FalconXn Quick Start Guide

v1.1.11

November 19, 2016

Contents

1 Contents

2 Introduction

2.1 Intended Audience

This document is intended for those users who would like to interface to the XIA FalconXn hardware using the Handel driver library. Users of the Handel driver library should be reasonably familiar with the C programming language and this document assumes the same.

2.2 Sample Code

The Quick Start Guide includes inline code examples to illustrate how specific features are used. In addition to the inline code examples, sample applications are included with Handel. Precompiled versions of the applications (for Windows) are also packaged with Handel.

Each application requires a file called falconxn.ini be present in the same directory as the application. A sample file is included in the distribution.

hqsg-falconxn

This application walks through all of the steps required to acquire a single MCA histogram with 5 seconds worth of data. This is the simplest example and shows how to use basic Handel from start to finish.

2.3 Understanding Handel

2.3.1 Header Files

Before introducing the details of programming with the Handel API, it is important to discuss the relevant header files and other external details related to Handel. All code intending to call a routine in Handel needs to include the file **inc/handel.h** as a header. To gain access to the constants used to define the various logging levels, the file **inc/md_generic.h** must be included; additional constants (preset run types, mapping mode controls, etc.) are located in **inc/handel_constants.h**. The last header that should be included is **inc/handel_errors.h**, which contains all of the error codes returned by Handel.

2.3.2 Error Handling

A good programming practice with Handel is to compare the returned status value with **XIA_SUCCESS** – defined in **handel_errors.h** – and then process any returned errors before proceeding. All Handel routines (except for some of the debugging routines) return an integer value indicating success or failure. While not discussed in great detail in this document, Handel does provide a comprehensive logging and error reporting mechanism that allows an error to be traced back to a specific line of code in Handel.

2.3.3 Thread Safety

Handel uses background threads to serialize IO processing, but the APIs are not thread-safe. It is the responsibility of any application making Handel calls to ensure that only one thread is allowed to access Handel at a time.

2.3.4 .ini Files

The last required file external to the actual Handel source code is an initialization, or ".ini" file. The .ini file defines the setup of your system by breaking the configuration down into the following categories: detector ([detector definitions]), firmware ([firmware definitions]), hardware ([module definitions]) and acquisition values ([default definitions] [1]). Each category in the .ini file contains a series of blocks, surrounded by a set of `START` / `END` delimiters. Each block represents one instance of the logical type associated with the category. For instance, each block in [detector definitions] represents a physical detector with some number of elements. Within each block is a series of key-value pairs used to define the configuration.

[detector definitions]

Each block in this section is used to define one physical detector made up of 1..N channels.

alias Human-readable string naming this detector. Other sections, such as [module definitions], that need to reference the detector will do so using this string.

number_of_channels An integer describing the number of elements in your detector. If this value is set to `N`, the strings `{alias}:{0}` through `{alias}:{N - 1}` will be available to map detector elements to hardware channels in the [module definitions] section.

type The detector type. This field remains in the format for backward compatibility but is not used for the FalconXn. Supported values are the strings `reset` and `rc_feedback`.

type_value For `reset` detectors, a double defining the reset delay time of the preamplifier(s) in microseconds. For `rc_feedback` detectors, a double defining the RC decay constant in microseconds.

channel{n}_gain The preamplifier gain, as a double, for detector element `n` specified in mV/keV.

channel{n}_polarity A string defining the polarity of detector element `n`. Allowed values are "+" or "pos" for a preamplifier whose pulses are positive and "-" or "neg" for one with negative pulses.

[firmware definitions]

Each block in this section defines a reference to a single .bin file, a text file containing pulse characterization information used to optimize spectroscopic performance for a particular detector and configuration. Handel generates the file. The user must not modify the file and only needs to ensure that it is present, i.e. always copy it around with the .ini file for deployment to different directories or systems. [2]

> **Note**
>
> FalconXn differs from other XIA products and their use of this configuration section in that it does not require firmware to be loaded during system startup (firmware resides on the card).

[1] The original name for acquisition values was "defaults".

[2] We plan to include characterization data in the .ini file itself in a future release to simplify management of this data.

alias Human-readable string naming this characterization file. Other sections, such as [module definitions], that need to reference the detector will do so using this string.

filename Path to the .bin file. The path must be an absolute path or relative to the process working directory. [3]

[default definitions]

All of the acquisition values for each channel are stored in this section. Handel auto-generates this section and the user is not expected to modify it.

