The Handel Quick Start Guide:

xMAP

Patrick Franz (software_support@xia.com)

Last Updated: January 17, 2007

XAN-DXP-050405

I Intended Audience

This document is intended for those users who would like to interface to the XIA xMAP 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.

II Conventions Used In This Document

- `This style` is used to indicate source code.
- `CHECK_ERROR` is a placeholder for user-defined error handling.

III Preliminary Details

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 *handel.h* as a header. To gain access to the constants used to define the various logging levels, the file *md_generic.h* must be included; additional constants (preset run types, mapping point controls, etc.) are located in *handel_constants.h*. The last header that should be included is *handel_errors.h*, which contains all of the error codes returned by Handel.

Error Codes

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.

.INI Files

The last required file external to the actual Handel source code is the initialization file, also called the .ini file for short. There are actually two methods one can use to configure Handel, but the most common method (and the one discussed in this document) is to use an .ini file. Included with this application note is an .ini file called *xmap_reset_std.ini*. This is a default .ini file intended for use with a single xMAP module and a reset-type detector.

The .ini file must be customized for your specific detector and xMAP system. The first step is to modify the detector settings in "[detector definitions]", specifically **number_of_channels**, **type**, **type_value**, **channel{n}_gain** and **channel{n}_polarity**.

- number_of_channels: This should be set to the number of elements in your detector.
- type: The allowed detector types are "reset" and "rc_feedback". Each type requires a different set of firmware from XIA.
- type_value: For "reset" detectors, this is the reset delay time of the preamplifier in microseconds. For "rc_feedback" detectors, this is the RC decay constant in microseconds.
- channel{n}_gain: This is the preamplifier gain for detector element *n* in mV/keV.
- channel{n}_polarity: The polarity of detector element *n*. The allowed values are "+", "-", "pos" and "neg".

After completing the detector configuration, the next step is to customize the firmware configuration ("[firmware definitions]"). Simply set **filename** equal to the absolute path of the FDD file provided by XIA[1]. The FDD file contains all of the firmware necessary to run an xMAP module. Its file format and other technical details are beyond the scope of this document, however.

The final configuration step is to edit the module data ("[module definitions]"). The most important parameters here are **pci_bus** and **pci_slot**. The best way to get these values is to extract them from the file *pxisys.ini*. On Win32 systems, this file is located in the c:\windows or c:\winnt directory, depending on the version of Windows you are running. Once you have located this file, you need to search for the slot number that your xMAP module is installed in and copy the value of PCIBusNumber to **pci_bus** and the value of PCIDeviceNumber to **pci_slot**.

Example Code
Included with this document is a file called *hqsg-xmap.c* that is meant to illustrate all of the points presented in this guide. *hqsg-xmap.c* is a sample program that initializes Handel, configures an xMAP module, starts a run, stops a run and reads out the MCA spectrum. The executable built from the sample code should be used in the same directory as the included .ini file: *xmap_reset_std.ini*. Also included is separate sample code (*hqsg-xmap-mapping.c*) and a separate .ini file (*xmap_reset_map.ini*) for use with the mapping mode discussion in the later sections of this document.

detChans
Most routines in Handel accept a **detChan** argument as their first parameter. The detChan is a unique value assigned to each channel in the system. Handel .ini files generated by the xManager program automatically assign these values starting at 0 for the first channel in the first module increasing up to $n - 1$, where n is the total number of channels in the system, for the last channel in the last module[2].

In addition to accepting individual detChans, most routines that set values allow the detChan "-1" to be passed in as an argument. detChan = -1 is a detChan representing all the channels in the system and it is automatically created by Handel. The "-1" detChan is convenient for operation such as setting an acquisition value for the entire system.

