value of SGA, the analog 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)

DIGGAIN: The digital gain factor for compensating the difference between the user-desired voltage gain and the SGA gain.

UNUSEDA0 or UNUSEDA1: Reserved.

The following block of data contains trigger/filter FPGA data. Once a new variable has been written to DSP memory it has to be activated by starting a run with RunTask 0 (Set DACs) and ControlTask 5 (Program FiPPI).

SLOWLENGTH: The rise time of the energy filter depends on SlowLength:

RiseTime = SlowLength * 2^FilterRange * 13.3 ns

SLOWGAP: The flat top of the energy filter depends on SlowGap:

FlatTop = SlowGap * 2^FilterRange * 13.3 ns.

There is a constraint concerning the sum value of the two parameters:

SlowLength + SlowGap < 127

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

RiseTime = FastLength * 13.3 ns.

Note the constraint: FastLength < 32

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

FlatTop = FastGap * 13.3 ns.

There is a constraint concerning the sum value of the two parameters:
FastLength + FastGap < 32

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:

Filter Range = 0: PeakSample = max(0, SlowLength + Slow Gap – 7)
Filter Range = 1: PeakSample = max(2, SlowLength + Slow Gap – 4)
Filter Range = 2: PeakSample = SlowLength + Slow Gap – 2
Filter Range >= 3: PeakSample = SlowLength + Slow Gap – 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.

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
If PeakSep>33, PeakSep=PeakSample+1

Note the constraint: 0 < PeakSep - PeakSample < 7

FASTADCTHR: Reserved.

FASTTHRESH: This is the trigger threshold used by the trigger/filter FPGA. The value relates to a trigger threshold through the formula:

FASTTHRESH = TriggerThreshold * FASTLENGTH

The TriggerThreshold can be set on the Settings tab of the PIXIE-4 Viewer.

MINWIDTH: Unused.

MAXWIDTH: This value aids the pile up inspector. MaxWidth is the maximum duration, in sample clock ticks (13.3 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 MaxWidth = FastLength + FastGap + SignalRiseTime/13.3 ns.

Note the constraint MaxWidth < 256

Setting Maxwidth=0 switches this part of the pile up inspector off. Indeed it is recommended to begin with MaxWidth=0. Once the other parameters have been optimized, 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.

PAFLENGTH: 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 PIXIE-4 Viewer it is called TraceDelay (measured in microseconds) and is available through the Settings tab.

The recommended setting for PafLength is:

PafLength = TriggerDelay + TraceDelay/13.3ns

Note the constraint: PafLength < 4092.

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

TRIGGERDELAY: 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 1.

RESETDELAY: This variable controls the restarting of the FIFO after it was halted to read the waveform. When triggers are distributed across channels and modules, a halted FIFO is automatically restarted if the trigger/filter FPGA does not receive the distributed event trigger within RESETDELAY 13.3ns clock ticks after the internal event trigger. The default value written by the PIXIE module should not be changed by the user.

FTPWIDTH: Unused.

This completes the list of values that control the trigger/filter FPGAs.

The following input parameters are used by the DSP program. They become active as soon as the first data taking run has been started. Only then will the output parameters reflect the changes made to the set of input parameters.

TRACELENGTH: This tells the DSP how many words of trace data to read. The action taken depends on FIFOlength , which is 1024. If TraceLength < FIFOlength, the DSP will read from the FIFO. In that case individual samples are 13.3 ns apart. If FIFOlength <= TraceLength, the PIXIE-4 code will force the TraceLength to be equal to FIFOlength.

XWAIT: Extra wait states. This parameter controls how many extra clock cycles the DSP waits when reading waveform data in real time rather than out of a FIFO memory. This occurs when acquiring data in list mode and asking for trace lengths longer than FIFOlength. The time between recorded samples is

$\Delta T = (3+XWAIT)*13.3ns$.

XWAIT is used differently when acquiring untriggered traces in a control run with ControlTask=4. In this case, the time between recorded samples is

$\Delta T =$ 4*13.3ns if XWAIT <= 4;
XWAIT*13.3ns if 4 < XWAIT <= 12;
(3 + XWAIT)*13.3ns if XWAIT > 13 (XWAIT has to be multiple of 5)

The following variables affect internal MCA histogramming of the PIXIE-4 module.