[module definitions]

Each FalconXn module in a system gets a separate block in this section. The blocks map each FalconXn channel to firmware, acquisition values, and a detector channel, and specify the hardware configuration for each module.

alias Human-readable string naming this module.

module_type Always set to `falconxn`.

number_of_channels Set to 1-8, depending on the number of channels supported by your FalconXn and how many of them you would like to use.

interface Always set to `inet`.

inet_address TCP address of the FalconXn.

inet_port TCP port of the data acquisition server on FalconXn. Typically set to 8756.

inet_timeout TCP timeout in milliseconds. 100 milliseconds should suffice on a fast network, which is desirable for reliable data acquisition, but if timeout errors are returned from Handel APIs you can increase this number.

channel{n}_alias Specifies a global, unique index for channel `n`, where `n` is 0 .. `N-1`. This could be This value is referenced throughout Handel as the `detChan`.

channel{n}_detector Associates channel `n`, where `n` is 0 .. `N-1`, with a specific detector element. The detector element is referenced as `{alias}:{m}` where `alias` is a valid detector alias and `m` is a valid channel for `alias`.

firmware_set_chan{n} Assigns a .bin file to module channel `n`, where `n` is 0..`N-1`. The .bin file is specified as a firmware alias.

2.3.5 detChans

Most routines in Handel accept a `detChan` integer as the first argument. As discussed in [module definitions], each channel in the system must be assigned a unique ID. Handel .ini files generated by ProSpect follow the convention of assigning 0 to the first channel in the first module and `N` - 1, where `N` is the total number of channels in the system, to the last channel in the last module.

Some routines in Handel – mostly routines that set a value – allow the special `detChan` -1 to be passed as an argument. This special value represents all of the `detChans` in the system. When calling routines like **xiaSet↩ AcquisitionValues()** the -1 `detChan` is a convenient shortcut that eliminates the need to loop over all channels.

3 MCA Data Acquisition

3.1 Initializing Handel

Before acquiring data with Handel it is necessary to initialize the library using an .ini file. This page assumes you have saved an .ini file using ProSpect and have copied both the .ini file and any referenced .bin files to the example working directory. See .ini Files for details on the .ini file format.

[3] We plan to encode support paths relative to the INI file in a future release.

```
status = xiaInit("falconxn.ini");
```

Once the initialization is complete, the next step is to call **xiaStartSystem()**. This is the first time that Handel attempts to communicate with the hardware specified in the .ini file. **xiaInit()**'s job is to prepare Handel for data acquisition, while **xiaStartSystem()**'s is to prepare the hardware.

```
status = xiaStartSystem();
```

Once **xiaStartSystem()** is complete, Handel is ready to perform data acquisition tasks. **xiaStartSystem()** only needs to be called once after an .ini file is loaded.

3.2 Configuring Data Acquisition

After **xiaStartSystem()** has run, the FalconXn system is ready to be configured for data acquisition. If the .ini file that was loaded with **xiaInit()** contains a [default definitions] section, then the hardware will be configured using those values. If the default definitions section is missing, then Handel uses a nominal set of default values. Default settings are sufficient for many acquisition values, but most systems will require optimization of at least the analog settings to obtain a good spectrum.

Handel provides a comprehensive set of *acquisition values* for controlling the settings the FalconXn. See the Acquisition Values section for the complete list.

Most MCA data acquisition setups will only need to set the handful of values described below. Once a working set of acquisition values has been created, they can be saved to an .ini file using the function **xiaSaveSystem()**.

The Handel routines to control acquisition values are **xiaSetAcquisitionValues()** and **xiaGetAcquisitionValues()**.

The basic setup of an FalconX system involves optimizing the following acquisition values for your system.

```
double detection_threshold = 0.010;
status = xiaSetAcquisitionValues(0, "detection_threshold", &detection_threshold);
CHECK_ERROR(status);

double min_pulse_pair_separation = 25.0;
status = xiaSetAcquisitionValues(0, "min_pulse_pair_separation",
                                 &min_pulse_pair_separation);
CHECK_ERROR(status);

double detection_filter = XIA_FILTER_MID_RATE;
status = xiaSetAcquisitionValues(0, "detection_filter", &detection_filter);
CHECK_ERROR(status);

double scale_factor = 2.0;
status = xiaSetAcquisitionValues(0, "scale_factor", &scale_factor);
CHECK_ERROR(status);
```

Note

Handel for FalconXN does not require the "apply" step you may be used to with other XIA products. Acquisition values are applied immediately. However, you may call the apply operation to check internal consistency of parameters on the board.