Unfortunately, not all routines are able to accept the "-1" detChan. xiaBoardOperation(), which does not support the "-1" detChan, is used with two operations, "apply" and "mapping_pixel_next", that need to be done once per module. When using the detChan scheme generated by software like xManager, it is easy to call xiaBoardOperation() once per module using the following code[3]:

```
int i;
int ignored = 0;

for (i = 0; i < TOTAL_CHANNELS_IN_SYSTEM; i += 4) {
    status = xiaBoardOperation(i, "apply", (void *)&ignored);
    CHECK_ERROR(status);
}
```

[1] XIA does not distribute FDD files with application notes since the firmware is updated at a much faster pace then the application notes are. Please use the latest FDD file from our website or the version included with your copy of the xManager software package.
[2]The detChan values assigned to each channel are easily modified in the .ini file by changing the channel{n}_alias entries in the [module definitions] section.
[3]This code will be explained in more detail later. For now, focus on the fact that xiaBoardOperation() is only being called once per module due to the incrementing of *i* by 4 each time through the loop.

IV Initializing Handel

The first step in any program that uses Handel is to initialize the software library. Handel provides two routines to achieve this goal: xiaInit() and xiaInitHandel().[4] The difference between these two initialization methods is that the former requires the name of an initialization file. In fact, xiaInit() is nothing more then a wrapper around the following two functions: xiaInitHandel() and xiaLoadSystem().

```
/*
 * Example1: Emulating xiaInit() using
 * xiaInitHandel() and xiaLoadSystem().
 */

int status;

status = xiaInitHandel();
CHECK_ERROR(status);

status = xiaLoadSystem("handel_ini", "xmap_reset_std.ini");
CHECK_ERROR(status);
```

The above example has the exact same behavior as

```
int status;

status = xiaInit("xmap_reset_std.ini");
CHECK_ERROR(status);
```

Calling xiaInit() is the preferred method for initializing the library.

V Starting The System

Once the initialization task has been completed, the next step is to "start the system". Starting the system performs several operations including validating the hardware information supplied in the initialization file, testing the specified communication interface (PCI, for the xMAP) and downloading the specified firmware to the hardware. Calling xiaStartSystem() is simple:

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

Once xiaStartSystem() has been called successfully, the system is ready to perform the standard DAQ operations such as starting a run, stopping a run and reading out the MCA. If a call is made to a routine like xiaLoadSystem() after xiaStartSystem() is called, then xiaStartSystem() needs to be called again to account for the data modified by xiaLoadSystem().

VI Configuring The xMAP For Data Acquisition

[4] Complete descriptions of all the routines discussed in this manual are in the *Handel API*, available on the XIA website: http://www.xia.com/DXP_Software.html

Setting Data Acquisition Parameters

By default, the hardware starts up with all of its acquisition values in a nominal state. For most systems, the default values will be sufficient to obtain some results from the hardware. However, in order to optimize the hardware for better results, Handel provides access to several "acquisition values" that provide various controls over the hardware. A partial list of the critical acquisition values includes[5]:

- peaking_time
- trigger_threshold
- calibration_energy
- dynamic_range

In this example, the following operating conditions will be assumed: A peaking time of 16 μs, 1000 eV threshold, a calibration energy of 5900 eV (x-rays from an Fe-55 source) and a dynamic range of 47200 eV.

The routine used to control the acquisition values is called xiaSetAcquisitionValues() and takes three arguments: a detChan, the name of the acquisition value to set and the value to set the acquisition value to. The most complicated aspect of this routine (and others in the Handel library) is that the acquisition value is prototyped as a pointer to a void. Many routines in Handel have arguments that are prototyped in the same manner. This is done so that those routines may accept values of different types. In the case of xiaSetAcquisitionValues(), all of the values are currently doubles, which means that calling the routine to set the calibration energy to 5900 eV should have this format:

```
int status;
double calib = 5900.0;

status = xiaSetAcquisitionValues(0,
                                 "calibration_energy",
                                 (void *)&calib);
CHECK_ERROR(status);
```

In other routines, some of the values are integers while others are unsigned long arrays. Using a pointer to a void, all of these types can be accommodated in a single routine.

