Handel API Manual

Contents

Handel: *XIA Hardware Description Layer*

XIA LLC

http://www.xia.com

support@xia.com

Introduction

This document is intended to aid the data collection programmer in developing software that controls and reads out data from all XIA x-ray processors. Handel provides a high level interface to the hardware, requiring as little knowledge about the hardware as is reasonable by the end-user.

The Handel library is written in ANSI C and has no dependencies other than the C standard library and hardware device drivers. XIA provides device drivers for Windows. Device driver availability on other platforms varies and depends in part on end user implementation. See Platform Support.

The organization of this document is as follows:

- Technical Support provides links and addresses for software updates and contacting XIA.
- Platform Support describes the operating system platforms for which Handel is released and built.
- Acquiring Handel gets you started with acquiring the libraries and/or source code.
- Building Handel describes the build environment.
- Header Files lists the headers you should include in your C/C++ program.
- Terms introduces some terms that are used throughout this document.
- Calling Conventions describes the calling conventions used by the routines that are included with the host software release.
- API describes the Handel library routines.
- Files shows the layers of the Handel libraries.
- INI Files specifies the configuration file format.

Technical Support

Software Updates

XIA updates Handel in the course of product development and in response to bug reports from customers. The latest source distribution and Windows binaries, as well as several back releases, are available product-specific variants at the Handel Release page on the XIA Support web site.

You can also obtain Handel headers and binaries from the SDK folder in your ProSpect installation.

Email Support

If you have any questions or encounter a bug with the library, please contact XIA by sending an email to support@xia.com with the following information:

1. Your name and organization
2. XIA hardware being used
3. Operating system and version
4. Description of the problem and steps to re-create
5. Supporting data in the form of Handel log files or relevant plots

6. Version of the Handel library (see the top of the log file or check the DLL file properties)
7. Other information you find relevant

License

This document is Copyright 2002-2017 XIA LLC, All rights reserved.

Information furnished by XIA is believed to be accurate and reliable. However, no responsibility is assumed by XIA for its use, nor for 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.

A majority of the Handel source code is available under a BSD-style license. The complete text of this license is available at: http://www.opensource.org/licenses/bsd-license.php. The main points of the license are summarized as follows:

- Redistributions of source code must retain the copyright notice provided in the source code, this list of conditions and the disclaimer provided in the source code.
- Redistributions in binary form must reproduce the copyright notice, this list of conditions and the disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of X-ray Instrumentation Associates nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Certain sub-systems in Handel are unable to be released under the BSD-style license. As of this writing, the microDXP drivers (corresponding to the libraries udxp, udxps, udxp_psl and udxps_psl) and the serial port drivers on Windows (corresponding to the seriallib library) are only available in binary form.

Additionally, third-party modules required by Handel are available in binary format only. The 3rd-party modules included in the distribution are: cdrvdl32.dll, cdrvhf32.dll, cdrvxf32.dll, and xw.dll.

Platform Support

The Handel library is programmed in C99 using only the standard library and compiles on various platforms using common C compilers such as msvc, gcc, and clang.

Support for particular platforms depends on availability of device drivers and may require end user implementation to adapt device drivers to the machine-dependent (MD) layer of Handel.

XIA provides full support for all x-ray products on Windows, including device drivers, graphical configuration and control applications, and technical support.

The Handel source release includes a Linux MD implementation for USB (built on libusb-0.1), EPP (sys/io), and serial interfaces (termios). USB- and EPP-based Mercury and Saturn spectrometers are used in Linux-based experiments around the world via the open source EPICS DXP module. As of this writing, the Linux serial interface is experimental but passes all internal tests at XIA using microDXP Rev H. While PLX drivers are available for Linux from National Instruments, installation support and MD hooks implementation must be supplied by the user.

Use of other platforms requires a full implementation of the MD layer supplied by the user. Given knowledge of the device driver interface for a given device, implementing the MD layer is a straightforward process of writing open, close, read, and write hooks and a bit of bookkeeping to store and map device handles to Handel modules. See md_win32.c and md_linux.c in the source distribution for reference implementations. Per the Handel license, users are welcome to keeep their implementations closed source. If you do wish to contribute device driver support to the official Handel release, please email support@xia.com.

> Note to microDXP users: While the Handel library provides high level control to nearly all settings and acquisition control, given the suitability of the microDXP for embedded applications, it can also be controlled via RS-232 with no additional software requirements. See the microDXP manuals for more information about the microDXP architecture and a downloadable RS-232 command specification.

Acquiring Handel

If you have XIA configuration and control software such as ProSpect installed, you already have the libraries and you may already have the source. Go to `C:\Program Files (x86)\XIA` and look for an `sdk` or `lib` directory under the application directory.

Otherwise, you can download ProSpect with compiled Windows libraries from our web site. Visit http://xia.com/products.html, click through the XIA product you're working with, and follow the Downloads link.

To download the complete Handel source package, visit Handel Release page and download the latest version for your product.

Linux

XIA does not provide a standalone binary distribution for users running on Linux OS's. Linux users will need to follow the instructions in Building Handel for compiling the libraries from source code.

Building Handel

Most Windows users will simply use the compiled libraries from our web site. However, for Linux or other platforms, you will need to compile from source. This section describes the build requirements and environment.

Requirements

Windows

Handel is developed on Windows 10 and tested on Windows 7 SP1. It is expected to work back through Vista and XP SP3. Handel may run on earlier versions of Windows, but we do not officially support this.

The Windows versions of Handel are compiled using Microsoft's `cl` compiler and `link` linker from Visual C++ 2010 for both x86 and x64 architectures.

Linux

We don't maintain an up-to-date list of supported Linux distributions, but most modern ones should work fine. Handel is compiled and tested Ubuntu 14.04. The build system is configured to work with the standard GNU make and gcc toolchain.

Build Environment

The build system is SCons with extension from swtoolkit. The SCons build supports Windows and Linux. While we do provide the SCons files in the source package, most customers prefer to work with their own Makefiles. If you wish to use the SCons build, consult README.linux.txt in the source distribution for more details on setting up SCons and swtoolkit.

A major goal of XIA's build system is to allow custom Handel configurations to be built. For instance, it is possible to build a Mercury-only version of Handel or a version that only supports PXI products.