3.3 Run Control

At this point, the hardware is properly configured and ready to acquire data. In this section we are interested in starting and stopping a normal MCA run and reading out the spectrum data. As one might expect the runs are started and stopped using the routines **xiaStartRun()** and **xiaStopRun()**.

The run only needs to be started and stopped once per module on the FalconXn. Calling the functions with detChan = 0 or -1 will suffice.

```
status = xiaStartRun(0, 0);
CHECK_ERROR(status);

printf("Waiting 5 seconds to collect data.\n");
Sleep((DWORD)5000);

printf("Stopping the run.\n");
status = xiaStopRun(0);
```

The MCA can be read either while the run is active or after it has been stopped. The MCA histogram is returned as an array of `unsigned long` integers via a call to **xiaGetRunData()** (See the Run Data section for a complete list of FalconXn run data names and data types.)

xiaGetRunData() expects an array equal to (or larger) than the requested MCA length. The MCA length is set using the number_mca_channels acquisition value and can be read as an `unsigned long` by passing the name "[mca_length][@ref rundata_mca_length]" to **xiaGetRunData()** or as a `double` from **xiaGetAcquisitionValues()**. With this in mind, a simple data acquisition session has the following basic footprint:

```
printf("Getting the MCA length.\n");

unsigned long mca_len = 0;
status = xiaGetRunData(0, "mca_length", &mca_len);
CHECK_ERROR(status);

/* If you don't want to dynamically allocate memory here,
 * then be sure to declare mca as an array of length 8192,
 * since that is the maximum length of the spectrum.
 */
printf("Allocating memory for the MCA data.\n");
unsigned long *mca = malloc(mca_len * sizeof(unsigned long));

if (!mca) {
    /* Error allocating memory */
    exit(1);
}

printf("Reading the MCA.\n");
status = xiaGetRunData(0, "mca", mca);
CHECK_ERROR(status);

/* Display the spectrum, write it to a file, etc... */

printf("Release MCA memory.\n");
free(mca);
```

3.3.1 Preset Runs

A common data acquisition technique is to do a "preset" run with a fixed metric of either time or events. A normal MCA run is both started and stopped by the host software; a preset MCA run is started by the host and stopped by the hardware. Allowing the hardware to end the run lets the host application repeatedly acquire data with similar characteristics.

The FalconXn supports three distinct preset run types: realtime, events and triggers. The constants used to define

these run types are in **handel_constants.h**. The realtime (**XIA_PRESET_FIXED_REAL**) preset run instructs the hardware to run until the specified realtime has elapsed. This time is specified in seconds.

The output events (**XIA_PRESET_FIXED_EVENTS**) and input events (**XIA_PRESET_FIXED_TRIGGERS**) preset runs complete when the specified number of input or output events have been collected.

3.4 Cleanup

The last operation that any Handel application must do is call **xiaExit()** to release all hardware and OS resources used by the library.

```
    status = xiaExit();
```

Once **xiaExit()** is called a new system must be loaded with **xialnit()** if you want to use Handel again.

