Handel Programmer's Guide - microDXP

Contents

Intended Audience

This document is intended for users of the XIA microDXP hardware who would like to interface to it using the Handel driver library. This document assumes that users of the Handel driver library are familiar with the C programming language.

Conventions Used in this Document

We use `fixed width style` to indicate source code, variables, and constants.

`CHECK_ERROR` is a placeholder for user-defined error handling. See the sample code for an example of how to implement such error handling.

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. The following headers must be included:

- *handel.h* Defines and imports all available Handel routines.
- *md_generic.h* Contains the constants for setting logging levels.
- *handel_errors.h* Error status constants including `XIA_SUCCESS`.
- *handel_constants.h* Contains the constants used for Handel calls.

Error Codes

A good programming practice with Handel is to compare the returned status value with `XIA_SUCCESS` and then deal with 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. Details on using this system are found in the *Handel API Manual*.

.INI Files

The final piece of information external to the actual Handel source code is the initialization file. The simplest way to supply the hardware configuration to Handel is to use the supplied microDXP initialization file (udxp_std.ini). The .ini file must specify the USB2 device or COM port used for the microDXP. To do this, edit the line:

```
device_number = 0
```

or:

```
com_port = 1              * Windows
device_file = /dev/ttyS0 * Linux
```

and set it to the appropriate value. USB2 devices start at 0. For serial ports, 1 represents COM1 and 2 represents COM2 and so on. Linux serial ports use device files instead of COM port numbers. The rest of the .ini file will have no effect since all of the other settings are read from the microDXP itself. For more details, see the comments in udxp_std.ini.

Example Code

Included with this document is a file called hqsg-udxp.c that illustrates all of the lessons presented in this tutorial. hqsg-udxp.c is sample code that initializes Handel, configures the microDXP hardware, starts a run, stops a run, reads out the MCA spectrum and saves the current configuration to the hardware.

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 set up preferably at the beginning of the application.

```
/* Direct logging to a local file, the different log levels can be found
 * in md_generic.h.
 */
xiaSetLogLevel(MD_DEBUG);
xiaSetLogOutput("handel.log");
```

To prevent memory leak and release the file handle, the log file needs to be closed at the end of application.

```
xiaCloseLog();
```

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(). 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", "udxp_std.ini");
CHECK_ERROR(status);
```

The above example has the exact same behavior as

```
int status;
status = xiaInit("udxp_std.ini");
CHECK_ERROR(status);
```

If the configuration file to be used is known ahead of time, then calling `xiaInit()` is the preferred method for initializing the library.

Starting the System

Once the initialization task has been completed, the next step is to start the system. This process performs several operations including validating the hardware information supplied in the initialization file and verifying the specified communication interface (RS-232, for the microDXP). Calling xiaStartSystem() is straightforward:

```
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, reading out the MCA and saving parameter information.

Configuring Data Acquisition

Unlike other XIA hardware, the microDXP stores all of its firmware and operating parameters on-board in non-volatile memory. Older versions of the microDXP can store up to 3 FPGA configurations (FiPPIs), each corresponding to a separate range of peaking times. Within each FiPPI peaking time range, 5 specific peaking times and their associated configurations can be saved to a PARameter SET (PARSET) in memory. For a microDXP configured with 3 FiPPIs, this yields a total of 15 different peaking time configurations.

The latest versions of the microDXP have an updated FPGA design and can store the entire range of peaking times in a single FiPPI. The updated FiPPI stores 24 different peaking time configurations.

In addition to the PARSETs, there are two other sets: GENeral SETs (GENSET) and GLOBal SETs (GLOBSET). The microDXP contains a total of 5 GENSETs. The GENSETs are not tied to a specific FiPPI and the parameters in this set are gain and spectrum related. There is a single GLOBSET which contain values specific to the detector preamplifier, debugging and run control.

A key component of configuring the microDXP is choosing values for the various PARSETs and GENSETs, and saving these values to non-volatile memory. As

illustrated in Appendix A, B and C, not all of the PARSET/GENSET parameters map directly to acquisition values in Handel. The fact is you only need to modify a small subset of the total number of available parameters in order to configure the microDXP for your system.