ENERGYLOW: Start energy histogram at ENERGYLOW. Only applies to list mode runs.

LOG2EBIN: This variable controls the binning of the histogram. Energy values are calculated to 16 bits precision. The LSB corresponds to $1/4^{th}$ of a 14-bit ADC. The PIXIEs, 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. Observe the following formula to find to 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 histogramming routine of the DSP takes care of spectrum overflows and underflows.

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

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.

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, or8.

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. Set BLCUT to zero to not check baselines, therefore reduce processing time.

ControlTask 6 can be used to measure baselines. Host computer can then histogram these baseline values and determine the appropriate value for BLCUT for each channel according to the standard deviation SIGMA for the averaged baseline value. BLCUT could be set to be three times SIGMA.

BASELINEPERCENT: This variable sets the DC-offset level in terms of the percentage of the ADC range.

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/0.0133
If intdt>13, XAVG = floor(65536/((intdt-3)/5))
If intdt<=13, XAVG = 65535.

CHANCSRC: Control and status register C. (to be implemented)

Bit 0: GFLT polarity.
Controls polarity of GFLT to be considered present

Bit 1: GATE polarity.
Controls polarity of GATE to be considered present

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

UNUSEDB0 or UNUSEDB1: Reserved.

CFDREG: Reserved for FPGA-based constant fraction discriminator.

LOG2BWEIGHT: The PIXIE 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 = DC + (DC_avg-DC) * 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.

As before, the DSP ensures that LOG2BWEIGHT will be negative. 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 PIXIE 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, and this parameter does not need to be adjusted afterwards.

PREAMPTAUA: High word of the preamplifier exponential decay time.
PREAMPTAUB: Low word of the above.

The two variables are used to store the preamplifier decay time. 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

This ends the block of channel input data. Note that there are four equivalent blocks of input channel data, one for each PIXIE-4 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 interrupt when InSynch was set to 0 prior to the run start. This requires the Busy--Synch loop to be closed; see the discussion above.

RealTime =(RealTimeA * 65536^2 + RealTimeB * 65536 + RealTimeC) * 13.3ns

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 real time allows judging the overhead due to data readout.
Compute the run time using the following formula:

RunTime =(RunTimeA * 65536^2 + RunTimeB * 65536 + RunTimeC) * 13.3ns

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 Busy-Synch 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 ChannelHeadLengths 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 not enough room in the level-1 buffer, which contains only 2048 words.

EMWORDS, EMWORDS2: Each of these variables are 32bit integers counting the number of 16 bit words written to external memory.

Below follow the addresses and lengths of a number of data buffers used by the DSP program. 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 will typically change with every new compilation. Therefore your code should never assume to find any given buffer at a fixed address.

Note that addresses in the DSP data memory fall into the range from 0x4000 to 0x7FFF. The word length in data memory is 16 bit. If an address falls in the range from 0 to 0x3FFF, it points to a location in program memory. Here the word lengths are 24 bits.

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

AOUTBUFFER: Address of the list mode data buffer.
LOUTBUFFER: Number of words in the list mode buffer.

AECORR: unused, reserved
LECORR: unused, reserved.
Formerly address and length of an array containing coefficients for energy calculations. Now these coefficients are calculated in the DSP code from the decay time.

ATCORR: unused, reserved
LTCORR: unused, reserved
Formerly address and length of an array containing coefficients for normalization and time of arrival corrections. Now these coefficients are calculated in the DSP code from the decay time.

HARDWAREID: ID of the hardware version
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. It excludes times during which the FPGA was prevented from sending triggers due to ongoing DSP data reads, or when the run was stopped. Convert the three LiveTime words into a live time using the formula:

LiveTime = (LiveTimeA * 65536^2 + LiveTimeB * 65536 + LiveTimeC) * 16*13.3ns

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

OVERFLOWA: reserved
OVERFLOWB:

INSPECA: reserved
INSPECB:

UNDERFLOWA: reserved
UNDERFLOWB:

ADCPERDACA: Gain variable.

ADCPERDACB: Both words currently unused, but reserved

NOUTA: The number of output counts in this channel (high, low)
NOUTB: (to be implemented)

4.5 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. Reprogramming the DACs is required to make effective changes in the values of the variables GAINDAC{0…3}, TRACKDAC{0…3}.