The following code illustrates how to set the acquisition values listed above:

```
int status;

double pt     = 16.0;   /* microseconds */
double thresh = 1000.0; /* eV */
double calib  = 5900.0; /* eV */
double dr     = 47200.0 /* eV */

status = xiaSetAcquisitionValues(0,
                                 "peaking_time",
                                 (void *)&pt);
CHECK_ERROR(status);

status = xiaSetAcquisitionValues(0,
                                 "trigger_threshold",
                                 (void *)&thresh);
```

[5] A complete list of the acquisition values for the xMAP may be found at the end of this application note.

```
CHECK_ERROR(status);

status = xiaSetAcquisitionValues(0,
                                 "calibration_energy",
                                 (void *)&calib);
CHECK_ERROR(status);

status = xiaSetAcquisitionValues(0,
                                 "dynamic_range",
                                 (void *)&dr);
CHECK_ERROR(status);
```

Applying Data Acquisition Parameters

When all of the acquisition values have been set to the desired numbers, you must "apply" them to the hardware using xiaBoardOperation():

```
int status;
int dummy = 0;

status = xiaBoardOperation(0, "apply", (void *)&dummy);
CHECK_ERROR(status);
```

Any time an acquisition value is modified, it must be applied to the hardware for the changes to take effect. The `dummy` variable is required since you may not pass a NULL value into xiaBoardOperation().

VII Controlling The MCA

At this stage, the board is configured and ready to begin data acquisition. For this example, the tasks we are interested in are starting a run, stopping a run and reading out the MCA spectrum data.

Starting/Stopping a Run

The Handel interface to starting and stopping the run are two simple routines: xiaStartRun() and xiaStopRun(). Both routines require a detChan (like xiaSetAcquisitionValues()) as their first argument. xiaStartRun() also requires an unsigned short that determines if the MCA is to be cleared when the run is started. To start a run with the MCA cleared, run for 5 seconds and then stop the run, the following code may be used:

```
int status;

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

/* If not on Windows, use the
 * appropriate system routine.
 */
Sleep((DWORD)5000);

status = xiaStopRun(0);
CHECK_ERROR(status);
```

Reading out the MCA Spectrum

The final step in the example program is to read out the collected MCA spectrum. There are two methods in which this can be done: the first is to statically allocate memory for the spectrum array and the second is to dynamically allocate memory for the spectrum at run-time. The first method is dicussed below, while the latter method is covered in the sample source code included with this document (*hqsg-xmap.c*).

The xMAP hardware has a maximum MCA spectrum length of 16k bins (or 16384), so to safely readout the spectrum without dynamically allocating any memory, an array of length 16384 needs to be statically allocated at compile time:

```
int status;

unsigned long mca[16384];

status = xiaGetRunData(0, "mca", (void *)mca);
CHECK_ERROR(status);
```

At this point, the spectrum may be processed as required.