Once the microDXP has been properly configured, your application should only need to swap between the different PARSET/GENSET entries. Furthermore, on power-up, the microDXP remembers which PARSET/GENSET was used last and loads it into memory so there is no need to track PARSET/GENSET in your application.

Now that we have a solid understanding of what the microDXP stores in memory, let's step through an example of how to configure a microDXP. In this example, we want to configure the microDXP with the following settings:

- FiPPI 0
- PARSET 0
- GENSET 0
- Medium bin width granularity
- 4k spectrum
- Trigger threshold of 20
- Positive polarity detector
- Base gain of 32768
- 100 microsecond reset interval for the preamplifier

Select the FiPPI

On the latest versions of the microDXP, there is only a single FiPPI and there is never a need to switch. Attempting to set the FiPPI to anything besides the default value of 0 will result in an error. On older versions of the microDXP, the FiPPI can be selected as follows:

```
int status;
ushort numberFippis = 0;
double fippi = 0.0;
unsigned short fippiMem = AV_MEM_FIPPI;

/* Read out supported number of FiPPI first. */
status = xiaBoardOperation(0, "get_number_of_fippis", (void *)&numberFippis);
CHECK_ERROR(status);

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

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

Setting Acquisition Values

The next step is to set all of the acquisition values. It is important after setting an acquisition value that you "apply" the new value with a call to xiaBoardOperation().

```
double binWidth = 2.0;
double nMCA = 4096.0;
double threshold = 20.0;
double gain = 32768.0;
double polarity = 1.0;
double resetInt = 100.0;
unsigned short parsetAndGenset = AV_MEM_PARSET | AV_MEM_GENSET;

status = xiaSetAcquisitionValues(0, "mca_bin_width", (void *)&binWidth);
CHECK_ERROR(status);

status = xiaSetAcquisitionValues(0, "number_mca_channels", (void *)&nMCA);
CHECK_ERROR(status);

status = xiaSetAcquisitionValues(0, "trigger_threshold", (void *)&threshold);
CHECK_ERROR(status);

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

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

status = xiaSetAcquisitionValues(0, "preamp_value", (void *)&resetInt);
CHECK_ERROR(status);

/* Need to call "apply" after setting acquisition values. */
status = xiaBoardOperation(0, "apply", (void*)&parsetAndGenset);
CHECK_ERROR(status);
```

Note that the apply operation specifies which section of the microDXP's memory needs to be applied. When switching to a different FiPPI you need to call xiaBoardOperation("apply") with the `AV_MEM_FIPPI` flag; when you modify PARSET, GENSET or GLOBSET acquisition values you need to use `AV_MEM_PARSET`, `AV_MEM_GENSET`, `AV_MEM_GLOB` or some combination therein.

Save the GENSET/PARSET

Now that we have configured the device and are happy with our settings, we want to save the parameters so that we can return to this state again.

```
unsigned short genset = 0;
unsigned short parset = 0;

status = xiaBoardOperation(0, "save_genset", (void *)&genset);
CHECK_ERROR(status);

status = xiaBoardOperation(0, "save_parset", (void *)&parset);
CHECK_ERROR(status);
```

Selecting a Peaking Time

In the previous code we simply saved our configuration to PARSET 0 for FiPPI 0 without any explanation of what PARSET 0 corresponds to. Each of the PARSETs are equal to a different peaking time. To discover what peaking times are available for the selected FiPPI, use the following code:

```
int status;
int i;
double *peakingTimes = NULL;

/* Read out number of peaking times to pre-allocate peaking time array */
status = xiaBoardOperation(0, "get_number_pt_per_fippi", &numberPeakingTimes);
CHECK_ERROR(status);

peakingTimes = (double *)malloc(numberPeakingTimes * sizeof(double));
CHECK_MEM(peakingTimes);

status = xiaBoardOperation(0, "get_current_peaking_times", peakingTimes);
CHECK_ERROR(status);

/* Print out the current peaking times */
for (i = 0; i < numberPeakingTimes; i++) {
    printf("peaking time %d = %lf\n", i, peakingTimes[i]);
}

free(peakingTimes);
```

where i corresponds to the PARSET responsible for that peaking time. To switch to peaking time/PARSET i, simply do:

```
double parset = (double)i;
status = xiaSetAcquisitionValues(0, "parset", (void *)&parset);
CHECK_ERROR(status);
```

In some applications, it may be useful to cache all the available peaking times for the board, so that additional readout can be skipped when user select a different FiPPI on the board.

