Handel Programmer's Guide - XMAP

Contents

Introduction

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.

Conventions

Each Handel API that is discussed is linked to the Handel API manual. Follow the link to view the function prototype and documentation of the API's use.

`CHECK_ERROR` is a placeholder for user-defined error handling.

Sample Code

The guide includes inline code examples to illustrate how specific features are used. In addition to the inline code examples, three sample applications are included with Handel: hqsg-xmap.c, hqsg-xmap-preset.c and hqsg-xmap-mapping.c. Precompiled versions of the applications (for Windows) are also packaged with Handel.

Each application requires a Handel .ini file to be passed as a command line argument. A sample file xmap_reset_std.ini is included in the distribution; it will need at least the pci_bus and pci_slot modified to run with your particular xMAP system.

hqsg-xmap

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.

hqsg-xmap-preset

Preset runs are an important component to most non-mapping xMAP applications. This example shows how to setup a preset run and poll waiting for the run to complete.

hqsg-xmap-mapping

Mapping applications introduce new terminology and uses of Handel; this sample shows how to configure an xMAP module and collect a few buffers worth of data. For a real mapping application, the pixel advance would occur using either a GATE or SYNC input. Unfortunately, configuring GATE or SYNC would make the sample code unnecessarily complicated, in addition to requiring a valid GATE/SYNC signal be present simply to run the sample. Instead we use the manual pixel advance feature on the xMAP running in a separate thread to simulate the normal pixel advance. Please keep this caveat in mind when reading the sample code.

Understanding Handel

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.

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. Additional debug information can be generated by setting up logging in the beginning of the application.

Thread Safety

Handel is 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. For an example of how to protect access to Handel, see the hqsq-xmap-mapping.c sample code.

.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.

> **Note:**
>
> XIA provides a tool as part of the ProSpect application, the *Configuration Wizard*, to generate an .ini file to initially start your system in ProSpect. XIA's general recommendation is to use ProSpect to optimize your system and save an .ini file for use in your application. However, by referring to an existing Handel .ini file and the specifications in this section, you can also edit .ini files by hand or generate them in your application.

[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. Handel currently supports the strings `reset` and `rc_feedback` detectors and uses the specified type to pick the right firmware to download to the xMAP.

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

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

and "-" or "neg" for one with negative pulses.

[firmware definitions]

Each block in this section defines a reference to a single FDD file. FDD files, discussed in more detail below, are a flat file containing logical sets of firmware for the xMAP hardware.

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

filename Valid path to the FDD file.

fdd_tmp_path Valid path to a temporary directory that your application has read/write permissions on. Handel will extract individual firmware files to this directory for subsequent download to the xMAP hardware.

[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 xMAP module is a system gets a separate block in this section. The blocks map each xMAP 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 `xmap`.

number_of_channels Always set to 4.

interface Always set to `pxi`.

pci_bus / pci_slot The PCI bus and slot that this module is connected to. To determine the bus value, you can use the Device Manager on Windows. As mentioned above, the *Configuration Wizard* in xManager will automatically generate an .ini with the correct bus and slot information for all of the xMAP modules attached to a computer.

channel{n}_alias Specifies a global, unique index for channel `n`, where `n` ranges from 0..3. This value is referenced throughout Handel as the `detChan`.

channel{n}_detector Associates channel `n`, where `n` ranges from 0..3, 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 an FDD file to module channel `n`, where `n` ranges from 0..3. The FDD file is specified as a firmware alias.

FDD Files

The xMAP has 4 specialized processors that require programming after the module has been powered up. The firmware used to program these processors is collectively stored in a Firmware Definition Database (FDD) file. XIA distributes current versions of the xMAP FDD files with both Handel and xManager.

detChans

Most routines in Handel accept a `detChan` integer as the first argument. As discussed in *[[module definitions]]*, each xMAP channel in the system must be assigned a unique global ID. Handel .ini files generated by the *Configuration Wizard* in xManager 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. If you know the number of channels in the system, you can treat the detChan parameter as a 0-based index. For more information on robustly querying the system, see Enumerating Modules and Channels in the Handel API Manual.

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 `detChan`s in the system and is automatically created by Handel when it loads a new .ini file. When calling routines like xiaSetAcquisitionValues() the -1 `detChan` is a convenient shortcut that eliminates the need to loop over all channels.