By default (calling the swtoolkit hammer wrapper with no flags), the build system builds all of the protocols and products supported on the host platform

for all architectures (x86 and x64). To specify different products, protocols and products, invoke make using the following syntax:

`hammer [--[no-]<product>] [--[no-]<protocol>] [<library>]`

See the top section of the file main.scons for a list of supported products and protocols. Call `hammer --help` for a list of libraries that can be built.

When a protocol is excluded from the build, all of the products that use the protocol are also removed from the build. For instance, if you invoke hammer with the option –no-serial, both of the microDXP drivers (udxp and udxps) are removed from the system as well as the serial port drivers.

Header Files

> This section assumes you are using the Handel libraries in a C or C++ programming environment.

handel.h is the main header file for the Handel libraries and should be included in every source file that calls a Handel library function. This header contains prototypes for all of the exported Handel functions and some useful constants.

handel_errors.h contains all of the error codes and a short comment describing the meaning of each error code. The constants in this file can be used to create descriptive error handling in source code. For instance:

```
#include "handel.h"
#include "handel_errors.h"

int status;
status = xiaInitHandel();

if (status == XIA_XERXES) {
    printf("Xerxes returned an error!\n");
}
else if (status == XIA_NOMEM) {
    printf("Out-of-memory!\n");
}
else if (status == XIA_SUCCESS) {
    printf("Success!\n");
}
```

If you choose not to interpret all possible returned error values, XIA highly recommends at least checking the returned status against the constant #XIA_SUCCESS. All of the routines in Handel follow the standard of returning an integer value indicating the success or failure of the routine.

```
int status;
status = xiaInitHandel();
```

```
if (status != XIA_SUCCESS) {
    printf("ERROR = %d\n", status);
    abort();
}

status = xiaStartSystem();
if (status != XIA_SUCCESS) {
    printf("ERROR = %d\n", status);
    abort();
}
```

> Handel may not be in a consistent state if a routine fails. If execution continues with the next Handel call, the behavior will be indeterminate and may result in unexpected application errors

handeldef.h defines `HANDEL_IMPORT`, which is used in handel.h to define the API routines. This file does not need to be included directly since handel.h includes it, but it does need to be in the same directory as your other Handel header files.

handel_generic.h defines constants like `MAXALIAS_LEN`, used for allocating memory to pass to Handel routines.

Include md_generic.h if you are going to use the constants associated with xiaSetLogLevel(): `MD_ERROR`, `MD_WARNING`, `MD_INFO`, and `MD_DEBUG`.

Linking

Handel is built as a shared library. Both the ProSpect SDK and the Handel binary distribution provide the necessary import libraries and headers for linking the libraries directly into your application. Similarly, building Handel using XIA's scons build ensures that the necessary import libraries will be created.

Terms

detChan A global value unique to each channel in the system. The detChan value is used to reference a channel independent of the module it is associated with. Furthermore, detChans may be grouped into detChan sets that are also given a unique value and may be used with most routines that accept a detChan as an argument. Additionally, detChan sets may reference both single detChans and other detChan sets, provided that none of the detChans refer back to the original set. Handel checks the integrity of the detChans and warns the user if it detects an infinite loop.

Driver Libraries Each XIA product requires two driver libraries: one that interfaces to the hardware at a low level (device-driver) and one that provides the device specific implementations. (PSL driver).

DSP This is the on-board digital signal processor (DSP) that controls the spectrometer functions and some general run functions. The DSP also contains memory for storing spectra, diagnostics, control words and an internal work area. The host computer must download a program to the DSP prior to starting tasks on the XIA processor. This device is complicated and XIA provides programming manuals for custom applications. A "standard" DSP program is provided with all XIA processors.

Firmware Definition Database (FDD) XIA will release firmware in special files created for Handel, called FDD files. Each file will contain all of the Firmware code required to configure and run an XIA processor. Special firmware will be distributed as FDDs separate from the general distribution.

FiPPI This is the field programmable gate array (FPGA) in which the Filter, Peak detection, Pileup Inspection logic is implemented. Like the DSP, a configuration file must be downloaded to the FiPPI before it can function.

Firmware Firmware refers to all FPGA(s) and DSP(s) on the XIA processors. When power is initially applied to an XIA processor, it has only enough firmware loaded to handle communication via the hardware interface; the rest of the firmware must be downloaded to the XIA processor prior to starting tasks.

System Chip/Memory Manager This is another FPGA that is present on some XIA processors. Its function is to control miscellaneous chips on the processor such as SRAM and FIREWIRE interfaces. As XIA processors evolve additional FPGAs may be added.

Hardware interface This is the method that each XIA module uses to communicate with the host computer. We currently support USB2 and serial ports on most Linux and Windows operating systems. PLX is supported on Windows. USB1 and EPP support remain in the codebase but are no longer tested by XIA.

Host This is the computer on which the data collection program runs and collects data via some hardware interface to the XIA device.

.ini File The .ini file is used by Handel to initialize the system, several options for initializing are available and discussed later in this document.

Module XIA product with at least one channel associated with it.

Product-specific Layer (PSL) A set of libraries that libraries contain the individual logic associated with each product. Host software should never call these libraries directly.

Read Transfer data from the XIA processor to the host computer.

Write Transfer data from the host computer to the XIA processor.

Calling Conventions

Language Interface

Handel is supported for calls from other C programs or libraries. XIA does not officially support other language interfaces such as Visual Basic or Fortran, but we provide sample C# interop definitions in the examples folder of the source distribution.

Integer Functions

If successful, all Handel routines return #XIA_SUCCESS, otherwise they return a status code indicating a problem (see handel_errors.h for error codes). In addition, all routines that sense an error print a message to either stdout (the default setting) or to the stream indicated by a call to xiaSetLogOutput(). This has the effect of producing a trace-back for identifying where a problem occurred.

Word Size

Handel APIs use standard integer types, not fixed width integer types. When interfacing to the driver library from languages other than C, the user must be careful to match the length of variable types across compilers.

For the x86 architecture, the word size is as follows:

- short/unsigned short = 2 bytes
- int/unsigned int = 4 bytes long/unsigned
- long = 4 bytes

DSP parameters are of length 2 bytes.

Searching For Files

Handel follows a standard search procedure when trying to find a file specified by the user:

1. Attempt to open the file in the current directory.
2. Attempt to open the file in the directory pointed to by the environment variable `XIAHOME`.
3. Attempt to open the file in the directory pointed to by the environment variable `DXPHOME`. (This is only for backwards compatibility with previous XIA libraries and should not be used.)