```
/* Read out number of fippis to pre-allocate peaking time array */
status = xiaBoardOperation(0, "get_number_of_fippis", &numberFippis);
CHECK_ERROR(status);

peakingTimes = (double *)malloc(numberPeakingTimes * numberFippis * sizeof(double));
CHECK_MEM(peakingTimes);

status = xiaBoardOperation(0, "get_peaking_times", peakingTimes);
CHECK_ERROR(status);

/* Print out the current peaking times */
for (i = 0; i < numberPeakingTimes * numberFippis; i++) {
    printf("peaking time %d = %lf\n", i, peakingTimes[i]);
}

free(peakingTimes);
```

Controlling the MCA

Once the microDXP is properly configured, it is ready to begin data acquisition tasks. This section discusses starting a run, stopping a run, reading out the MCA spectrum and, lastly, configuring the microDXP for preset length runs.

Starting and Stopping a Run

The Handel interface to starting and stopping the run are two simple routines: xiaStartRun() and xiaStopRun(). Both routines require a detector channel number (like xiaSetAcquisitionValues()) as their first argument, while `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;
unsigned short clearMCA = 0;

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

/* Windows API call. Use your platform's sleep API to wait.
 */
Sleep((DWORD)5000);

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

> For historical reasons, Handel and the microDXP RS-232 command for starting a run have a different idea of how to interpret the clear MCA value. The RS-232 command uses 0 to mean "resume run" and 1 to mean "clear the MCA". Handel uses 0 to mean "clear the MCA" and 1 to mean "resume run".

Reading out the MCA Spectrum

Assuming that we are still running with FiPPI 0 and the PARSET 0 configuration from the previous section, we know that our MCA spectrum length is 4096. In order to reduce the number of bytes that have to be sent across the serial port connection, you can request either 1, 2, or 3 bytes per bin. The default setting in Handel is 3 bytes per bin, which is the same as the raw value stored in the DSP's memory. If you want to use 3 bytes per bin then you do not have to change anything. If you want to only return a single byte per bin, then use the following code:

```
double bytesPerBin = 1.0;
status = xiaSetAcquisitionValues(0, "bytes_per_bin", (void *)&bytesPerBin);
CHECK_ERROR(status);
```

> If the number of counts in a bin exceeds the requested bytes per bin, the microDXP does not return an error. For example, if there are 0xADCDEF counts in a bin and you read out the MCA spectrum with bytes per bin set to 1, that bin will return 0xEF!

With the bytes per bin configured correctly, we are now ready to read out the MCA spectrum.

```
unsigned long mca[4096];
status = xiaGetRunData(0, "mca", (void*)mca);
CHECK_ERROR(status);
```

Preset Length runs

The microDXP supports preset runs, which allow you to specify that a run stop automatically after a certain amount of time has passed or other criteria have been met. The four types of preset runs are fixed livetime, fixed realtime, fixed output counts and fixed input counts. A fixed livetime run will execute until the specified amount of livetime has elapsed. Similarly, a fixed realtime run will execute until the specified amount of realtime has elapsed. The fixed input and output count runs continue until the requested number of counts have occured.

The following is an example of setting a fixed realtime run for 5 seconds, including how to poll the device waiting for the run to complete:

```
int status;
double realtime = 5.0;
double realtimeType = XIA_PRESET_FIXED_REALTIME;
double presetData[2];
unsigned short clearMCA = 0;
unsigned short runActive;

presetData[0] = realtimeType;
presetData[1] = realtime;
status = xiaBoardOperation(0, "set_preset", (void*)presetData);
CHECK_ERROR(status);