3.5 Full Example Listing

```
/**
 * @example hqsg-falconxn.c
 *
 * @brief This code accompanies the XIA Application Note "Handel Quick Start
 * Guide: FalconXn". This sample code shows how to start and manually stop
 * an MCA data acquisition run.
 */

/*
 * Copyright (c) 2016 XIA LLC
 * All rights reserved
 *
 * Redistribution and use in source and binary forms,
 * with or without modification, are permitted provided
 * that the following conditions are met:
 *
 *   * Redistributions of source code must retain the above
 *     copyright notice, this list of conditions and the
 *     following disclaimer.
 *   * Redistributions in binary form must reproduce the
 *     above copyright notice, this list of conditions and the
 *     following disclaimer in the documentation and/or other
 *     materials provided with the distribution.
 *   * Neither the name of XIA LLC
 *     nor the names of its contributors may be used to endorse
 *     or promote products derived from this software without
 *     specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 * CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 * IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
 * TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
```

```
 * THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */


#include <stdio.h>
#include <stdlib.h>

#pragma warning(disable : 4115)

/* For Sleep() */
#include <windows.h>

#include "handel.h"
#include "handel_errors.h"
#include "handel_constants.h"
#include "md_generic.h"


static void CHECK_ERROR(int status);


int main(int argc, char *argv[])
{
    int status;

    /* Set up logging */
    printf("Configuring the Handel log file.\n");
    xiaSetLogLevel(MD_WARNING);
    xiaSetLogOutput("handel.log");

    printf("Loading the .ini file.\n");
    status = xiaInit("falconxn.ini");
    CHECK_ERROR(status);

    /* Boot hardware */
    printf("Starting up the hardware.\n");
    status = xiaStartSystem();
    CHECK_ERROR(status);

    printf("Setting the acquisition values.\n");

    /* [Configure acquisition values] */
    double detection_threshold = 0.010;
    status = xiaSetAcquisitionValues(0, "detection_threshold", &detection_threshold);
    CHECK_ERROR(status);

    double min_pulse_pair_separation = 25.0;
    status = xiaSetAcquisitionValues(0, "min_pulse_pair_separation",
                                     &min_pulse_pair_separation);
    CHECK_ERROR(status);

    double detection_filter = XIA_FILTER_MID_RATE;
    status = xiaSetAcquisitionValues(0, "detection_filter", &detection_filter);
    CHECK_ERROR(status);

    double scale_factor = 2.0;
    status = xiaSetAcquisitionValues(0, "scale_factor", &scale_factor);
```

```
    CHECK_ERROR(status);
    /* [Configure acquisition values] */

    /* Start a run w/ the MCA cleared */
    printf("Starting the run.\n");
    status = xiaStartRun(0, 0);
    CHECK_ERROR(status);

    printf("Waiting 5 seconds to collect data.\n");
    Sleep((DWORD)5000);

    printf("Stopping the run.\n");
    status = xiaStopRun(0);
    CHECK_ERROR(status);

    /* [Read MCA spectrum] */
    printf("Getting the MCA length.\n");

    unsigned long mca_len = 0;
    status = xiaGetRunData(0, "mca_length", &mca_len);
    CHECK_ERROR(status);

    /* If you don't want to dynamically allocate memory here,
     * then be sure to declare mca as an array of length 8192,
     * since that is the maximum length of the spectrum.
     */
    printf("Allocating memory for the MCA data.\n");
    unsigned long *mca = malloc(mca_len * sizeof(unsigned long));

    if (!mca) {
        /* Error allocating memory */
        exit(1);
    }

    printf("Reading the MCA.\n");
    status = xiaGetRunData(0, "mca", mca);
    CHECK_ERROR(status);

    /* Display the spectrum, write it to a file, etc... */

    printf("Release MCA memory.\n");
    free(mca);

    /* [Read MCA spectrum] */

    printf("Cleaning up Handel.\n");
    status = xiaExit();
    CHECK_ERROR(status);

    return 0;
}


/*
 * This is just an example of how to handle error values.  A program
 * of any reasonable size should implement a more robust error
 * handling mechanism.
 */
```

```
static void CHECK_ERROR(int status)
{
    /* XIA_SUCCESS is defined in handel_errors.h */
    if (status != XIA_SUCCESS) {
        printf("Error encountered! Status = %d\n", status);
        getchar();
        exit(status);
    }
}
```

4 Legal

4.1 Licenses

4.1.1 Handel

4.1.2 Documentation

THIS DOCUMENTATION IS PROVIDED BY XIA LLC "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL XIA LLC BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN C↩ ONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

4.1.3 Disclaimer

Information furnished by XIA LLC is believed to be accurate and reliable. However, XIA assumes no responsibility for its use, nor any infringements of patents or other rights of third parties, which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of XIA. XIA reserves the right to change specifications at any time without notice. Patents have been applied for to cover various aspects of the design of the DXP Digital X-ray Processor.

4.1.4 Patents

Patent Notice

5 Module Index

5.1 Modules

Here is a list of all modules:

6 Module Documentation

6.1 FalconXn Product-specific Members

Names, data types, units, and ranges of the FalconXn product-specific APIs.

Contents

6.1.1 Acquisition Values

Analog Section

Acquisition values controlling the analog front end.

analog_gain

Acts as as a gain multiplier on the analog front end. Range: [1,16].

analog_offset

Sets the offset on the analog front end. Range: [-2048, 2047].

detector_polarity

The input signal polarity, specified as 1, 0, "+", "-", "pos" or "neg". Setting to 0 or negative effectively inverts the signal.

termination

Input termination impedance. 0=1kohm, 1=50ohm. Default: 0.

attenuation

Input attenuation. 0=0dB, 1=-6dB, 2=ground. Default: 0.

coupling

AC or DC coupling. 0=AC, 1=DC. Default: 0.

decay_time

This setting applies to AC coupling mode. Default: 0.