4. Interpret the filename as an environment variable that points to a different file.
5. Interpret the filename as an environment variable that points to a different file located in the directory pointed to by the environment variable `XIAHOME`.
6. Interpret the filename as an environment variable that points to a different file located in the directory pointed to by `DXPHOME`.

If all of the search steps fail then an error is returned.

Files

Handel is a framework of several libraries that interact to create the interface Handel provides.

The Handel layer (handel.dll) contains the xia* routines you will call. These routines are the focus of the API section.

Lower layers are for internal use. (Many of these were formerly distributed as separate DLLs but are now statically linked into Handel.dll.)

- Product Specific Layer (PSL) (saturn_psl, xmap_psl, etc.): product specific acquisition values, run types, run data types, and board operations.
- XerXes (xerxes): Formerly the public API for spectrometer control, now an internal dependency used by the PSL.
- Device driver layer (saturn, xmap, etc.): Device-level communication and utilities.
- Machine dependent (MD) (md, md_win32, md_linux, etc.): Operating system-level utilities. An implementation of the MD routines is needed for each platform Handel runs on. XIA provides full support for Windows (md_win32.c) and minimal support for Linux (md_linux.c). Patches are welcome to flesh out support for Linux or other operating systems.
- Protocol layer (xia_usb2.dll, xia_plx.dll): System device driver wrappers for the protocols supported by Handel.

INI File Format

Handel .ini files contain all the information needed to restore a system to an exact known configuration across program invocations and different host computers. A typical use case is to optimize a configuration for a particular detector type and spectrometer using XIA software such as ProSpect, and then save the system

for later use in loading into your own application via Handel. See System for the APIs used to load and save .ini files.

Handel uses the standard .ini file format of bracketed section headings ([Section]) followed by name-value pairs that define information for that section. Handel extends this format by allowing multiple aliases to be specifed under a single section heading. Each alias and its information is surrounded by the `START \#n` and `END \#n` keywords. A comment line is denoted by a "*" character at the start of a line.

The allowed section headings are "detector definitions", "firmware definitions" and "module definitions". Additionally, there is a section heading called "default definitions" that is generated by Handel. Users who are creating an .ini file from scratch should not include the "default definitions" section. Furthermore, the default_chan{n} value, in the "module definitions" section should also be left out since Handel will generate it automatically.

Detector

```
[detector definitions]
START #1
alias = detector1
number_of_channels = 1
type = reset
type_value = 10.0
channel0_gain = 6.6
channel0_polarity = +
END #1

START#2
alias = detector2
number_of_channels = 1
etc..
END #2

... START #3 etc...
```

Firmware

```
[firmware definitions]

* This firmware definition uses an FDD
START #1
alias = firmware1
filename = saturn_std.fdd
```

```
num_keywords = 0
END #1

* This firmware definition uses PTRRs
START #2
alias = firmware2
ptrr = 0
min_peaking_time = .25
max_peaking_time = 1.25
fippi = fxpd0g.fip
dsp = saturn.hex
num_filter = 2
filter_info0 = 2
filter_info1 = 2
ptrr = 1
min_peaking_time = 1.251
max_peaking_time = 5.0
fippi = fxpd2g.fip
dsp = saturn.hex
num_filter = 2
filter_info1 = 2
filter_info1 = 2
END #2

... START #3 etc...
```

Module

```
[module definitions]
START #1
alias = module1
module_type = saturn
number_of_channels = 1
interface = epp
epp_address = 0x378
channel0_alias = 0
channel0_detector = detector1:0
channel0_gain = 1.0
firmware_set_all = firmware1
END #1

... START #2 etc...
```

API

This section documents the public Handel API, divided into groups by functionality.

- Initializing Handel: initializing the library
- Detectors: querying detectors
- Firmware: querying firmware sets
- Modules: querying modules
- System: starting the system
- Run Params: setting and getting run parameters
- Run Control: starting and stopping runs and reading out data
- System Configuration: load and saving the system configuration
- Logging: configuring Handel's internal logging

Initializing Handel

Handel uses four global structures to manage a DAQ system: Detector, Firmware, Acquisition Values and Module information. The system is configured by calling xiaInit() and passing the name of a preconfigured .ini file. XIA software applications such as ProSpect save configurations for use in this manner. See INI Files for the file format specification.

- xiaInit()
- xiaInitHandel()
- xiaExit()

xiaInit()

```
int xiaInit(char *iniFile)
```

Initializes the Handel library and loads in an .ini file. The functionality of this routine can be emulated by calling xiaInitHandel() followed by xiaLoadSystem()("handel_ini", iniFile). Either this routine or `xiaInitHandel` must be called prior to using the other Handel routines.

Parameters:

iniFile Name of the configuration file to be loaded. This name may be an absolute or relative path and is located according to the conventions in Calling Conventions.

Return Codes:

Code	Description
XIA_XERXES	Called XerXes routine returned an error. Check error output.
XIA_NOMEM	Internal Handel error. Contact XIA.
XIA_OPEN_FILE	Unable to open specified .ini file.

Usage:

```
int status;
status = xiaInit("myFile.ini");
if (status != XIA_SUCCESS) {
    /* ERROR initializing library or loading .ini file */
}
```

xiaInitHandel()

```
int xiaInitHandel(void)
```

Initializes the library. Either this routine or xiaInit() must be called before any other Handel routines are used.

Return Codes:

Code	Description
XIA_XERXES	Called XerXes routine returned an error. Check error output.
XIA_NOMEM	Internal Handel error. Contact XIA.

Usage:

```
int status;
status = xiaInitHandel();
if (status != XIA_SUCCESS) {
    /* ERROR initializing Handel */
}
```

xiaExit()

```
int xiaExit(void)
```

Disconnects from the hardware and cleans up Handel's internal data structures.

Return Codes:

This routine has no specific error codes of its own. If anything besides XIA_SUCCESS is returned, check the log error output.

Usage:

```
int status;
status = xiaExit();
if (status != XIA_SUCCESS) {
    /* ERROR cleaning up Handel */
}
```

Detectors

Routines for querying information about physical detectors.