Note that not all routines accept the -1 `detChan`. The primary situation where one is required to loop over all necessary channels is when calling xiaBoardOperation(), specifically with "apply" and "mapping_pixel_next". Note though that both of these operations only need to be called once per module using code like this [2]:

```
int i;
int ignored = 0;

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

[2] This code will be explained in more detail later. The key point in this example is that `xiaBoardOperation` is only being called once per module.

MCA Data Acquisition

Setting up Logging

Handel provides a comprehensive logging and error reporting mechanism that allows an error to be traced back to a specific line of code in Handel. To utilize the logging system, a log file needs to be designated, preferably at the beginning of the application:

```
CHECK_ERROR(xiaSetLogLevel(MD_DEBUG));
CHECK_ERROR(xiaSetLogOutput("handel.log"));
```

See xiaSetLogLevel() and xiaSetLogOutput().

When cleaning up your program resources, you can release the log file handle and redirect logging to stdout by calling [xiaCloseLog()].

Initializing Handel

Before acquiring data with Handel it is necessary to initialize the library using an .ini file:

```
CHECK_ERROR(xiaInit("xmap.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. xiaStartSystem() is one of the longest running functions in Handel, generally speaking, since it needs to validate the hardware configuration supplied in the .ini file, test the communication interface (PCI, for the xMAP) and download all of the required firmware to the module(s). Like xiaInit(), xiaStartSystem() is easy to use:

```
CHECK_ERROR(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.

Configuring Data Acquisition

After xiaStartSystem() has run, the xMAP 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. For most systems, the nominal values are sufficient to obtain some results from the software, but some of the settings should be optimized for actual data acquisition.

Handel provides a comprehensive set of *acquisition values* for controlling the settings of each xMAP module [3]. Most normal 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 xMAP system involves optimizing the following acquisition values for your system:

- peaking_time
- dynamic_range
- trigger_threshold
- calibration_energy

As an example, we will configure the system with a peaking time of 16.0 microseconds, a calibration energy of 5900 eV (an Fe-55 source emitting Mn K-alpha x-rays), a dynamic range of 47200 eV and a trigger threshold of 1000 eV:

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

CHECK_ERROR(xiaSetAcquisitionValues(-1, "peaking_time", &pt));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "trigger_threshold", &thresh));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "calibration_energy", &calib));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "dynamic_range", &range));
```

Once the acquisition values are set, it is necessary to "apply" them with xiaBoardOperation() as discussed in the detChans section.

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. The routines xiaStartRun() and xiaStopRun() control the run. Each routine need only be called once per module on the XMAP.

The MCA can be read while the run is active and/or after it has been stopped. The MCA histogram is returned as an array of `unsigned long` integers via a call to xiaGetRunData() [4].

[3] A complete list of the available xMAP acquisition values and their defaults can be found in the Acquisition Values section.

[4] A complete list of the available xMAP run data names and their return data types can be found in the Run Data section.

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" to xiaGetRunData() or as a `double` from xiaGetAcquisitionValues(). With this in mind, a simple data acquisition session has the following basic footprint:

```
#include <stdlib.h>

unsigned long *mca = NULL;

unsigned long mca_length = 0;

CHECK_ERROR(xiaGetRunData(0, "mca_length", &mca_length));

mca = malloc(mca_length * sizeof(unsigned long));

if (!mca) {
  /* Unable to allocate enough memory. */
}

CHECK_ERROR(xiaStartRun(-1, 0));
/* Collect as much data as you want. */
CHECK_ERROR(xiaStopRun(-1));

CHECK_ERROR(xiaGetRunData(0, "mca", mca));
/* Do something with the MCA histogram. */

free(mca);
```

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 xMAP supports four distinct preset run types: realtime, energy livetime, events and triggers. The constants used to define these run types are in handel_constants.h. The realtime (`XIA_PRESET_FIXED_REAL`) and energy livetime (`XIA_PRESET_FIXED_LIVE`) preset runs instruct the hardware to run until the specified realtime/energy livetime has elapsed. These times are specified in seconds with a granularity of 320 ns. Even though the time granularity is 320 ns, the hardware only checks to see if enough time has elapsed every 8 ms.

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.

```
double preset_realtime = 20.0;
double preset_type = XIA_PRESET_FIXED_REAL;

int ignored = 0;
int n_channels_done;

unsigned long run_active;

CHECK_ERROR(xiaSetAcquisitionValues(-1, "preset_type", &preset_type));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "preset_value", &preset_realtime));

/* Assume only one module in system */
CHECK_ERROR(xiaBoardOperation(0, "apply", &ignored));

CHECK_ERROR(xiaStartRun(-1, 0));

do {
    int i;

    n_channels_done = 0;
    for (i = 0; i < TOTAL_CHANNELS_IN_SYSTEM; i++) {
        CHECK_ERROR(xiaGetRunData(i, "run_active", &run_active));
        if ((run_active & 0x1) == 0) {
            n_channels_done++;
        }
    }

    Sleep(1);
} while (n_channels_done != TOTAL_CHANNELS_IN_SYSTEM);

CHECK_ERROR(xiaStopRun(-1));

/* Read out data here. */
```