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

Control Task 2: **Disconnect inputs**
Open the input relay to disconnect the PIXIE 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 list mode buffer. At the end of the task the list mode 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 1638 units is chosen. For bipolar signals, like the induced waveforms from a segmented detector, the ADC offset would be 8192 units, i.e., midway between 0 and 16384.

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 PIXIE-4 input. The control task makes 16 measurements at each DAC step and uses the last computed DC-offset value to enter into the data 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 * 13.3ns apart). In the PIXIE-4 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 PIXIE-4 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**
This task writes 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:

$$Baseline = (B1 - B0 * \exp(-XP)) * Bnorm$$

with

$B1 = (B1L + B1H * 65536)$

$B0 = (B0L + B0H * 65536)$

$XP = (SlowLength + SlowGap) * 2^{FILTERRANGE} / (75 * TAU)$

$Bnorm = 2^{-9}$ / SlowLength for Filterrange >=2
$= 2^{-8}$ / SlowLength for Filterrange =1
$= 2^{-7}$ / SlowLength for Filterrange =0

$TAU = PreampTauA + PreampTauB/65536$

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 7..23: reserved

5 Appendix A — User supplied DSP code

5.1 Introduction

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

5.2 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-2185 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 PIXIE-4 Viewer we provide powerful diagnostic tools to aid code developing and a data interface to exchange data between the host and the user code. The PIXIE-4 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.

5.3 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.

5.4 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 1000 data words.

Host interface as supported by the PIXIE-Viewer:

UserIn[16] 16 words of input data
UserOut[16] 16 words of output data

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 to 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 PIXIE-4 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:

As noted above, the following arrays are part of the DSP parameters saved in settings files:

UserIn[16] 16 words of input data
UserOut[16] 16 words of output data

In addition, a number of globals have been defined by the main DSP code and are accessible by user code. The full list of variables and arrays can be found in the file "interface.inc". Except for the arrays URetVal and Uglobals, **they should not be modified by user code.**

The return value of UserChannel 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 2 to 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, 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 or 0x202, and no words in RunTask 0x103 or 0x203).

The array Uglobals is used to pass values computed by user routines for other purposes to the main DSP code. This is only used in specific custom functions.

Register usage:

The user routines may use all computational registers without having to restore them. However, the secondary register set cannot be used, because the XIA interrupt routines use these.

The usage of the address registers I0..I7 and the associate registers M0..M7, and L0..L7 is subject to restrictions. These are listed below for the various routines.

The associate registers L,M are preset and guaranteed as follows:

L0..L7 = 0
M0 = 0; M1 = 1; M2 = -1;
M4 = 0; M5 = 1; M6 = -1;
M3 and M7 have no guaranteed values.

UserBegin, UserRunInit, and UserRunFinish:
No further restrictions, but user code must leave the associated registers listed above in exactly this state when exiting.

UserChannel:

I5,I6,I7 L5,L6, M0,M1,M2,M4,M5,M6	These registers may not even temporarily be overwritten, because there are interrupt functions, which depend on the contents of these registers. I5 points to the start of the current event
I0,I1,I3,I4 L0,L1,L2,L3,L4,L7	These registers may be altered, but must be restored on exit.
I2 M3,M7	These registers may be altered and need not be restored

UserEvent:

I5,I6,I7 L5,L6, M0,M1,M2,M4,M5,M6	These registers may not even temporarily be overwritten, because there are interrupt functions, which depend on the contents of these registers. I5 points to the start of the current event
I4 L0,L1,L2,L3,L4,L7	These registers may be altered, but must be restored on exit.
I0,I1,I2,I3 M3,M7	These registers may be altered and need not be restored

5.5 Debugging tools

Besides the debugging tools that are accessible through the PIXIE-4 Viewer, it is also possible to download data into the PIXIE 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 output 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.

6 Appendix B — User supplied Igor code

Starting in version 1.38, Igor contains a number of user procedures that are called at certain points in the operation. These user procedures are contained in a separate Igor procedure file "user.ipf" that is automatically loaded when opening the Pixie Viewer (Pixie4.pxp). By default, the user procedures do nothing, but they can be edited to perform custom functions. It is recommended that the modified procedures be "saved as" a new procedure file user_XXX.ipf and the generic user.ipf be removed ("killed") from the main .pxp file.