- xiaGetNumDetectors()
- xiaGetDetectors()
- xiaGetDetectors_VB()
- xiaGetDetectorItem()

xiaGetNumDetectors()

```
int xiaGetNumDetectors(unsigned int *numDet)
```

Gets the number of detectors currently defined in the system.

Parameters:

numDet User-allocated pointer to store the result.

Usage

```
int status;
unsigned int numDet = 0;

/* Assume that a system has already been
 * created or loaded and that it defines
 * two detectors.
 */
status = xiaGetNumDetectors(&numDet);

if (status != XIA_SUCCESS) {
    /* ERROR getting number of detectors */
}
```

```
printf("There are currently %u detector(s) defined.\n", numDet);
```

xiaGetDetectors()

```
int xiaGetDetectors(char *detectors[])
```

Gets a list of aliases of the detectors defined in the system.

> The caller must allocate for `detectors`. Typically this is done by calling xiaGetNumDetectors() and using the number of detectors to initialize a string array. Allocate a char * of length `MAXALIAS_LEN` for each detector.

Parameters:

detectors A string array of the proper length: number of detectors times MAXALIAS_LEN.numDet by `MAXALIAS_LEN` (defined in handel_generic.h).

Usage

```
int status;
unsigned int numDet = 0;
unsigned int i;
char **detectors = NULL;

/* Assume that a system has already been loaded. */
status = xiaGetNumDetectors(&numDet);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of detectors */
}

/* Allocate the memory we need for the string array */
detectors = (char **)malloc(numDet * sizeof(char *));
if (detectors == NULL) {
    /* ERROR allocating memory for detectors */
}

for (i = 0; i < numDet; i++) {
    detectors[i] = (char *)malloc(MAXALIAS_LEN * sizeof(char));
    if (detectors[i] == NULL) {
        /* ERROR allocating memory for detectors[i] */
    }
}

status = xiaGetDetectors(detectors);
```

```
if (status != XIA_SUCCESS) {
    /* ERROR getting detectors list */
}

for (i = 0; i < numDet; i++) {
    printf("detectors[%u] = %s\n", i, detectors[i]);
}

for (i = 0; i < numDet; i++) {
    free((void *)detectors[i]);
}

free((void *)detectors);
detectors = NULL;
```

xiaGetDetectors_VB()

```
int xiaGetDetectors_VB(unsigned int index, char *alias)
```

Gets a detector alias by detector index.

This routine serves as a replacement for xiaGetDetectors() for use with Visual Basic or other languages that will not allow an array of strings to be passed into the Handel DLL. This returns a single detector alias, where `index` ranges from 0 to numDetectors-1. The standard idiom is to get the number of detectors in the system with a call to xiaGetNumDetectors() and to then loop from 0 to numDetectors- 1 to get all of the detector aliases in the system.

Parameters:

index Position of detector alias in system, 0-based.

alias Returns the alias of the detector located at the specified index. The caller must allocate the proper amount of memory (char * of length `MAXALIAS_LEN`.

Return Codes:

Code	Description
XIA_BAD_INDEX	The specified detector index is out of range.

Usage

```
int status;
unsigned int numDetectors = 0;
```

```
unsigned int i;
char **aliases = NULL;

/* Assume that a valid system has been setup */
status = xiaGetNumDetectors(&numDetectors);
if (status != XIA_SUCCESS) {
    /* ERROR getting # of detectors in system */
}

/* Must allocate proper amount of memory */
aliases = (char **)malloc(numDetectors * sizeof(char *));
if (aliases == NULL) {
    /* ERROR allocating memory for aliases array */
}

for (i = 0; i < numDetectors; i++) {
    aliases[i] = (char *)malloc(MAXALIAS_LEN * sizeof(char));
    if (aliases[i] == NULL) {
        /* ERROR allocating memory for aliases[i] */
    }
}

for (i = 0; i < numDetectors; i++) {
    status = xiaGetDetectors(i, aliases[i]);
    if (status != XIA_SUCCESS) {
        /* ERROR getting detector alias at index i */
    }
}

for (i = 0; i < numDetectors; i++) {
    printf("Detector alias at index = %u: %s", i, aliases[i]);
}

for (i = 0; i < numDetectors; i++) {
    free((void *)aliases[i]);
}

free(aliases);
```

xiaGetDetectorItem()

```
int xiaGetDetectorItem(char *alias, char *name, void *value)
```

Retrieves information from a detector's configuration. All of the names listed in below may be retrieved using this routine.

Detector Items

name	value Type	Description
number_of_channels	unsigned int	The number of detector elements. **Must be added first**.
channel{n}_gain	double	The preamplifier gain in mV/keV for channel n (0-based).
channel{n}_polarity	char *	Polarity of channel n. "+" and "pos" indicate positive polarity. "-" an
type	char *	The type of preamplifier this detector has. "reset" or "rc_feedback".
type_value	double	The value associated with the preamplifier type. For reset detectors pa

Parameters:

alias A valid detector alias
name Name from the detector items table to get.
value Void pointer to the variable in which the returned data will be stored. See the detector items list above for the correct data type to allocate for each `name`.

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist
XIA_BAD_VALUE	`value` could not be converted to the required type.
XIA_BAD_NAME	`name` is not a valid detector item.

Usage

```
int status;
double gain;

/* Create a detector w/ alias detector1 here and then add all of the
 * the necessary information to it. We will only retrieve the gain
 * here, but the others follow the same pattern.
 */

status = xiaGetDetectorItem("detector1", "channel0_gain", (void *)&gain);
if (status != XIA_SUCCESS) {
    /* ERROR getting channel 0 gain */
}

printf("Gain (channel 0) = %lf\n", gain);
```

Firmware

Routines for querying information about firmware sets, firmware files, and mappings between peaking time ranges and firmware.

- xiaGetNumFirmwareSets()
- xiaGetFirmwareSets()
- xiaGetFirmwareSets_VB()
- xiaGetNumPTRRs()
- xiaGetFirmwareItem()

xiaGetNumFirmwareSets()

```
int xiaGetNumFirmwareSets(unsigned int *numFirmware)
```

Returns the number of firmware sets defined in the system.