SCA Settings

The number of SCA regions can be set via the acquisition values number_of_scas, after which each limits can be set with acquisition values in the format "sca{n}_[lo|hi]", e.g. "sca0_lo", "sca1_hi". After the run has stopped, sca values from each region can be read out from xiaGetRunData() as run data "sca", in an array of doubles, with length equal to the number of predefined SCA regions.

```
double nSCAs = 2.0;
char scaStr[80];

double scaLowLimits[]  = {0.0, 1024.0};
double scaHighLimits[] = {1023.0, 2047.0};

double SCAs[2];

/* Set the number of SCAs */
printf("-- Set SCAs\n");
status = xiaSetAcquisitionValues(-1, "number_of_scas", (void *)&nSCAs);
CHECK_ERROR(status);

/* Set the individual SCA limits */
for (i = 0; i < (int)nSCAs; i++) {
    sprintf(scaStr, "sca%d_lo", i);
    status = xiaSetAcquisitionValues(-1, scaStr, (void *)&(scaLowLimits[i]));
    CHECK_ERROR(status);

    sprintf(scaStr, "sca%d_hi", i);
    status = xiaSetAcquisitionValues(-1, scaStr, (void *)&(scaHighLimits[i]));
    CHECK_ERROR(status);
}

/* Read out the SCAs from the data buffer */
status = xiaGetRunData(0, "sca", (void *)SCAs);
CHECK_ERROR(status);
```

Cleanup

The last operation that any xMAP application must do is call xiaExit(). Once xiaExit() is called a new system must be loaded with xiaInit() if you want to use Handel again.

Mapping Mode

The xMAP is designed to support high-speed mapping operations. The current version can store full spectra for each mapping pixel (MCA mode), up to 64 non-overlapping SCAs per pixel (SCA mode) or time-stamped events (List-mode). In general, the controls are the same for each mapping mode variant though different strategies may be required for performance reasons.

To better understand how mapping mode works, it is helpful to briefly describe how the xMAP's memory is organized. In order to allow for continuous mapping

data acquisition, 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 during data acquisition. Each memory bank is 16 bits wide and 1 Mword (2^{20} words) deep. For standard, non-mapping MCA acquisition, these banks are combined to form a single 32-bit x 1 Mword bank.

For continuous mapping, the host computer must be able to read out and clear an entire buffer for all of the modules in the system in less time then it takes to fill one buffer. The minimum pixel dwell time for continuous mapping operation is defined by the readout speed, the buffer clear time, the number of pixels that can be stored in one buffer and the size of the system. As an example, if the system contains 4 xMAP modules so that the total transferred data for a single buffer is 8 MB [5] and the "burst" transfer speed is 25 MB / s [6], it takes 320 ms to read out a single buffer. If that buffer holds data for 64 pixels [7], the minimum pixel dwell time that allows for continuous mapping is 320 ms / 64 pixels = 5 ms / pixel.

Pixel Advance Modes

The xMAP supports three modes of pixel advance: GATE, SYNC and host control.

GATE

One of the primary methods for advancing the pixel is to use the GATE input as a pixel clock, where the pixel number advances on a defined edge transition of the input signal. In MCA mode, the GATE signal is used to inhibit data acquisition or to coordinate the acquisition with an external system. Using the default gate polarity setting, if GATE is low then data acquisition is disabled. The polarity is controlled with the "input_logic_polarity" acquisition value; to disable data acquisition when the GATE is high, set it to 1.0.

For mapping mode, transitions on the GATE signal can be used to trigger a pixel advance. Like MCA mode, the default setting is to use the high-to-low transition to trigger the pixel advance and, similarly, "input_logic_polarity" can be set to 1.0 if the low-to-high transition is desired instead. Since the GATE signal requires transitions from high-to-low (or low-to-high), there is necessarily

[5] (4 modules) * (1 Mword per module) * (2 bytes per word) = 8 MB

[6] Why do we use 25 MB / s when estimating pixel dwell time? 25 MB / s certainly seems slow if you have spent any time with the PXI specification or the documentation for the National Instruments(tm) PXI Remote Control hardware. The initial theoretical transfer speed of 80 MB / s is initially reduced in half when mapping since we are transferring the data as 16-bit words over a 32-bit interface. As such, half of the data is immediately discarded on the host side when doing these transfers. There is also some overhead involved in reading the data out.

[7] 64 pixels is the maximum in a buffer when the number of MCA bins is set to 4096.

a transition time between states where data acquisition is inhibited. The default setting, controlled with the "gate_ignore" acquisition value, is to ignore the data during the transition period. However if "gate_ignore" is set to 1.0, data acquisition will remain active during the transition.