6.1 Igor User Procedures

The Igor user procedures called from the current version of the main code are listed below.

Function User_Globals()
This function is called from InitGlobals. It can be used to define and create global specific for the user procedures.
By default it creates a user variable "UserVariant" which can be used to track and identify different user procedure code variants. Variant numbers 0x7FFF are reserved for user code written by XIA.

Function User_StartRun()
This function is called at end of Pixie_StartRun (which is executed at beginning of a data acquisition run) for runs with polling time>0. It can be used to set up customized runs, i.e. initialize parameters etc.

Function User_NewFileDuringRun()
When Igor is set to store output data in new files every N spills or seconds, this function is called at the end of making the new files, before the run has resumed. It can be used to e.g. change acquisition parameters or save the Igor experiment during these multi-file runs.

However, it will interfere with the polling routine, so the time to execute User_NewFileDuringRun should be less than the polling time.

Function User_StopRun()
This function is called at the end of the run. By default it calls another function to duplicate the output data displayed in the standard Igor graphs and panels into a data folder called "root:results". It can be used to process output data

Function User_ChangeChannelModule()
This function is called when changing Module Number or Channel Number. By default it calls a function to update the variables in the User Control panel.

Function User_ReadEvent()
This function is called when changing event number in list mode trace display or digital filter display. By default it duplicates traces and list mode data into the "results" data folder

Function User_TraceDataFile()
This function is called when changing the file name in list mode trace display.

6.2 Igor User Panels

The Igor user panels defined in the current version of the user code are listed below:

Window User_Control()
this is the main user control panel, listing DSP input and output variables and showing several action buttons. This panel can be modified to set user variables and control user procedures.

Window User_Version(ctrlName)
This panels displays the version and variants of the user code:

```
UserVersion      // the version of the user function calls defined by XIA
UserVariant      // the variant of the code written by the user
USEROUT[0]       // the version of the DSP code written by the user
```

6.3 Igor User Variables

The main Igor code defines the global variables and waves below for use in user procedures. The user code can modify these values without interfering with the main code. (An exception is the "UserVersion", which should not be modified, but used to ensure the user code is compatible with the main code.

```
NewDataFolder/o root:results     //the Igor data folder where results for user are stored
```

```
Variable/G root:results:RunTime              // Run Time from run statistics panel
Variable/G root:results:EventRate            // Event rate from run statistics panel
Variable/G root:results:NumEvents            // Total number of events
Wave root:results:ChannelLiveTime            // Channel live time 0..3
Wave root:results:ChannelInputCountRate // Channel input count rate 0..3
String/G root:results:StartTime              // Start time from run statistics panel
String/G root:results:StopTime               // Stop time run statistics panel

Wave root:results:MCAch0                     // Channel 0 histogram
Wave root:results:MCAch1                     // Channel 1 histogram
Wave root:results:MCAch2                     // Channel 2 histogram
Wave root:results:MCAch3                     // Channel 3 histogram
Wave root:results:MCAsum                     // Sum histogram

Wave root:results:trace0                     // channel 0 list mode trace
Wave root:results:trace1                     // channel 1 list mode trace
Wave root:results:trace2                     // channel 2 list mode trace
Wave root:results:trace3                     // channel 3 list mode trace
Wave root:results:eventposlen                // contains trace location (in list mode file)
Wave root:results:eventwave                  // contains data for selected list mode event

NewDataFolder/o root:user          //create the folder for variables defined by user
Variable/G root:user:UserVersion   // the version of the user function calls defined by XIA
Variable/G root:user:UserVariant   // the variant of the code written by the user
```

The format of the wave root:results:eventwave is as follows:

Poition	Content
0	event location in file
1	location of corresponding buffer header in file
2	length of event in file
3..6	tracelength for channel 0..3
7..12	buffer header (see user's manual)
13..15	event header (see user's manual)
16..24	channel header for channel 0 (see user's manual)
25..33	channel header for channel 1
34..42	channel header for channel 2
43..51	channel header for channel 3
52+	trace data for channel 0,1,2,3 (use above tracelength to extract)

7 Appendix C — New double buffer mode for list mode readout

Traditionally, list mode runs acquired one 8K buffer of data at a time, stored in DSP memory. Later, 32 8K buffers were stored in external memory (EM) for faster readout, but

the acquisition still stopped as soon as the DSP filled the 32nd buffer and only resumed after the host read out the Pixie-4 module(s).