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

do {
    Sleep((DWORD)1);
    status = xiaGetRunData(0, "run_active", (void*)&runActive);
    CHECK_ERROR(status);
} while (runActive);

/* Once the run is no longer active, we know that the preset run has
 * completed and that it is safe to stop the run.
 */
status = xiaStopRun(0);
CHECK_ERROR(status);

/* Read out and process the spectrum. */
```

Special Runs

The microDXP supports several special run types for diagnosis and custom operations. Their parameter and data type is outlined in Appendix D. The following is an example to do a "snapshot" special run, then read out the resulting data. The sample code hqsg-udxp-snapshot.c contains a complete operation.

```
int status

unsigned long mca_length;
unsigned long *mca = NULL;
double clearspectrum[1] = {0.};

status = xiaGetSpecialRunData(0, "snapshot_mca_length", &mca_length);
mca = malloc(mca_length * sizeof(unsigned long));
```

```
/* start a run and take snapshots */
status = xiaStartRun(-1, 0);
CHECK_ERROR(status);

status = xiaDoSpecialRun(0, "snapshot", &clearspectrum);
CHECK_ERROR(status);

status = xiaGetSpecialRunData(0, "snapshot_mca", mca);
CHECK_ERROR(status);

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

free(mca);
```

Firmware Upgrades

Upgrades to the microDXP firmware still requires custom built tools in the current Handel release. Futuer versions will support functions to handle XUP file format.

Cleaning Up

Before exiting Handel, call xiaExit() to safely shutdown the serial port driver:

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

Appendix A – Acquisition Values List

Below is a list of all of the supported acquisition values for the microDXP. All of the acquisition values are of type *double*.

Filter

parset The current PARSET.
genset The current GENSET.
fippi The current FiPPI.

clock_speed The digitizing clock in MHz. This value will be rounded to the nearest setting supported by the hardware, which is either DSPCLK, DSPCLK/2, DSPCLK/4 or DSPCLK/8. Not all selections are available on all hardware.

energy_gap_time The gap time of the energy filter, specified in microseconds.

trigger_peak_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_length The number of samples averaged together for the baseline filter.

trigger_threshold Trigger filter threshold in arbitrary units.

baseline_threshold Baseline filter threshold in arbitrary units.

energy_threshold Energy filter threshold in arbitrary units.

peak_interval_offset The peak interval specified as an offset from the peaking time and gap time, specified in microseconds. Effectively sets PEAKINT = SLOWLEN + SLOWGAP + peak_interval_offset. Added in v1.2.2.

peak_sample_offset Energy filter sampling time measured backward from the peaking time and gap time, specified in µs.Effectively sets PEAKSAM = SLOWLEN + SLOWGAP - peak_sample_offset. Added in v1.2.2.

max_width The value of MAXWIDTH, specified in microseconds.

peak_mode The value of PEAKINT. Sets the value of PEAKMODE to "Peak-Sensing" (PEAKMODE=0) or "Peak-Sampling" (PEAKMODE=1). Added in v1.2.2.

[**Deprecated**] **peak_interval** The value of PEAKINT, specified in microseconds. Deprecated in v1.2.2, use **peak_interval_offset** instead.

[**Deprecated**] **peak_sample** The value of PEAKSAM, specified in µs. Deprecated in v1.2.2, use **peak_sample_offset** instead.

Detector

polarity The detector preamplifier polarity, where the allowed values are 0 = negative and 1 = positive.

preamp_value Either the reset interval, for reset-type preamplifiers, or the decay time, for RC feedback-type detectors. The reset interval is specified in microseconds and the decay time is specified in terms of the digitization clock period.

Gain

gain The base gain in arbitrary units.

gain_trim Adjusts the base gain per PARSET, specified in arbitrary units.

Preset Run Control

preset_type Set the preset run type. See handel_constants.h for the constants that can be used. The supported preset type for microDXP are:

- `XIA_PRESET_FIXED_REALTIME`
- `XIA_PRESET_FIXED_LIVETIME`
- `XIA_PRESET_FIXED_TRIGGERS`
- `XIA_PRESET_FIXED_EVENTS`

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).

MCA Data Acquisition

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

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

bytes_per_bin The number of bytes returned per bin when reading out the MCA spectrum. Can be either 1, 2 or 3 bytes.

adc_trace_wait When acquiring an ADC trace for readout, the amount of time to wait between ADC samples, specified in microseconds.

auto_adjust_offset Whether the DAC will remain static until next power cycle or re-adjusted whenever analog gain or other settings are changed. (0: static, 1: auto adjusted).

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.

Appendix B – Run Data List

These are the different types of run data that can be read using xiaGetRunData(). The C type of the run data is printed in *italics* after the name.

Status

run_active (*unsigned long*) The current state of the processor. If the value is non-zero then a run is currently active on the channel.

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.

baseline_length (*unsigned long*) The current size of the baseline histogram buffer.

baseline (*unsigned long **) The baseline histogram.

Statistics

energy_livetime (*double*) The calculated energy filter livetime, reported in seconds.

trigger_livetime (*double*) The calculated trigger livetime, reported in seconds.

runtime (*double*) The runtime, 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.

events_in_run (*unsigned long*) The total number of events in the current run, implemented as the sum of the MCA bins.