SYNC

The other primary method of advancing the pixel is to use the SYNC input as a pixel clock. Using this method, the pixel will advance for every `N` positive pulses, where `N` is set using the "sync_count" acquisition value. `N` can range from 1 to 65535. Finally, the pulses must be at least 40 ns wide to be recognized by the xMAP.

Host Control

Lastly, it is possible to advance the pixel directly in Handel using the board operation "mapping_pixel_next". Manually advancing the pixel is slower and does not provide good real-time control and, as such, is suitable only for debugging and evaluation purposes.

GATE/SYNC Signal Distribution

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 the LEMO connector on the xMAP front panel and routes the signal to the other modules on the bus segment using a line on the PXI backplane. Therefore, each bus segment must have its own master module and the input LEMO for each master module must use the same signal source.

Using Mapping Mode: A Walkthrough

Each step is labelled with the mapping modes it supports; no label means the step is relevant to all modes.

Enable mapping mode

```
double mode = 1.0;
CHECK_ERROR(xiaSetAcquisitionValues(-1, "mapping_mode", &mode));
```

When "mapping_mode" is set greater then 0.0, Handel downloads the proper firmware to the xMAP modules. Handel also updates the firmware with any mapping-specific acquisition values. To switch back to normal MCA data acquisition, set "mapping_mode" to 0.0. The xMAP currently supports 3 mapping modes: MCA (1.0), SCA (2.0) and List (3.0).

Set the number of bins in the spectrum (MCA mode)

```
double nBins = 4096.0;
CHECK_ERROR(xiaSetAcquisitionValues(-1, "number_mca_channels", &nBins));
```

The number of bins in the spectrum affects the number of pixels that can fit into each buffer.

Set the number of ROIs and their bounds (SCA mode)

```
double nSCAs = 2.0;
double sca0Lo = 100.0;
double sca0Hi = 1000.0;
double sca1Lo = 2000.0;
double sca1Hi = 2010.0;

CHECK_ERROR(xiaSetAcquisitionValues(-1, "number_of_scas", &nSCAs));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "sca0_lo", &sca0Lo));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "sca0_hi", &sca0Hi));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "sca1_lo", &sca1Lo));
CHECK_ERROR(xiaSetAcquisitionValues(-1, "sca1_hi", &sca1Hi));
```

SCA mapping mode supports a maximum of 64, non-overlapping SCAs.

Set the variant (List-mode)

```
double variant = 1.0;
CHECK_ERROR(xiaSetAcquisitionValues(-1, "list_mode_variant", &variant));
```

List-mode supports three variants: energy plus GATE count (0.0), energy plus SYNC count (1.0) and energy plus clock time (2.0). For a detailed description of the variants, please consult the xMAP User's Manual.

Set the total number of pixels to be acquired in this run (MCA/SCA mode)

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

"num_map_pixels" can also be set to 0.0 if data acquisition should continue indefinitely.

Set the number of pixels per buffer

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

Setting "num_map_pixels_per_buffer" to -1.0 instructs the DSP to automatically use as many pixels as possible given either the MCA size or the number of SCAs. Of course a specific number can be passed in for "num_map_pixels_per_buffer" as well. Unfortunately getting the actualy number of pixels per buffer requires an additional function call:

```
double actualMapPixelsPerBuffer = 0.0;
CHECK_ERROR(xiaGetAcquisitionValues(0, "num_map_pixels_per_buffer",
            &actualMapPixelsPerBuffer));
```

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.

Configure pixel control

At the beginning of the run, the pixel number starts at 0 and is advanced using one of the techniques discussed in the Pixel Advance Modes section. The examples below only show the case where the entire system is on a single bus segment. If the xMAP system spans multiple bus segments then additional master modules are required with the correct pixel advance mode setting. Also note that the master setting is only valid for the first channel in a module; settings on all other channels will be ignored.

GATE

```
#include "handel_constants.h"

double enabled = 1.0;
double pixelMode = XIA_MAPPING_CTL_GATE;

CHECK_ERROR(xiaSetAcquisitionValues(0, "gate_master", &enabled));
CHECK_ERROR(xiaSetAcquisitionValues(0, "pixel_advance_mode", &pixelMode));
```

SYNC

```
#include "handel_constants.h"

double enabled = 1.0;
double pixelMode = XIA_MAPPING_CTL_SYNC;
double nTicksPerPixel = 100.0;

CHECK_ERROR(xiaSetAcquisitionValues(0, "sync_master", &enabled));
CHECK_ERROR(xiaSetAcquisitionValues(0, "pixel_advance_mode", &pixelMode));
CHECK_ERROR(xiaSetAcquisitionValues(0, "sync_count", &nTicksPerPixel));
```

HOST

Manual pixel advance from the host is always available and does not need to be explicitly configured. To advance the pixel, use the following code:

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

Note that the pixel only needs to be advanced once per module. Advancing it once per channel (per module) will actually advance it **4 times** (per module).