Parameters:

numFirmware Pointer to a variable to store the returned number of firmware sets in

Usage

```
int status;
unsigned int numFirmware = 0;

/* Assume that a system has already been
 * created or loaded and that it defines
 * two firmware sets.
 */
status = xiaGetNumFirmwareSets(&numFirmware);

if (status != XIA_SUCCESS) {
    /* ERROR getting number of firmware sets */
}

printf("There are currently %u firmware set(s) defined.\n", numFirmware);
```

xiaGetFirmwareSets()

```
int xiaGetFirmwareSets(char *firmware[])
```

Returns a list of the aliases of the firmware sets defined in the system.

The caller must allocate memory for `firmware`.

Parameters:

firmware A string array of the proper length: numFirmware by `MAXALIAS\_LEN` (defined in handel_generic.h) Typically this is done by calling xiaGetNumFirmwareSets() and using the number of firmware sets to initialize a string array. Allocate a `char \*` of length `MAXALIAS_LEN` for each firmware set.

Usage

```
int status;
unsigned int numFirmware = 0;
unsigned int i;
char **firmware = NULL;

/* Assume that a system has already been loaded. */
status = xiaGetNumFirmwareSets(&numFirmware);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of firmware sets */
}

/* Allocate the memory we need for the string array */
firmware = (char **)malloc(numFirmware * sizeof(char *));
if (firmware == NULL) {
    /* ERROR allocating memory for firmware sets */
}

for (i = 0; i < numFirmware; i++) {
    firmware[i] = (char *)malloc(MAXALIAS_LEN * sizeof(char));
    if (firmware[i] == NULL) {
        /* ERROR allocating memory for firmware[i] */
    }
}

status = xiaGetFirmwareSets(firmware);
if (status != XIA_SUCCESS) {
    /* ERROR getting firmware set list */
}

for (i = 0; i < numFirmware; i++) {
    printf("firmware[%u] = %sn", i, firmware[i]);
}

for (i = 0; i < numFirmware; i++) {
    free((void *)firmware[i]);
}
```

```
free((void *)firmware);
firmware = NULL;
```

xiaGetFirmwareSets_VB()

```
int xiaGetFirmwareSets_VB(unsigned int index, char *alias)
```

Gets a firmware set alias by index. This routine serves as a replacement of the routine xiaGetFirmwareSets() for use with Visual Basic or other languages that will not allow an array of strings to be passed into the Handel DLL. The difference between this routine and `xiaGetFirmwareSets` is that `xiaGetFirmwareSets` returns a list of all of the firmware aliases that are currently defined in the system. `xiaGetFirmwareSets_VB` returns a single firmware alias, where index ranges from 0 to numFirmware – 1. The standard idiom is to get the number of firmware sets in the system with a call to xiaGetNumFirmwareSets() and to then loop from 0 to numFirmware –1 in order to get all of the firmware aliases in the system. See the Usage section for an example of how this is done.

The caller must allocate memory for `alias`.

Parameters:

index Position of firmware alias in system where index ranges from 0 to numFirmware – 1. For instance, if you have a system where 3 firmware sets are defined, the valid values for index are 0, 1 and 2.

alias Alias of the firmware located at the specified index. The caller must allocate memory a `char \*` of length `MAXALIAS_LEN` (defined handel_generic.h).

Return Codes:

Code	Description
XIA_BAD_INDEX	The specified index is out-of-range

Usage

```
int status;
unsigned int numFirmware = 0;
unsigned int i;
char **aliases = NULL;

/* Assume that a valid system has been setup
 * here.
 */
status = xiaGetNumFirmwareSets(&numFirmware);
```

```
if (status != XIA_SUCCESS) {
    /* ERROR getting # of firmware sets in system */
}

/* Must allocate proper amount of memory */
aliases = (char **)malloc(numFirmware * sizeof(char *));
if (aliases == NULL) {
    /* ERROR allocating memory for aliases array */
}

for (i = 0; i < numFirmware; i++) {
    aliases[i] = (char *)malloc(MAXALIAS_LEN * sizeof(char));
    if (aliases[i] == NULL) {
        /* ERROR allocating memory for aliases[i] */
    }
}

for (i = 0; i < numFirmware; i++) {
    status = xiaGetFirmwareSets(i, aliases[i]);
    if (status != XIA_SUCCESS) {
        /* ERROR getting firmware alias at index i */
    }
}

for (i = 0; i < numFirmware; i++) {
    printf("Firmware alias at index = %u: %s", i, aliases[i]);
}

for (i = 0; i < numFirmware; i++) {
    free((void *)aliases[i]);
}

free(aliases);
```

xiaGetNumPTRRs()

```
int xiaGetNumPTRRs(char *alias, unsigned int numPTRR)
```

Returns the number of PTRRs that are defined for the firmware set with the specified alias. If the firmware has an FDD defined instead of PTRRs an error will be returned.

Parameters:

alias A valid firmware alias

numPTRR A pointer to a variable to store the number of PTRRs in.

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist
XIA_LOOKING_PTRR	The specified firmware has an FDD defined

Usage

```
int status;
unsigned int numPTRR = 0;

/* Assume firmware with alias "firmware1" already
 * exists in the system.
 */
status = xiaGetNumPTRRs("firmware1", &numPTRR);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of PTRRs */
}

printf("firmware1 has %u PTRR(s).\n", numPTRR);
```

xiaGetFirmwareItem()

```
int xiaGetFirmwareItem(char *alias, unsigned short ptrr, char *name, void *value)
```

Retrieves information from a firmware set's configuration. All of the names that are listed below may be retrieved. If you get the "filename" item for a firmware alias that uses PTRRs, a blank string will be returned.

Firmware Items

FDD

name	type Value	Description
filename	null-terminated string	Name of FDD file to be used. This file will be searched for using the stand
keyword	null-terminated string	Each time keyword is used as a name, another keyword is appended to the

No FDD

name	type Value	Description
mmu	null-terminated string	The filename of the memory management unit, if present. (Op

ptrr	unsigned short	A unique identifier for a Peaking Time Range Reference. Each
min_peaking_time	double	The minimum peaking time value for the current PTRR in Âµ
max_peaking_time	double	The maximum peaking time value for the current PTRR in Âµ
fippi	null-terminated string	The filename of the FiPPI program to be downloaded for this
dsp	null-terminated string	The filename of the DSP program to be downloaded for this P
user_fippi	null-terminated string	The filename of the user-defined FiPPI program to be downloa
filter_info	unsigned short	Add another filter information value to the existing informatio

Additional Items

num_filter `(unsigned short)` The number of elements in the filter_info array for the specified PTRR. The typical use of this value is for allocating enough memory to retrieve the entire filter_info array using the **filter_info** parameter.

filter_info `(unsigned short)`

Parameters:

alias A valid firmware alias

ptrr The PTRR that corresponds to the information to be retrieved. Not all names to be retrieved require a PTRR, in which case it may be set to NULL.

name Name of value to retrieve

value Void pointer to variable in which the returned data will be stored. It is very important that the type of this variable is appropriate for the data to be retrieved. See the tables above for more information. See the Usage section for more information on how to use void pointers in this context.