VIII Cleanup

The last operation that any xMAP application must do is call xiaExit(). This call is essential because it releases resources required by the xMAP driver back to the OS. If your application repeatedly starts, but never calls xiaExit(), a situation could occur where the resources required to operate the xMAP hardware are not available.

```
int status;

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

IX Mapping Mode

The DXP-xMAP is specifically designed to support high speed mapping operations. The current version can store full spectra for each mapping pixel. Future versions will also store multiple regions of interest (defined in up to 32 ROIs) or list mode data in each pixel. The control of the mapping operation is the same independent of the type of data stored.

To better understand how Handel works in mapping mode, it is helpful to briefly describe how the xMAP's memory is organized:

In order to allow for continuous mapping, the memory is organized into two independent banks called buffer 'a' and buffer 'b'. A single bank can be read out by the host while the other is filled by the active data acquisition. Each memory bank is 16 bits wide and 1 Mword (2^20, or 1,048,576 words) deep. For standard spectrum acquisition mode, these banks are combined to form a single 32-bit wide by 1 Mword bank of memory. For continuous mapping, the host computer must be able to read out one entire buffer for the complete system in less than the time it takes to fill one buffer. The minimum pixel dwell time for continuous mapping operation is defined by the readout speed and the size of the system, as well as the number of pixels that can be stored in one buffer; for example, if the system contains four DXP-xMAP modules (so the total data transferred in one buffer read is 4 * (2 bytes) * 1M = 8 MB) and the burst readout speed is 25 MB/sec (a conservative estimate that takes into account system overhead), it would take 320 ms to read out one entire buffer. If that buffer can hold data from 64 pixels (typical when storing full spectra), the minimum pixel dwell time that allows for continuous mapping would be 320/64 = 5 ms[6].

The other key component of mapping data acquisition is understanding how the mapping pixels are advanced. The xMAP supports three modes of pixel advance explained below.

Pixel Advance on GATE Edge

The primary method for advancing the pixel number is to use the GATE digital input as a pixel clock, where the pixel number advances on a defined transition of the signal. In MCA mode, the GATE signal is used to disable data taking, or to coordinate the data taking with an external system (for example, so that the DXP-xMAP data can be correlated with the data from the main ionization counter in an XAFS beam line); in the default polarity setting, if GATE is pulled low, then data acquisition is disabled (the signal polarity can also be set to work the opposite way).

For mapping, transitions on this signal (either low-to-high or high-to-low) can be used to trigger a pixel advance; typically, the trailing edge of the GATE signal (the transition from active data acquisition to the inactive state) is used to trigger the advance. Using transitions on the GATE signal requires that the GATE go to the inactive state briefly between pixels; it is possible to set up the system so that it continues to be active during these transition periods using the acquisition value “gate_ignore”. Please see the DXP-xMAP User's Manual for information on how to select options for the GATE signal (polarity, etc).

Pixel Advance using SYNC Clock

The SYNC signal can also be used to generate the pixel advance. Using this method, the pixel will advance for every N positive pulses on the SYNC line, where N can be set from 1 to 65535. Note that the pulses must be greater than 40 ns wide to be guaranteed to be recognized.

Pixel Advance under Host Control

It is also possible to advance the pixel using Handel. Manually advancing the pixels is slower and does not provide real time control, but it does provide an easy way to test mapping operations.

GATE/SYNC Signal Distribution

Before detailing how to configure each pixel advance mode, it is useful to understand how the xMAP uses the GATE/SYNC signal to synchronize pixel advance across multiple modules. A typical PXI crate backplane is broken into one or more “bus segments”. Small crates, 8 slots or less, will contain a single bus segment, while larger crates can contain as many as 3 bus segments. For GATE/SYNC pixel advance, a single module on each bus segment is designated as a 'master' module. The master module accepts a GATE/SYNC logic signal via a LEMO connector on the front panel and routes the signal to the other modules on the segment using a PXI backplane line. The key point here is that **each bus segment needs a master module** if GATE and/or SYNC signals are used.

Now that the memory organization and pixel advance modes are explained, we can discuss the mechanics of actually configuring the hardware for mapping mode operation[7]:

[6]These numbers are meant to be used as simple guidelines. Depending on the specific system in use, the dwell time may be shorter or longer.

[7]Since the mapping mode code is more involved the other examples in this guide it is in a separate example file called *hqsg-xmap-mapping.c*.

1) Enable mapping mode

```
double mapMode = 1.0;
status = xiaSetAcquisitionValues(-1, "mapping_mode",
                                 (void *)&mapMode);
CHECK_ERROR(status);
```

When "mapping_mode" is set to 1.0, Handel downloads the proper firmware to the xMAP modules if they are not already running it. Handel also updates any acquisition values that are specific to mapping mode. Similarly, setting "mapping_mode" to 0.0 switches the hardware back to MCA acquisition mode.

2) Set the number of bins in the spectrum

```
double nBins = 2048.0;
status = xiaSetAcquisitionValues(-1, "number_mca_channels",
                                 (void *)&nBins);
CHECK_ERROR(status);
```

The number of bins in the spectrum has a direct impact on the number of pixels that can be fit into each buffer.

3) Set the total number of pixels to be acquired in this run.

```
double nMapPixels = 100.0;
status = xiaSetAcquisitionValues(-1, "num_map_pixels",
                                 (void *)&nMapPixels);
CHECK_ERROR(status);
```

Setting "num_map_pixels" to 0 will cause the run to continue indefinitely.

4) Set the number of pixels per buffer.