Apply the settings

```
int ignored = 0;
CHECK_ERROR(xiaBoardOperation(0, "apply", &ignored));
```

See also the detChans section.

Get buffer length (MCA/SCA mode)

After all of the settings are applied, the xMAP can be queried for the size of a returned buffer. This value can then be used to allocate the appropriate amount of memory.

```
unsigned long bufferLength = 0;
unsigned long *buffer = NULL;

CHECK_ERROR(xiaGetRunData(0, "buffer_len", &bufferLength));

buffer = malloc(bufferLength * sizeof(unsigned long));

if (!buffer) {
    /* Out-of-memory */
}
```

Start the run

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

Monitor the buffer status

Once the run is started, pixels are added to the first buffer ('a') until it is full. To see if the buffer is full, use the following code:

```
unsigned short isFull = 0;

while (isFull == 0) {
    CHECK_ERROR(xiaGetRunData(0, "buffer_full_a", &isFull));

    /* Sleep for a short time here using a routine like Sleep() on win32
     * or usleep() on linux.
     */
}
```

Each module in the system should be polled to determine when all of the modules are ready to be read.

Read full buffer (MCA/SCA mode)

```
/* Assumes that buffer was previously allocated. */
CHECK_ERROR(xiaGetRunData(0, "buffer_a", buffer));
```

Read full list-mode buffer (List-mode)

```
unsigned long bufferLength = 0;
unsigned long *buffer = NULL;

CHECK_ERROR(xiaGetRunData(0, "list_buffer_len_a", &bufferLength));

buffer = malloc(bufferLength * sizeof(unsigned long));

if (!buffer) {
    /* Out-of-memory */
}

CHECK_ERROR(xiaGetRunData(0, "buffer_a", buffer));

free(buffer);
```

Unlike MCA and SCA mapping mode, the length of the list-mode buffer varies slightly from read to read.

Signal that the read has completed

Once a buffer is read, it is important to let the xMAP know that it is available to be filled again. Failure to do this in a timely manner can potentially cause overrun errors.

```
char currentBuffer = 'a';
CHECK_ERROR(xiaBoardOperation(0, "buffer_done", &currentBuffer));
```

Wait for buffer 'b' to fill

With buffer 'a' read and signaled as done, the next step is to wait for buffer 'b'.

```
unsigned short isFull = 0;

while (isFull == 0) {
    CHECK_ERROR(xiaGetRunData(0, "buffer_full_b", &isFull));

    /* Sleep for a short time here using a routine like Sleep() on win32
     * or usleep() on linux.
     */
}
```

Then signal "buffer_done" using the same board operation call as for the 'a' buffer.

Repeat until all the pixels are collected

Continue reading, signaling complete and polling while switching between buffers 'a' and 'b'. You may monitor the for run completion by manually checking the pixel number:

```
unsigned long nMapPixels; /* As set above. */
unsigned long currentPixel;
CHECK_ERROR(xiaGetRunData(0, "current_pixel", &currentPixel));

if (currentPixel >= nMapPixels) {
    /* The run is complete. Break the monitoring loop and continue. */
}
```

Or simply monitor the run status as in normal MCA data acquisition:

```
unsigned long runActive;
CHECK_ERROR(xiaGetRunData(0, "run_active", &runActive));

if (!runActive) {
    /* The run is complete. Break the monitoring loop and continue. */
}
```

Stop the run

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

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

Mapping Tips

This section describes (and reiterates) various tips and techniques to make sure 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 set these values again.

2. **Designate one detChan per module as the "mapping channel"**

   Most of the acquisition values related to mapping mode are module-wide settings and do not need to be set on each channel. As an example, the *Configuration Wizard* in xManager uses the first detChan on each module when generating .ini files for use with Handel.

3. **Cache the mapping buffer length**

   For all modes except for list-mode, the mapping buffer length, retrieved by passing "buffer_len" to xiaGetRunData(), only needs to be read once before the mapping run starts; it will not change once the run is active.

4. **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.

5. **Check for buffer overruns**

   If the per-pixel dwell time is too short for the xMAP to keep up with, it is possible to overrun one of the buffers. 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 is overrun the value of the run data type "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 overrruns 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 system needs some more tuning to run with the dwell time that caused the overrun.

Parsing List-mode data buffers

The list-mode data has some additional complications not found when using MCA or SCA mapping mode. After the buffer header the stream of event data packets is found. In addition to the normal event packets, there are two special record types: end of buffer and rollover. The end of buffer record is provided to verify the buffer integrity and is inserted once at the end of the buffer [8]. The rollover record is used to indicate when the 32-bits of time/tick are exhausted in the normal event data and what the new upper words for those values are. The rollover record will appear once per channel per rollover. When parsing the buffers it is critical that the rollover records be tracked so that the event times can be properly reconstructed.