- 0: long
- 1: medium
- 2: short
- 3: very short

dc_offset

Digital DC offset. Range: [-1, 1]

reset_blanking_enable

Enables reset blanking. 1 = enabled, 0 = disabled. Default: enabled.

reset_blanking_threshold

The threshold for reset blanking, in arbitrary units. Range: [-0.99999, 1]. Default: -0.05.

reset_blanking_presamples

Number of samples before reset detection to blank. Range: [4, 125]. Default: 50.

reset_blanking_postsamples

Number of samples after reset detection to blank. Range: [4, 1000]. Default: 50.

clock_speed

Read-only value to get the sample rate in MHz.

adc_trace_length

The number of samples to collect for an ADC trace. This is set automatically by **xiaDoSpecialRun()** using the given `info` argument, so setting the acquisition value effectively has no effect.

Pulse Detection

Acquisition values controlling pulse detection.

detection_threshold

Minimum height for a pulse to be detected. Range: [0, 0.999]. Default: 0.05.

min_pulse_pair_separation

Minimum number of samples between pulses. This setting controls the balance of throughput and resolution. Range: [0, 1023]. Default: 50.

detection_filter

Default: **XIA_FILTER_MID_RATE**.

- **XIA_FILTER_LOW_ENERGY**
- **XIA_FILTER_LOW_RATE**
- **XIA_FILTER_MID_RATE**
- **XIA_FILTER_HIGH_RATE**
- **XIA_FILTER_MAX_THROUGHPUT**

scale_factor

Scale factor for bin scaling and spectrum calibration. Range: [0.5, 200.0]. Default: 2.

MCA Control

Acquisition values controlling MCA data acquisition.

number_mca_channels

The number of bins in the MCA spectrum, specified in bins. This applies to both MCA mode and fast mapping mode. Narrowing the range may improve network performance in fast mapping applications. Range: [128, 4096]. Default: 4096.

mca_spectrum_accepted

Whether to return the accepted spectrum in mca. The accepted spectrum is the standard MCA histogram and so must be enabled in any typical spectroscopy application.

mca_spectrum_rejected

Whether to return the rejected spectrum in mca. The rejected spectrum consists of pulses that triggered but were rejected due to the detection threshold or other pulse processing criteria. It is available for diagnostics purposes only, and, if enabled, is appended on the end of the mca buffer readout. If both the accepted and rejected spectra are enabled, the application must allocate a buffer that is twice the length of number_mca_channels.

mca_start_channel

The lowest bin number in the range of MCA bins to return. This may be used in conjunction with number_mca_channels to narrow the range the range of bins and improve network performance but is not needed in typical applications. Default: 0.

mca_refresh

MCA refresh period in seconds. This controls how often MCA updates are sent from the FalconXn to the client machine. Default: 0.1.

mca_bin_width

MCA bin width in eV/bin. This value is not used to configure the FalconXn but may serve as convenient storage for a multiplier value for scaling graphical axes in user programs.

preset_type

Criteria to stop the run. Include **handel_constants.h** to access constants for the allowed values:

- **XIA_PRESET_NONE**: run indefinitely
- **XIA_PRESET_FIXED_REAL**: run for a fixed elapsed time. Set preset_value in seconds.
- **XIA_PRESET_FIXED_EVENTS**: run until a fixed number of output pulses are counted. Set preset_value to the number of events.
- **XIA_PRESET_FIXED_TRIGGERS**: run until a fixed number of input pulses are counted. Set preset_↩ value to the number of triggers.

preset_value

Preset run criteria specified in counts or seconds. Required when preset_type is anything other than **XIA_↩ PRESET_NONE**.

mapping_mode

Toggles between the various mapping modes. Supported values are:

- 0.0 = mapping mode disabled
- 1.0 = MCA mapping mode
- 2.0 = SCA mapping mode
- 3.0 = List mode

sca_trigger_mode

Include **handel_constants.h** to access constants for the allowed values:

- **SCA_TRIGGER_OFF**
- **SCA_TRIGGER_HIGH**
- **SCA_TRIGGER_LOW**
- **SCA_TRIGGER_ALWAYS**

sca_pulse_duration

Duration of the emitted SCA pulses in ns, will be rounded to multiple of 4ns. Range: [4, 262140]. Default: 400.0.