```
double nMapPixelsPerBuffer = -1.0;
status = xiaSetAcquisitionValues(-1, "num_map_pixels_per_buffer",
                                 (void *)&nMapPixelsPerBuffer);
CHECK_ERROR(status);
```

By setting "num_map_pixels_per_buffer" to -1.0, theDSP will automatically calculate the maximum number of pixels that can fit in each buffer given the number of MCA bins. A specific number of pixels per buffer may be set by specifying a value other then -1.0 for "num_map_pixels_per_buffer". Setting "num_map_pixels_per_buffer" to -1.0 does not report the number of map pixels per buffer that will actually be used. To get the calculated number, simply call:

```
double calculatedPixPerBuf = 0.0;
status = xiaGetAcquisitionValues(0, "num_map_pixels_per_buffer",
                                 (void *)&calculatedPixPerBuf);
CHECK_ERROR(status);
```

If the number of mapping pixels per buffer is set larger then the maximum amount the buffer can hold, it will be truncated to the maximum value by the DSP.

5) Configure pixel control.

At the beginning of a run, the pixel number starts at zero; the pixel number advances in several possible ways, either using digital hardware lines for real time applications or by computer control. These methods are described briefly below:

GATE

```
double enabled = 1.0;
double pixMode = XIA_MAPPING_CTL_GATE;

status = xiaSetAcquisitionValues(0, "gate_master",
                                 (void *)&enabled);
CHECK_ERROR(status);

status = xiaSetAcquisitionValues(0, "pixel_advance_mode",
                                 (void *)&pixMode);
CHECK_ERROR(status);
```

This code sets the module containing detChan 0 as a GATE master module.

SYNC

```
double enabled = 1.0;
double pixMode = XIA_MAPPING_CTL_SYNC;

status = xiaSetAcquisitionValues(0, "sync_master",
                                 (void *)&enabled);
CHECK_ERROR(status);

status = xiaSetAcquisitionValues(0, "pixel_advance_mode",
                                 (void *)&pixMode);
CHECK_ERROR(status);
```

This code sets the module containing detChan 0 as a SYNC master module.

HOST

To advance the point manually when using the Host Control method, the following code can be used:

```
int ignored = 0;
status = xiaBoardOperation(0, "mapping_pixel_next",
                           (void *)&ignored);
CHECK_ERROR(status);
```

Note: To advance the pixel, xiaBoardOperation() only needs to be called **once** per module. Calling it once for each channel on the module will advance the pixel **4 times** per module, which is probably not the desired result.

6) Start a run.

To start the mapping run, start a run on all channels using the usual method:

```
status = xiaStartRun(-1, 0);
CHECK_ERROR(status);
```

7) Monitor the buffer status.

At this point, the first buffer ('a') starts to fill with data. If the pixel advance mode is set to host control, then the points should be advanced manually here. In other pixel advance modes, the pixels will be added to the buffer until it is full. To check the buffer status, use the following code:

```
unsigned short isFull = 0;

while (!isFull) {
    status = xiaGetRunData(0, "buffer_full_a", (void *)&isFull);
    CHECK_ERROR(status);
}
```

This code only determines the status for a single module. For multi-module systems, xiaGetRunData() needs to be called once per module.

8) If a buffer is full, read it out.

Once the buffer is full it needs to be read out. However, before reading the buffer it is useful to know how much memory is required for each buffer:

```
unsigned long bufferLen = 0;
status = xiaGetRunData(0, "buffer_len", (void *)&bufferLen);
CHECK_ERROR(status);
```

The buffer length only needs to be calculated after a change to the number of pixels in each buffer. When the buffer is ready to be read, we can allocate the memory we need and read the buffer:

```
unsigned long *buf = NULL;

buf = malloc(bufferLen * sizeof(unsigned long));

if (!buf) {
    /* ERROR */
}

status = xiaGetRunData(0, "buffer_a", (void *)buf);
CHECK_ERROR(status);

/* Process data here. */
```

9) Signal that the buffer read is complete.

Now that the buffer has been read, the buffer must be cleared so that the hardware can use it to store more pixels:

```
char bufDone = 'a';

status = xiaBoardOperation(0, "buffer_done", (void *)&bufDone);
CHECK_ERROR(status);
```

10) Wait for the next buffer to fill.

With buffer 'a' now cleared, the next step is to wait for buffer 'b' to be full:

```
unsigned short isFull = 0;

while (!isFull) {
    status = xiaGetRunData(0, "buffer_full_b", (void *)&isFull);
    CHECK_ERROR(status);
}
```

11) Goto (7).

Once buffer 'b' is full, it needs to be read out and cleared and buffer 'a' needs to be checked. This process continues until the total number of pixels are collected. To check the current pixel count:

```
unsigned long curPixel = 0;

status = xiaGetRunData(0, "current_pixel", (void *)&curPixel);
CHECK_ERROR(status);

printf("Current pixel = %lu\n", curPixel);
```

12) Stop the run when all of the pixel points are complete.

Once all of the pixels have been collected, the run must be stopped as usual:

```
status = xiaStopRun(-1);
CHECK_ERROR(status);
```

That covers the basics of using the xMAP's mapping mode features. Future application notes will address other advanced modes and uses of the xMAP.

X Mapping Tips

This section describes tips and techniques to be aware of when doing mapping data acquisition runs. Following these guidelines will ensure that your mapping application runs smoothly.

1) **Enabling mapping mode updates all parameters**

   When `mapping_mode` is enabled, all of the relevant acquisition values are downloaded to the hardware. There is no need to try and set these acquisition values again.

2) **Create a mapping "channel" for each module**

   Most of the acquisition values related to mapping mode are module-wide settings and do not need to be set on each channel. Setting them on a single channel per module is sufficient. The technique XIA uses in the Configuration Wizard in xManager is to save all the mapping parameters to the first detChan on each module.

3) **Remove GATE/SYNC master values to disable them**

The temptation with the "gate_master" and "sync_master" values is to set them to 0.0 when they are to be disabled. There is nothing wrong with this technique provided that another master isn't set to 1.0 at the same time. For instance, do not set "gate_master" to 0.0 and "sync_master" to 1.0 on the same module; the result of doing so is dependent on the order in which "gate_master" and "sync_master" are evaluated by Handel. The proper way to remove a module's "gate_master" or "sync_master" status is to use xiaRemoveAcquisitionValues():