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist
XIA_BAD_VALUE	No PTRRs defined for this alias
XIA_BAD_PTRR	Specified PTRR does not exist
XIA_BAD_NAME	Specified name is invalid

Usage

```
int status;
double min_ptime;

/* Create a firmware w/ alias "firmware1" using PTRRs and add all of
 * the necessary information to it. We will only retrieve the minimum
```

```
 * peaking time for PTRR 0 here. Retrieving other values should follow
 * similar patterns.
 */

status = xiaGetFirmwareItem("firmware1", 0, "min_peaking_time", (void *)&min_ptime);

if (status != XIA_SUCCESS) {
    /* ERROR Getting PTRR 0 minimum peaking time */
}

printf("Minimum peaking time (PTRR 0) = %lf\n", min_ptime);
```

Modules

Routines for querying information about modules. A module is an XIA card consisting of one or more channels.

- xiaGetNumModules()
- xiaGetModules()
- xiaGetModules_VB()
- xiaGetModuleItem()
- Enumerating Modules and Channels

xiaGetNumModules()

```
int xiaGetNumModules(unsigned int *numModules)
```

Returns the number of modules currently defined in the system.

Parameters:

numModules Pointer to a variable to store the returned number of detectors in

Usage

```
int status;
unsigned int numModules = 0;

/* Assume that a system has already been
 * created or loaded.
 */
status = xiaGetNumModules(&numModules);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of modules */
```

```
}

printf("There are currently %u modules defined.\n", numModules);
```

xiaGetModules()

```
int xiaGetModules(char *modules[])
```

Returns a list of the aliases of the modules defined in the system.

The caller must allocate memory for `modules`. Typically this is done by calling xiaGetNumModules() and using the number to initialize a string array. Allocate a `char \*` of length `MAXALIAS_LEN` for each module.

Parameters:

modules A string array of the proper length: numModules by MAXALIAS_LEN (defined in handel_generic.h)

Usage

```
int status;
unsigned int numModules = 0;
unsigned int i;
char **modules = NULL;

/* Assume that a system has already been loaded. */
status = xiaGetNumModules(&numModules);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of modules */
}

/* Allocate the memory we need for the string array */
modules = (char **)malloc(numModules * sizeof(char *));
if (modules == NULL) {
    /* ERROR allocating memory for modules */
}

for (i = 0; i < numModules; i++) {
    modules[i] = (char *)malloc(MAXALIAS_LEN * sizeof(char));
    if (modules[i] == NULL) {
        /* ERROR allocating memory for modules[i] */
    }
}

status = xiaGetModules(modules);
```

```
if (status != XIA_SUCCESS) {
    /* ERROR getting module list */
}

for (i = 0; i < numModules; i++) {
    printf("modules[%u] = %sn", i, modules[i]);
}

for (i = 0; i < numModules; i++) {
    free((void *)modules[i]);
}

free((void *)modules);
modules = NULL;
```

xiaGetModules_VB()

```
int xiaGetModules_VB(unsigned int index, char *alias)
```

Gets a module alias by index. This routine serves as a replacement of the routine xiaGetModules() for use with Visual Basic or other languages that will not allow an array of strings to be passed into the Handel DLL. This returns a single module alias, where `index` ranges from 0 to numModules – 1. The standard idiom is to get the number of modules in the system with a call to xiaGetNumModules() and then loop from 0 to numModule –1 in order to get all of the module aliases in the system.

The caller must allocate memory for `alias`.

Parameters:

index Position of module alias in system where index ranges from 0 to numModule – 1. For instance, if you have a system where 3 modules are defined, the valid values for index are 0, 1 and 2.

alias Alias of the module located at the specified index. The caller must allocate a string of length `MAXALIAS\_LEN` (defined in handel_generic.h).

Return Codes:

Code	Description
XIA_BAD_INDEX	The specified index is out-of-range

Usage

```
int status;
unsigned int numModules = 0;
unsigned int i;
char **aliases = NULL;

/* Assume that a valid system has been setup */
status = xiaGetNumModules(&numModules);
if (status != XIA_SUCCESS) {
    /* ERROR getting # of modules in system */
}

/* Must allocate proper amount of memory */
aliases = (char **)malloc(numModules * sizeof(char *));
if (aliases == NULL) {
    /* ERROR allocating memory for aliases array */
}

for (i = 0; i < numModules; i++) {
    aliases[i] = (char *)malloc(MAXALIAS_LEN * sizeof(char));
    if (aliases[i] == NULL) {
        /* ERROR allocating memory for aliases[i] */
    }
}

for (i = 0; i < numModules; i++) {
    status = xiaGetModules_VB(i, aliases[i]);
    if (status != XIA_SUCCESS) {
        /* ERROR getting module alias at index i */
    }
}

for (i = 0; i < numModules; i++) {
    printf("Module alias at index = %u: %s", i, aliases[i]);
}

for (i = 0; i < numModules; i++) {
    free((void *)aliases[i]);
}

free(aliases);
```

xiaGetModuleItem()

```
int xiaGetModuleItem(char *alias, char *name, void *value)
```

Retrieve information from a module's configuration. All names listed below may be retrieved using this routine except for *firmware_set_all*. Consult the list for the data types to use for `value`.