triggers (*unsigned long*) The number of input triggers in the current run.

module_statistics_2 (*double **) Returns an array containing statistics for the module. The caller is responsible for allocating enough memory for at least 9 elements and passing it in as the `value` parameter. The returned data is stored in the array as follows: [runtime, trigger_livetime, energy_livetime, triggers, events, icr, ocr, underflows, overflows]

[Deprecated] all_statistics (*double[6]*) Returns an array of the six statistics available for the microDXP: livetime, runtime, triggers, events in run, input count rate and output count rate. **module_statistics_2** was introduced to provide support for additional statistics data as a replacement for this run data.

SCA

max_sca_length (*unsigned short*) Maximum number of SCA elements supported by the system. Equivalent to the **number_of_scas** acquisition value.

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.

Appendix C – Board Operations List

The allowed board operations for the microDXP, accessed via. xiaBoardOperation(). The C type of the `value` parameter is printed in *italics* after the name.

Note that the board operations **get_number_of_fippis**, **get_peaking_time_ranges**, **get_current_peaking_times** and **get_peaking_times** utilize a command that stops any active run on the current board.

get_serial_number (*char[16]*) Get the microDXP board's serial number.

get_peaking_time_ranges (*double **) Returns an array of doubles with size (# of FiPPIs * 2). For each FiPPI the shortest peaking time and longest peaking time are returned, in that order.

get_number_of_fippis (*unsigned short*) Gets the number of FiPPIs that are on the board.

get_number_pt_per_fippi (*unsigned short*) Gets the number of peaking times in each FiPPI. 5 or 24, depending on the variant.

get_current_peaking_times (*double[N]*) Get the current peaking times for the selected FiPPI, where the peaking time at index i in the returned list corresponds to PARSET i for the selected FiPPI. N is the number of peaking times per FiPPI, retrieved via board operation *get_number_pt_per_fippi*.

get_peaking_times (*double[N]*) Get array of all peaking times supported by the board, in the order of peaking times for each PARSET, often used when the peaking times need to be cached by the application. N can be derived from *get_number_of_fippis* multiplied by *get_number_pt_per_fippi*.

get_temperature (*double*) Returns the current temperature of the board, accurate to 1/16th of a degree of Celsius.

apply (*none*)[1] Applies the current DSP parameter settings to the hardware. This should be done after modifying any acquisition values.

save_parset (*unsigned short*) Saves the current DSP parameter settings to the specified PARSET.

save_genset (*unsigned short*) Saves the current DSP parameter settings to the specified GENSET.

[Deprecated] set_preset (*double[2]*) Configure a preset run by passing in the preset type and value. The allowed types, defined in handel_constants.h are:

- `XIA_PRESET_FIXED_REALTIME`
- `XIA_PRESET_FIXED_LIVETIME`
- `XIA_PRESET_FIXED_TRIGGERS`
- `XIA_PRESET_FIXED_EVENTS`

The values are defined as time in seconds, for the time based runs and counts for the other types.

The acquisition values *preset_type and* preset_value have been implemented to provide r/w access to preset run properties, and should be used instead.

get_board_info (*unsigned char[26]*) Returns the array of board information listed in command 0x49 of the RS-232 Command Reference.

The returned data is stored in the array as follows, each line representing a byte. Although the pre-allocated size is fixed, the returned content is dependent on the number of FiPPIs. For products with a single FiPPI, unused bytes can be ignored.

0. PIC Code Variant
1. PIC Code Major Version
2. PIC Code Minor Version
3. DSP Code Variant
4. DSP Code Major Version
5. DSP Code Minor Version
6. DSP Clock Speed
7. Clock Enable Register
8. Number of FiPPIs
9. Gain Mode
10. Gain (mantissa low byte)
11. Gain (mantissa high byte)
12. Gain (exponent)
13. Nyquist Filter
14. ADC Speed Grade
15. FPGA Speed
16. Analog Power Supply
17. FiPPI 0 Decimation
18. FiPPI 0 Version
19. FiPPI 0 Variant

Bytes 20-25 repeat the FiPPI pattern for 1 and 2, if available.

get_usb_version (*unsigned long*) Returns the USB firmware version number packed into an unsigned long as follows: [3]Major [2]Minor [1]Reserved [0]Build. Offsets refer to byte indexes in the unsigned long. For example,

the following expression may be used to get the major version: `(value >> 24) & 0xFF`.

This operation is only supported for microDXP firmware Rev H or later and the UltraLo.

get_preamp_type (*unsigned short*) Returns the current preamplifier type, where 0 = reset and 1 = RC feedback.

set_xup_backup_path (*char **) Sets the path where XUP backups are written.

get_hardware_status (*unsigned char[5]*) Returns the array of status information listed in command 0x4B of the RS-232 Command Reference.

passthrough (*{byte*, int*, byte*, int*}*) Pass a command through to a UART attached to the processor. This command requires custom hardware and firmware and is not supported on all variants. If the variant does not implement the custom command, xiaBoardOperation will return a Handel error code.

The `value` type is *void***, an array pointing to the following elements:

- *byte** send: an array of bytes to send with the command.
- *int** send length: number of bytes in the send array.
- *byte** receive: an array of bytes to return the command response.
- *int** receive length: number of bytes to read in the command response.

Sample usage:

```
byte send[32] = {1, 2, 3};
int send_len = sizeof(send) / sizeof(send[0]);
byte receive[32] = {0};
int receive_len = sizeof(receive) / sizeof(receive[0]);
void* value[4] = {send, &send_len, receive, &receive_len};

int status = xiaBoardOperation(0, "passthrough", value);
CHECK_ERROR(status);

/* Process receive... */
```

Appendix D – Special Run Types List

This section lists the special runs supported by xiaDoSpecialRun for microDXP 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
adjust_offsets	N/A	double	info[0]: ADC Offset to adjust acceptable range 0-16383 (14-bit ADC)
adc_trace	Yes	double	info[1]: Amount of time to wait between ADC samples in nanosecond
snapshot	Yes	double	info[0]: Option to clear spectrum memory 0: no action 1: clear memo

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
adc_trace	unsigned long *	An array containing the data from "adc_trace" special run. T
snapshot_mca_length	unsigned long	The length of the snapshot_mca buffer to be read from the p
snapshot_mca	unsigned long *	An array containing the snapshot MCA data from "snapshot"
snapshot_statistics_length	unsigned long	The length of snapshot statistics, as number of doubles.
snapshot_statistics	double *	An array containing snapshot statistics from "snapshot" spec
snapshot_sca_length	unsigned long	The length of the snapshot_sca buffer, as number of doubles.
snapshot_sca	double *	An array containing the snapshot SCA data from "snapshot"

Legal

Licenses

Handel

Documentation

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