```
status = xiaRemoveAcquisitionValues(0, "gate_master");
CHECK_ERROR(status);
```

4) **Cache the mapping buffer length**

The mapping buffer length, "buffer_len", only needs to be read once before the mapping run starts; it will not change during the mapping run.

5) **Assign a single master module per bus segment**

For GATE and SYNC pixel advance modes there needs to be exactly one master module of the appropriate type per bus segment.

6) **Check for buffer overruns**

If the per-pixel acquisition time is too short for the data acquisition system to keep up with, it is possible to overrun the mapping buffer. When the mapping buffer is overrun, the additional pixels will accumulate in the last pixel of the last active buffer. To signal that the buffer has overrun, the value of "buffer_overrun" will be set to 1.0. Additional information may be retrieved from the DSP parameters MAPERRORS and BUFMAPERRORS. MAPERRORS is the total number of overruns in the current run and BUFMAPERRORS is the number of overruns in the current buffer.

In general, once a buffer overrun condition has occurred it can be problematic to reconstruct the data even though nothing has been discarded. XIA recommends treating the buffer overrun condition as an indication that the data acquisition needs more tuning to run at the speed that caused the overrun.

XI Where To Go Next

The information presented above should serve as a basic introduction to operating the xMAP hardware using Handel. Handel, of course, has many more features that were not discussed in this document. The next step to learn more about Handel is to explore the *Handel API* and become more familiar with what Handel has to offer.

If you think you have found a bug, have a question or have a suggestion, please contact XIA at software_support@xia.com.

List of Acquisition Values

- **peaking_time**: Peaking time of the energy filter, specified in µs.
- **dynamic_range**: Energy range corresponding to 40% of the total ADC range, specified in eV.
- **trigger_threshold**: Trigger filter threshold, specified in eV.
- **baseline_threshold**: Baseline filter threshold, specified in eV.
- **energy_threshold**: Energy filter threshold, specified in eV.
- **calibration_energy**: Calibration energy, specified in eV.
- **adc_percent_rule**: Percent of ADC used for a single step with energy equal to the calibration energy. This parameter is mostly provided for backwards compatibility: recommend that you use **calibration_energy** and **dynamic_range** to set the gain of your xMAP system.
- **mca_bin_width**: Width of an individual bin in the MCA, specified in eV.
- **preamp_gain**: Preamplifier gain, specified in mV/keV. This value mirrors the value set in the .ini file under [detector definitions].
- **number_mca_channels**: The number of bins in the MCA spectrum, specified in bins.
- **detector_polarity**: The input signal polarity, specified as "+", "-", "pos" or "neg"
- **reset_delay**: The amount of time that the processor should wait after the detector resets before processing the input signal, specified in µs. This setting is ignored if the preamplifier type is not set to reset.
- **gap_time**: The gap time of the energy filter, specified in µs. NOTE: This acquisition value is read-only. To set the gap time, please see **minimum_gap_time**.
- **trigger_peaking_time**: The peaking time of the trigger filter, specified in µs.
- **trigger_gap_time**: The gap time of the trigger filter, specified in µs.
- **baseline_average**: The number of samples averaged in the baseline, specified as number of samples.
- **preset_type**: Sets the preset run type. See *handel_constants.h* for constants that can be used to set this value.
- **preset_value**: When a preset run type other then 0 is set, this value is either the number of counts or the time (specified in seconds).
- **number_of_scas**: Sets the number of SCAs. (SCAs are discussed in a seperate application note.)
- **sca{n}_[lo|hi]**: The SCA limit (low or high) for the requested SCA (n), specified as a bin number. (SCAs are discussed in a seperate application note.)

- **mapping_mode**: Toggles between the various mapping modes by switching the firmware as appropriate and downloading the necessary acquisition values. Currently, the following values are allowed: 0.0 = standard MCA mode, 1.0 = MultiMCA mapping mode.
- **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 to be 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 will automatically set the value to the maximum allowed per buffer.
- **input_logic_polarity**: Sets the polarity of the logic signal connected to the front panel of the xMAP to either non-inverted (0.0) or inverted (1.0). The default is non-inverted.
- **gate_master**: Sets the current module as a GATE master. Only one GATE master is needed per PXI bus segment. If **gate_master** is set for a given module, all of the other detChans in that module are considered to be GATE master also. Setting **gate_master** to 1.0 enables this feature. Note: To disable **gate_master** in a module, either set **gate_master** to 0.0 or replace **gate_master** with **sync_master**, provided you want to use SYNC control instead of GATE. Do not define **gate_master** and **sync_master** at the same time even if one is set 0.0 and the other is set to 1.0. The result of doing such an operation is undefined in Handel. Another solution is to remove **gate_master** from the acquisition value list via xiaRemoveAcquisitionValues(), which requires an additional call to xiaStartSystem() to properly reset the system.