Module Items

Items in the following list are supported for all module types.

module_type ***null-terminated string*** The name corresponding to the type of module. Supported types are saturn (and dxpx10p and x10p for backward compatibility), udxp, udxps, xmap, mercury, and stj. Note that a build of Handel may support only one product line; the other products are compiled out of the code. **This must be specified first.**

interface ***null-terminated string*** The physical interface type for this module. Supported interfaces are "genericEPP", "epp", "usb", "usb2", "serial", "pxi". It is not strictly necessary to specify the interface, as Handel can recognize the interface based on the first interface item that is added to the system.

number_of_channels ***unsigned int*** Number of channels associated with a module of this type. **Not the number of channels in use.** That information will be specified elsewhere. If you have a 4-channel XMAP module and only plan to use 2 of the channels, this value must still be set to 4. This value must be added before any of values listed below are added.

channel{n}_alias ***int*** The "detChan" value for channel n. Each physical channel in the system has a unique "detChan" value associated with it. This value is important because it is how an individual channel is operated on by other Handel routines. To disable a channel, set its detChan value to -1.

channel{n}_detector ***null-terminated string*** The alias of the detector that channel n is attached to. The format of this string is "detector_alias:m" where is m is the channel number of the actual **detector channel** (preamplifier) that this module channel is attached to. The alias must be a valid detector alias defined in the detector section of the .ini file.

channel{n}_gain ***double*** Gain, in this context, describes any modifications made to the input gain stage of a channel. Most users should set this to 1.00. This value is not the same as the preamplifier gain, as described in the Detectors section, nor is it the same as the gain set by the DSP parameter `GAINDAC`.

firmware_set_all ***null-terminated string*** Same as firmware_set_chan{n} except that the same alias will be used for all channels in the module.

The following lists illustrate the names that apply to specific interfaces.

genericEPP and epp:

epp_address ***unsigned int*** The address of the EPP port on the host com-

puter. Typically the address is 0x378 or, ocassionally, 0x278.

daisy_chain_id ***unsigned int*** The daisy chain ID for this module. Should only be specified if the module specifically implements the daisy chain protocol. (Optional)

usb and usb2:

device_number ***unsigned int*** The USB device number associated with this module. Typically this is 0 for a single module system, and increments by one for each additional module.

serial:

com_port ***unsigned int*** The one-based serial port number to which the module is attached. Port number N corresponds to COMx where x = N-1.

device_file ***char **_*** On Linux, a device file to open for terminal IO. This replaces *com_port* on that platform.

baud_rate ***unsigned int*** The baud rate of the serial port. Typically 115200 but may be firmware-dependent.

pxi:

pci_slot ***byte_t***

pci_bus ***byte_t***

Parameters:

alias A valid module alias

name Name of value to retrieve

value Void pointer to variable in which the returned data will be stored. It is very important that the type of this variable is appropriate for the data to be retrieved. See the lists above for more information. Consult the Usage section below for more information on how to use void pointers in this context.

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist
XIA_BAD_NAME	Specified name is invalid
XIA_WRONG_INTERFACE	Specified name does not apply to the current interface
XIA_BAD_CHANNEL	Internal Handel error. Contact XIA.

Usage

```
int status;
int detChan;

/* Create a module with alias module1 here and then add all of the
 * information associated with a DXP-X10P to it. We will only
 * retrieve the detChan value of channel 0 here but the other
 * names follow the same pattern.
 */

status = xiaGetModuleItem("module1", "channel0_alias", (void *)&detChan);
if (status != XIA_SUCCESS) {
    /* ERROR Getting channel0_alias */
}

printf("Channel 0 detChan = %d\n", detChan);
```

Enumerating Modules and Channels

While XIA systems are defined hierarchically as a collection of modules each containing a number of channels, acquisition APIs require a unique channel alias, also known as a detChan, as the primary identifying argument. Thus some means of querying the system is needed to obtain the detChans to use in calling acquisition APIs.

While detChans by definition are only constrained to be unique for each channel, XIA tools do generate .ini files with zero-based whole number channel aliases. Therefore if you know the number of channels in the system, you can treat the detChan as a zero-based index. This may be easy in a single module system, but for robustness we recommend looping through the modules to get the aliases.

The following example shows the easiest way to loop through all modules and channels and perform an API call on each detChan. The same loop structure could be used at the beginning of your application to cache the detChans in a module structure for later use.

> The examples in this section use the routine xiaGetModules_VB() for convenience to avoid allocating memory for all the module aliases. You may also get all the aliases in one call using the techniques demonstrated in xiaGetModules().

```
int status;
unsigned int numModules = 0;
unsigned int mod;

/* Assume that a valid system has been setup */
status = xiaGetNumModules(&numModules);
if (status != XIA_SUCCESS) {
```

```
    /* ERROR getting # of modules in system */
}

/* For each module in the system */
for (mod = 0; mod < numModules; mod++) {
    char module[MAXALIAS_LEN];
    int numChannels;
    int chan;

    status = xiaGetModules_VB(mod, module);
    if (status != XIA_SUCCESS) {
        /* ERROR getting module alias at index mod */
    }

    status = xiaGetModuleItem(module, "number_of_channels", &numChannels);
    if (status != XIA_SUCCESS) {
        /* ERROR getting number of channels for module */
    }

    /* For each channel in the module */
    for (chan = 0; chan < numChannels; chan++) {
        char item[20];
        int detChan;

        sprintf(item, "channel%d_alias", chan);
        status = xiaGetModuleItem(module, item, &detChan);
        if (status != XIA_SUCCESS) {
            /* ERROR getting channel alias */
        }

        /* Performing an acquisition task with detChan here will do it
         * for each channel in the system.
         */
    }
}
```

To perform a task on the first channel in each module, simply operate on "channel0_alias" instead of all channels, as shown in the following example. The same loop structure could be used with module item "number_of_channels" to count the total number of channels in the system.

```
int status;
unsigned int numModules = 0;
unsigned int mod;

/* Assume that a valid system has been setup */
status = xiaGetNumModules(&numModules);
```

```
if (status != XIA_SUCCESS) {
    /* ERROR getting # of modules in system */
}

/* For each module in the system */
for (mod = 0; mod < numModules; mod++) {
    char module[MAXALIAS_LEN];
    int numChannels;
    int chan;

    /* Get the module alias */
    status = xiaGetModules_VB(mod, module);
    if (status != XIA_SUCCESS) {
        /* ERROR getting module alias at index mod */
    }

    /* Get the detChan of the first channel in the module */
    status = xiaGetModuleItem(module, "channel0_alias", &detChan);
    if (status != XIA_SUCCESS) {
        /* ERROR getting channel alias */
    }

    /* Performing an acquisition task with detChan here will do it
     * for the first channel in each module, e.g. get
     * run data module_statistics_2.
     */
}
```

System

Routines for starting the system and loading firmware.