In the new "double buffer" mode the DSP fills an 8K buffer and transfers it to EM 16 times, automatically resuming acquisition after each transfer. Buffer fills and transfers are synchronized between modules. After the 16th buffer transfer, the DSP indicates data is ready, then switches to a different block in the external memory and resumes acquisition without waiting for host readout. The host can read out the data from the external memory in its own time, notify the DSP the memory block is now available again, and wait for the next 16 buffers to become available. Runs can continue indefinitely unless the host is too slow to read out the memory – if the DSP ever fills both blocks, it stops the acquisition and waits for readout/resume by the host. Note that in systems with several modules, there are now *groups of 16 buffers per module* following each other in the output data file, not individual buffers.

In low count rate applications, buffers might fill slowly, so it might take a long time to get 16 buffers for the next update of list mode data. For more frequent updates, set MAXEVENTS to a smaller number so that the 8K buffers are only partially filled before transfer to the external memory.

The transfer dead time – time between last event in a filled buffer and the start of the next buffer – is about 550us. This includes readout and processing of the last event and resetting counters etc when resuming acquisition. Buffer fill times depend on runtype, length of traces, and count rates. Host readout times are typically ~30ms per module, but may be longer if the host PC is busy with other processes. As long as the host reads out faster than 16 buffers are filled and transferred, there is no additional readout dead time. Otherwise, since 32 buffers are read out in less than twice the time of 16 buffers (PCI setup takes longer than actual burst read), the older 32x buffer mode may still be more efficient.

To use this function in Igor, select the corresponding checkbox in the Run Tab to enable. Select the number of (16 buffer) spills and start a run as usual. The run finishes when the number of spills is reached; but there is likely one extra (partial) spill that was in progress when Igor read the last spill and ended the run.

When programming modules directly, make the following changes:

To enable, set bit 0 of the DSP variable DBLBUFCSR to 1 and clear MODCSRA bit 1 in all modules.

During run, poll the CSR by calling "AcquireData" with Runtype 0x40FF which returns the full CSR value. Bit 14 is set to 1 by the DSP when data is ready. Bit 13 is 1 while the run is active, if it is zero the run ended because both EM blocks are filled.

To read out, first read the DSP output parameters EMwords and EMwords2. Compute the number of 16 bit words in blocks 1 (and 2) as

Nw16_1 = (EMwords) * 65536 + (EMwords+1)

Nw16_2 = (EMwords2) * 65536 + (EMwords2+1)

If Nw16_1 >0, data is in block 1. If Nw16_2>0, data is in block 2. If both are nonzero, the run will have stopped because the DSP has no room to store additional data. Both blocks have to be read (block 1 first if DBLBUFCSR, bit 3 =1) and the run

has to be resumed.
The readout itself follows the usual procedure in Pixie_IOEM of a) setting bit 2 in the CSR to request access to the EM, b) waiting for CSR bit 7 to be clear, c) reading Nw32_1 (or Nw32_2) 32 bit words from block 1 (or block 2), and finally d) clearing bit 2 of the CSR again. Number of words and address are defined as follows:

Nw32_1(2) = ½ * Nw16_1(2), add 1 if Nw16_1(2) is odd,
EM address block 1 = 128K
EM address block 2 = 196K

After readout, the host must set bit 1 in DBLBUFCSR and write it into DSP memory to notify the DSP that the block just read is now freed up. The bit is cleared by the DSP when it updates its internal counters during the next buffer transfer. The host should also perform a dummy read from the Word Count Register to clear CSR bit 14.

As an example, see Write_List_Mode_File, which is the only function in the Pixie-4 C library modified for the double buffer readout.

To stop runs, it is recommended that the host keeps count of the number of spills and/or time and issues a Stop Run command when a user defined number of spills and/or acquisition time is reached. There is no counting of time or spills in the DSP; runs only stop when both EM blocks are filled or due to a host stop.

In the future, the information in EMwords, EMwords2 and DBLBUFCSR may be moved to CSR bits and the Word Count Register so that no reads/writes to DSP memory are required during readout. There will also be further modifications to reduce the transfer dead time.