- **sync_master**: Sets the current module as a SYNC master. Only one SYNC master is needed per PXI bus segment. If sync_master is set for a given module, all of the other detChans in that module are considered to be SYNC master also. To disable/remove **sync_master**, see the discussion for **gate_master**.
- **sync_count**: Sets the number of SYNC pulses to use for each pixel. Once **sync_count** pulses have been detected, the pixel will be advanced.
- **gate_ignore**: Determines if data acquisition should continue or be halted during pixel advance while GATE is asserted.
- **lbus_master**: Sets the current module as an LBUS master. To disable/remove **lbus_master**, see the discussion for **gate_master**.
- **pixel_advance_mode**: Sets the pixel advance mode to be used in mapping mode. The supported types are XIA_MAPPING_CTL_GATE and XIA_MAPPING_CTL_SYNC, defined in *handel_constants.h*. Note: Manual pixel advance via. `xiaBoardOperation()` is always available and does not need to be set using this acquisition value.

- **peak_sample_offset{n}**: Sets the peak sampling time offset constant for DECIMATION n. This value is optional; setting it will override the defined value of the offset in the FDD file. This value is specified in µseconds.
- **minimum_gap_time**: Sets the minimum gap time for the energy/slow filter in µseconds.
- **synchronous_run**: This parameter is used in conjunction with **lbus_master** to enable a synchronous data acquisition run.
- **maxwidth**: Sets the maximum peak width for Pile-Up Inspection in µseconds.
- **preamp_type**: Sets the detector preamplifier type. The allowed types are XIA_PREAMP_RESET and XIA_PREAMP_RC, defined in *handel_constants.h*. The value set for **preamp_type** is synchronized with the detector settings and can be set either in the .ini file or via this acquisition value.
- **decay_time**: Sets the decay time, in µs, for an RC preamplifier. This setting is ignored if the preamplifier type is not RC.

List of Run Data Values and Types†

- **mca_length**: (*unsigned long*) The current size of the MCA data buffer for the specified channel.
- **mca**: (*unsigned long **) The MCA data for the specified channel.
- **baseline_length**: (*unsigned long*) The current size of the baseline data buffer for the specified channel.
- **baseline**: (*unsigned long **) The baseline data for the specified channel.
- **runtime**: (*double*) The realtime run statistic, reported in seconds.
- **realtime**: (*double*) Alias for **runtime**.
- **events_in_run**: (*unsigned long*) The total number of events in the current run.
- **trigger_livetime**: (*double*) The livetime run statistic as measured by the trigger filter, reported in seconds.
- **livetime**: (*double*) The calculated energy filter livetime, reported in seconds.
- **input_count_rate**: (*double*) The measured input count rate reported as counts/second.
- **output_count_rate**: (*double*) The output count rate reported as counts/secound.
- **sca_length**: (*unsigned short*) The number of elements in the SCA data buffer for the specified channel.
- **sca**: (*double **) The SCA data buffer for the specified channel.
- **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.
- **buffer_full_a**: (*unsigned short*) The current status of buffer 'a'. If the value is non-zero then the buffer is full. NOTE: Requires mapping mode.
- **buffer_full_b**: (*unsigned short*) The current status of buffer 'b'. If the value is non-zero then the buffer is full. NOTE: Requires mapping mode.
- **buffer_len**: (*unsigned long*) Size of the mapping buffer. For a given mapping run, this value will remain constant and, therefore, only needs to be read out once at the start of a run. NOTE: Requires mapping mode.
- **buffer_a**: (*unsigned long **) The data in buffer 'a'. NOTE: Requires mapping mode.
- **buffer_b**: (*unsigned long **) The data in buffer 'b'. NOTE: Requires mapping mode.
- **current_pixel**: (*unsigned long*) The current pixel number. The pixel number is reset to 0 at the start of each new mapping run. NOTE: Requires mapping mode.
- **buffer_overrun**: (*unsigned short*) If non-zero, indicates that the current buffer has overflowed. NOTE: Requires mapping mode.

†The italicized value indicates the type of the argument passed into the `value` parameter.

- **module_statistics**: (*double **) Returns an array containing all of the statistics for the module. This array must contain a minimum of 28 elements and must be allocated prior to calling xiaGetRunData(). The returned data is stored in the array as follows:
  [channel0_realtime,
  channel0_trigger_livetime,
  channel0_energy_livetime,
  channel0_triggers,
  channel0_events,
  channel0_icr,
  channel0_ocr,
  ...
  channel3_realtime,
  channel3_trigger_livetime,
  etc.]
- **module_mca**: (*unsigned long **) Returns the MCA data for all 4 channels in a module flattened into a single array. Requires that the MCA length be the same for each channel. The array that the MCA data is stored in must be allocated prior to calling this routine.

List of Board Operations[†]

- **apply**: Apply any recent acquisition value changes to the firmware.
- **buffer_done**: (*char*) Signal that the specified buffer ('a' or 'b') has been read out and may be used for mapping acquisition again. NOTE: Requires mapping mode.
- **mapping_pixel_next**: Advance to the next pixel in a mapping data acquisition run. NOTE: Requires mapping mode.

[†]The italicized value indicates the type of the argument passed into the `value` parameter. If no type is given, then a non-null dummy value needs to be passed in.