number_of_scas

Number of SCA regions. Read the run data max_sca_length for maximum number of SCA regions supported.

sca

The sca limits *name* should have the format sca{n}_[lo|hi], where *n* refers to the 0-indexed SCA number.

num_map_pixels

Total number of pixels to acquire in the next mapping mode run. If set to 0.0, then the mapping run will continue indefinitely.

num_map_pixels_per_buffer

The number of pixels stored in each buffer during a mapping mode run. If the value specified is larger then the maximum number of pixels the buffer can hold, it will be rounded down to the maximum. Setting this to -1.0 or 0.0 will automatically set the value to the maximum allowed per buffer.

pixel_advance_mode

Sets the pixel advance mode for mapping mode. The supported types are listed in **handel_constants.h**. Manual pixel advance using **xiaBoardOperation()** is always available; use **XIA_MAPPING_CTL_USER** is the default and can be used to explicitly disable GATE processing on the FalconXN.

- **XIA_MAPPING_CTL_USER**
- **XIA_MAPPING_CTL_GATE**

input_logic_polarity

When pixel_advance_mode is set to use the GATE signal, this acquisition value determines which logic transition stops data acquisition and clears the spectrum for the next pixel.

- **XIA_GATE_COLLECT_HI**: acquire data while the GATE signal is high and trigger pixel advance on the high-to-low transition. If gate_ignore is set to 1.0, continue collecting data during the transition period.
- **XIA_GATE_COLLECT_LO**: acquire data while the GATE signal is low and trigger pixel advance on the low-to-high transition. If gate_ignore is set to 1.0, continue collecting data during the transition period.

gate_ignore

Determines if data acquisition should continue or be halted during pixel advance while GATE is asserted. Set to 1.0 to keep data acquisition active during the transition. Default: 0.0 (ignore data during the transition).

sync_count

Sets the number of SYNC pulses to use for each pixel. Once "sync_count" pulses have been detected, the pixel is advanced.

6.1.2 Run Data

mca_length (unsigned long)

The current size of the MCA data buffer for the specified channel.

mca (unsigned long *)

The MCA data array for the specified channel. The caller is expected to allocate an array of length mca_length and pass that in as the *value* parameter when retrieving the MCA data.

run_active (unsigned long)

The current run status for the specified channel. If the value is non-zero then a run is currently active on the channel.

module_statistics_2 (double *)

Returns an array containing statistics for the module. The caller is responsible for allocating enough memory for at least 9*n elements (where *n* is the number of channels in the module) and passing it in as the *value* parameter. The returned data is stored in the array as follows:

- 0. channel 0 realtime
- 1. channel 0 trigger livetime
- 2. reserved
- 3. channel 0 triggers
- 4. channel 0 MCA events
- 5. channel 0 input count rate
- 6. channel 0 output count rate
- 7. reserved
- 8. reserved
- [repeat for channels 1-7]

6.1.3 Special Runs

adc_trace (double)

Configures an ADC trace according to the two-element *info* array. Element 0 is the number of samples to collect. This value may be coerced according to the allowed range of values, so users should check it after **xiaDoSpecialRun()** returns successfully. The value returned in element 0 should be used to size the array passed to **xiaGetSpecialRunData()**.
The actual trace is triggered and read to the host when **xiaGetSpecialRunData()** is called.

6.1.4 Special Run Data

adc_trace (unsigned int)

Reads out an array of ADC trace samples. The caller is responsible to allocate an array as long as the length value passed back from **xiaDoSpecialRun()** (or by subsequently reading sprundata_adc_trace_length).

adc_trace_length (double)

Gets the length set by the last call to **xiaDoSpecialRun()**.

6.1.5 Board Operations

apply (null)

Acquisition values are applied immediately by xiaSetAcquisitinValues. The apply operation may be called as a debugging step to check internal consistency of the parameters on the board.

get_connected (int)

Pings the FalconXn and returns greater than zero if it is connected. This is provided for diagnostics purposes but may not be needed in typical applications, as the other Handel APIs will return errors if the connection has been lost.

get_channel_count (int)

Returns the number of channels supported by the connected FalconXn. This is provided for diagnostics purposes only. *number_of_channels* as specified in the Handel .ini file is the official number of channels initialized for use in the APIs.

get_serial_number (char ∗)

Returns the serial number as a string. The caller is responsible for allocating *value* of length at least 32.

get_firmware_version (char ∗)

Returns the firmware version as a string. The caller is responsible for allocating *value* of length at least 32.

Index