Below is a heavily annotated code sample that shows how to parse a buffer for a single module and print out the channel, time and MCA bin for each event.

```
/* The returned data from the xMAP is in an array of 16-bit words, but
 * we often need to convert two words into a 32-bit value.
 */
#define MAKE_WORD32(x, i) (unsigned long)((x)[(i)] | \
                           ((unsigned long)(x)[(i) + 1] << 16))

int i;

unsigned long upperTimeWords[4];

unsigned long nEventRecords;
unsigned long nSpecialRecords;
unsigned long totalRecords;
unsigned long j;


/* Assume that the variable buffer is already filled in up to bufferLen
 * with the header and event data.
```

[8] The end of buffer record is the only record that doesn't encode any channel information with it.

```
 */

/* Load the current upper time words from the buffer header. The
 * upper time words for channel 0 begin at offset 72 in the header.
 * And each channel has 12 words of data with it, so we need to increment
 * by that much for each channel.
 */
for (i = 0; i < 4; i++) {
    upperTimeWords[i] = MAKE_WORD32(buffer, 72 + (i * 12));
}

/* Get the number of non-special records from the buffer header. */
nEventRecords = MAKE_WORD32(buffer, 66);

/* Get the number of special records from the buffer header. */
nSpecialRecords = MAKE_WORD32(buffer, 116);

nTotalRecords = nEventRecords + nSpecialRecords;

for (j = 0; j < nTotalRecords; j++) {
    unsigned short record[3];

    /* Copy each event into its own record for further processing. The
     * buffer header is 256 words and each record is 3 words.
     */
    memcpy(&record[0], &buffer[256 + (j * 3)], 3 * sizeof(unsigned short));

    if (record[0] & 0x8000 > 0) {
       /* This is a special record. */

       if (record[0] == 0x8000) {
          /* We have hit the end of the buffer. */
          break;

       } else {
          /* This is a rollover special record. We need to update the
           * upper word for the appropriate channel. The channel that
           * this rollover record describes lives in the lower 4 bits of
           * the first word of the record. The remaining two words are the
           * the new upper time words for that channel.
           */
          int chan = record[0] & 0x000F;
          upperTimeWords[chan] = MAKE_WORD32(record, 1);
       }

    } else {
```

```
        /* Normal event record. The channel is stored in bits 13 and 14 of
         * the first word and the energy bin is stored in the lower 13 bits.
         */
        unsigned short chan = (record[0] & 0x6000) >> 13;
        unsigned short bin = record[0] & 0x1FFF;

        /* Use double and ldexp() to create a 64-bit value. Not all platforms
         * support 64-bit integral types.
         */
        double timestamp = ldexp((double)upperTimeWords[chan], 32) +
                           (double)record[1] +
                           ldexp((double)record[2], 16);

        printf("Timestamp %0.1f, Channel %hu, Bin %hu\n", timestamp,
               chan, bin);
    }
}
```

This code is meant to serve as a rough guideline; it is missing many validation checks based on the values in the buffer header.

Acquisition Values

This section lists the allowed names for use with xiaGetAcquisitionValues() and xiaSetAcquisitionValues().

Filter

peaking_time Peaking time of the energy filter, specified in microseconds.
trigger_threshold Trigger filter threshold, specified in eV.
baseline_threshold Baseline filter threshold, specified in eV.
energy_threshold Energy filter threshold, specified in eV.
gap_time The gap time of the energy filter, reported in microseconds. 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 microseconds.
trigger_gap_time The gap time of the trigger filter, specified in microseconds.
baseline_average The number of samples averaged together for the baseline.
peak_sample_offset{N} Sets the peak sampling time offset constant for decimation `N`, specified in microseconds. This value is optional; setting it will override the defined offset value in the FDD file.

peak_interval_offset{N} Sets the peak interval time offset constant for decimation `N`, specified in microseconds. This value is optional; setting it will override the pre-defined value of the offset in the FDD file.

minimum_gap_time Sets the minimum gap time for energy filter in microseconds. This value will be used for all decimations.

maxwidth Sets the maximum peak width for pile-up inspection in microseconds.

Gain

dynamic_range Energy range corresponding to 40% of the total ADC range, 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 specified calibration energy. This parameter is provided for backwards compatibility with .ini files generated for previous versions of Handel. XIA recommends using "calibration_energy" and "dynamic_range" to set the gain of your xMAP system.

preamp_gain Preamplifier gain specified in mV / keV. This value is synchronized with the gains in the *[detector definitions]* section of the .ini file.

Detector

detector_polarity The input signal polarity, specified as "+", "-", "pos" or "neg". Like "preamp_gain", this value is synchronized with the appropriate entries in the .ini file.

reset_delay The amount of time that the processor should wait after the detector resets before processing the input signal, specified in microseconds.

MCA Data Acquisition

number_mca_channels The number of bins in the MCA spectrum, specified in bins.

mca_bin_width Width of an individual bin in the MCA, specified in eV.

preset_type Set the preset run type. See handel_constants.h for the constants that can be used.

preset_value When a preset run type other then `XIA_PRESET_NONE` is set, this value is either the number of counts or a time (specfied in seconds).

SCA Data Acquisition

number_of_scas Sets the number of SCAs.

****sca{N}_[lo|hi]**** The SCA limit (low or high) for the requested SCA, `N`, specified as a bin number. `N` ranges from 0 to "number_of_scas" - 1.

GATE Control

The settings in this section are applicable to both MCA and Mapping mode runs, note that gate_master must be set even for single module system for GATE signal to be used.

gate_master When 1.0, sets the current module as a GATE master. Only one GATE master is needed per PXI bus segment. To clear this setting, set "gate_master" to 0.0. To switch from "gate_master" to a different master, set the other master to 1.0 and "gate_master" will be automatically cleared. This setting is only valid for the first channel in a module and is ignored for other channels.

gate_ignore Determines if data acquisition should continue or be halted during pixel advance while GATE is asserted.

gate_mode Determines whether the GATE signal, when asserted, will stop the real time clock (0.0) or not (1.0).

input_logic_polarity Sets the polarity of the logic signal connected via the front panel LEMO to either normal (0.0) or inverted (1.0).

Mapping Mode

mapping_mode Toggles between the various mapping modes by switching to the appropriate firmware and downloading the necessary acquisition values. Supported values are: 0.0 = mapping mode disabled, 1.0 = MCA mapping mode, 2.0 = SCA mapping mode, 3.0 = List 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 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. The current, on-board value may be retrieved at any time using xiaGetAcquisitionValues().

sync_master When 1.0, sets the current module as a SYNC master. Only one SYNC master is needed per PXI bus segment. To clear this setting, set "sync_master" to 0.0. To switch from "sync_master" to a different master, set the other master to 1.0 and "sync_master" will be automatically cleared. This setting is only valid for the first channel in a module and is ignored for other channels.

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

lbus_master When 1.0, sets the current module as an LBUS master. Only one LBUS master is needed per PXI bus segment. To clear this setting, set "lbus_master" to 0.0. To switch from "lbus_master" to a different master, set the other master to 1.0 and "lbus_master" will be automatically cleared. This setting is only valid for the first channel in a module and is ignored for other channels.

pixel_advance_mode Sets the pixel advance mode for mapping mode data acquisition. The supported types are listed in handel_constants.h under `XIA_MAPPING_CTL_*`. Manual pixel advance using xiaBoardOperation() is always available and does not need to be set using this acquisition value.

synchronous_run This parameter is used in conjunction with "lbus_master" to enable a synchronous data acquisition run.

buffer_clear_size Specifies how much of the buffer should be cleared when the "board_done" operation is signalled in mapping mode. The default setting of 0.0 causes the entire buffer to be cleared. Any setting greater than 0.0 causes only that number of words to be cleared. XIA recommends for most applications that the default setting of 0.0 is used as miscalculating the correct length may cause data corruption.

list_mode_variant The list mode variant to acquire data with. The allowed variants are energy plus GATE (0.0), energy plus SYNC (1.0) and energy plus clock time (2.0).

Run Data

This section lists the allowed names for use with xiaGetRunData().

The italicized type that follows each name is the type of the argument passed into the `value` parameter of xiaGetRunData(). For scalar values, you will need to pass a pointer to the specified type. For array values, a pointer is specified and the address of the first element should be passed. See xiaGetRunData() for usage examples.

Status

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.

MCA 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.

module_mca (*unsigned long **) Returns the MCA data for all 4 channels flattened into a single array. The MCA length is required to be the same for all 4 channels in the module. The caller is expected to allocate an array of "mca_length" * 4 and pass that in as the `value` parameter when retrieving the buffer data.

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. The caller is expected to allocate an array of length "baseline_length" and pass that in as the `value` parameter when retrieving the baseline data.

Statistics

runtime (*double*) The realtime run statistic, reported in seconds.

realtime (*double*) Alias for "runtime".

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 / second.

module_statistics_2 (*double **) Returns an array containing statistics for the module. The caller is responsible for allocating enough memory for at least 36 elements and passing it in as the `value` parameter. The returned data is stored in the array as follows:

```
[channel 0 realtime,
channel 0 trigger livetime,
channel 0 energy livetime,
channel 0 triggers,
channel 0 MCA events,
channel 0 input count rate,
channel 0 output count rate,
channel 0 underflows,
channel 0 overflows,
...
channel 3 realtime,
channel 3 trigger livetime,
```

```
...
channel 3 overflows]
```

mca_events (*double*) Returns the number of events in the MCA histogram for the specified channel.

total_output_events (*unsigned long*) The total number of events in the current run defined as the sum of the events in the MCA, the underflow events and the overflow events.

events_in_run (*unsigned long*) (DEPRECATED) *Alias for total_output_events.*

module_statistics (*double **) (DEPRECATED) *This statistics value has been replaced by module_statistics_2.*

Returns an array containing statistics for the module. The caller is responsible for allocating enough memory for at least 28 elements and passing it in as the `value` parameter. The returned data is stored in the array as follows:

```
[channel 0 realtime,
channel 0 trigger livetime,
channel 0 energy livetime,
channel 0 triggers,
channel 0 MCA events,
channel 0 input count rate,
channel 0 output count rate,
...
channel 3 realtime,
channel 3 trigger livetime,
...
channel 3 output count rate]
```

SCA Data

max_sca_length (*unsigned short*) Maximum number of SCA elements supported by the system.

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. The caller is expected to allocate an array of length "sca_length" and pass that in as the `value` parameter when retrieving the SCA data.

Mapping Mode

buffer_len (*unsigned long*) (mapping) The size of a mapping buffer. For a given mapping run, this value will remain constant and, therefore, only needs to read once at the start of the run.

buffer_full_a (*unsigned short*) (mapping [9]) The current status of buffer 'a'. If the value is non-zero then the buffer is full.

buffer_full_b (*unsigned short*) (mapping) The current status of buffer 'b'. If the value is non-zero then the buffer is full.

buffer_a (*unsigned long **) (mapping) The data in buffer 'a'. The caller is expected to allocate an array of length "buffer_len" and pass that in as the `value` parameter when retrieving the buffer data.

buffer_b (*unsigned long **) (mapping) The data in buffer 'b'. The caller is expected to allocate an array of length "buffer_len" and pass that in as the `value` parameter when retrieving the buffer data.

current_pixel (*unsigned long*) (mapping) The current pixel number. The number is reset to 0 at the start of each new mapping run.

buffer_overrun (*unsigned short*) (mapping) If non-zero, indicates the the current buffer has overflowed.

list_buffer_len_a (*unsigned long*) (mapping) The length of list buffer 'a'. The length of the list buffer varies between reads.

list_buffer_len_b (*unsigned long*) (mapping) The length of list buffer 'b'. The length of the list buffer varies between reads.

Board Operations

This section lists the allowed names for use with xiaBoardOperation(). Data types are specified as in Run Data. See xiaBoardOperation() for usage examples.

apply (*none* [10]) Apply any recent acquisition value changes to the firmware.

buffer_done (*char*) (mapping [11]) Signal that the specified buffer ('a' or 'b') has been read and may be used for mapping acquisition again. This operation blocks until the buffer is cleared, which takes around 25ms for a 1 Mword buffer. If you need to speed up this operation for small buffer sizes, see acquisition value "buffer_clear_size."

mapping_pixel_next (*none*) (mapping) Advance to the next pixel in a mapping data acquisition run.

buffer_switch (*none*) (mapping) Manually force the firmware to switch to filling the other buffer.

Special Run Types List

This section lists the special runs supported by xiaDoSpecialRun() for xMap applications.

Each special run accepts a different set of parameters via the info array. The Read Data column indicates if corresponding xiaGetSpecialRunData() data are available to be read out.

Name	Read Data?	Type	Info
adc_trace	Yes	double	info[1]: Amount of time to wait between ADC samples in nanosec
baseline_history	Yes	double	info[1]: Amount of time to wait between samples in nanoseconds.
baseline_history	Yes	double	info[1]: Amount of time to wait between samples in nanoseconds.

Below is the table of special run data that can be read by xiaGetSpecialRunData:

Name	Type	Description
adc_trace_length	unsigned long	The length of the ADC trace to be read from the processor in u
adc_trace	unsigned long *	An array containing the data from “adc_trace” special run. The
baseline_history_length	unsigned long	The length of the baseline history to be read from the processor
baseline_history	unsigned long *	An array containing the data from “baseline_history” special ru

Legal

Licenses

Handel

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.

Documentation

Redistribution and use in source (Markdown) and 'compiled' forms (HTML, PDF, LaTeX and so forth) with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code (Markdown) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.
- Redistributions in compiled form (transformed to other DTDs, converted to PDF, PostScript, HTML, LaTeX, RTF and other formats) 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.

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 CONTRACT, 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.

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.

Patents

Patent Notice