- xiaStartSystem()
- xiaDownloadFirmware()
- xiaBoardOperation()

xiaStartSystem()

```
int xiaStartSystem(void)
```

Starts the system previously defined via an .ini file. Connects to the hardware and downloads the firmware and acquisition values to all active channels in order to set the system up for data acquisition. This routine must be called after configuring the system with a configuration file (see Initializing Handel).

This routine also performs several validation steps to insure that all of the configuration information required to run the system is present. Specifically, the firmware and detector information is validated by Handel while the module is verified by the Product Specific Layer. If an inconsistency is found, it will be reported back as an error and should be fixed before attempting to call xiaStartSystem() again.

This routine may block for up to several seconds, depending on the size of the system and timing of firmware downloading.

Return Codes:

Code	Description
XIA_FIRM_BOTH	Both a FDD file and PTRR information have been specified in one of the fi
XIA_PTR_OVERLAP	A PTRR has a peaking time range that overlaps with another PTRR.
XIA_MISSING_FIRM	The DSP and/or FiPPI information is missing for a PTRR.
XIA_MISSING_POL	The polarity isn't defined for a detector.
XIA_MISSING_GAIN	The preamplifier gain isn't defined for at least one detector channel.
XIA_MISSING_TYPE	The detector type isn't defined for a detector.
XIA_NO_DETCHANS	No detChans are defined in the system.
XIA_INFINITE_LOOP	Problem with detChan and detChan Set definitions such that an infinite loo
XIA_UNKNOWN	Internal error. Contact XIA.
XIA_INVALID_DETCHAN	A detChan in the system does not refer to an existing module.
XIA_NO_ALIAS	Internal Handel error. Contact XIA.
XIA_BAD_NAME	Internal Handel error. Contact XIA.
XIA_WRONG_INTERFACE	Internal Handel error. Contact XIA.
XIA_BAD_CHANNEL	Internal Handel error. Contact XIA.
XIA_UNKNOWN_BOARD	Board type in system does not exist in Handel.
XIA_MISSING_INTERFACE	A module in the system is missing interface information.
XIA_MISSING_ADDRESS	**(For Saturn/Mercury)** EPP address missing from interface information.
XIA_INVALID_NUMCHANS	The number of channels set for a board type is incorrect.
XIA_BINS_OOR	The bin range is out-of-range for the board type.
XIA_XERXES	Error reported by Xerxes routine called by Handel.
XIA_BAD_VALUE	Internal Handel error. Contact XIA.
XIA_FILEERR	Error getting firmware from FDD file.

Usage:

```
/* Assume a system has been created dynamically or loaded from
 * an .ini file.
 */

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

```
if (status != XIA_SUCCESS) {
    /* ERROR Starting system */
}
```

xiaDownloadFirmware()

```
int xiaDownloadFirmware(int detChan, char *type)
```

Downloads the specified firmware type to a detChan. The following firmware types are recognized: “dsp”, “fippi”, “user_dsp”, “user_fippi” and “system_fpga”.

> The task of downloading firmware to the system is typically handled by xiaStartSystem(), so this routine should only be used for situations where special firmware is required.

Parameters:

detChan detChan to download firmware to. May be either a single detChan or detChan set. -1 is not allowed.

type The type of firmware to be downloaded. Must be one of “dsp”, “fippi”, “user_dsp”, “user_fippi” and “system_fpga”.

Return Codes:

Code	Description
XIA_NO_ALIAS	Internal Handel error. Contact XIA.
XIA_BAD_VALUE	Internal Handel error. Contact XIA.
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_NO_ALIAS	Internal Handel error. Contact XIA.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel.
XIA_NOSUPPORT_FIRM	The specified type of firmware to download is not supported for this board ty
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_UNKNOWN_FIRM	The specified type of firmware to download is unknown.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;

/* Set up a valid system here */
status = xiaStartSystem();
if (status != XIA_SUCCESS) {
    /* ERROR starting system */
```

```
}

/* Want to start DSP code again */
status = xiaDownloadFirmware(0, "dsp");
if (status != XIA_SUCCESS) {
    /* ERROR downloading DSP to detChan o */
}
```

xiaBoardOperation()

```
int HANDEL_API xiaBoardOperation(int detChan, char *name, void *value)
```

Performs product-specific queries and operations.

Parameters:

detChan detChan to apply the acquisition value to. May only be a single detChan; sets are not allowed.

name Type of data to pass or return. Reference the product-specific Handel manuals for complete lists by product.

value Variable to return the data in, cast into a `void *`. See the product-specific Handel manuals for the required data type for each name.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_XERXES	Error reported by Xerxes routine called by Handel.
XIA_BAD_TYPE	`detChan` refers to a detChan set, which is not allowed in this routine.
XIA_UNKNOWN	Internal Handel error. Contact XIA.
XIA_BAD_CHANNEL	Internal Handel error. Contact XIA.

Usage

```
int status;
double new_threshold = 1000.0;

/* Set up a valid system here */
status = xiaStartSystem();
if (status != XIA_SUCCESS) {
    /* ERROR starting system */
}
```

```
/* Change trigger threshold to 1000 eV */
status = xiaSetAcquisitionValue(0, "trigger_threshold", (void *)&new_threshold);
if (status != XIA_SUCCESS) {
    /* ERROR setting trigger threshold */
}

printf("Trigger threshold now set to: %lf\n", new_threshold);
```

Run Params

Routines for setting and getting acquisition values, or run parameters.

- xiaSetAcquisitionValues()
- xiaGetAcquisitionValues()
- xiaRemoveAcquisitionValues()
- xiaUpdateUserParams()
- xiaGainOperation()
- xiaGainChange()
- xiaGainCalibrate()
- xiaGetParameter()
- xiaSetParameter()
- xiaGetNumParams()
- xiaGetParamData()
- xiaGetParamName()

xiaSetAcquisitionValues()

```
int xiaSetAcquisitionValues(int detChan, char *name, void *value)
```

Translates a high-level acquisition value into the appropriate DSP parameter(s) in the hardware. Product-specific Handel manuals list the acquisition values for each product.

This is the preferred method for modifying the DSP settings of a module since Handel and the PSL are responsible for making all of the necessary calculations and setting all of the necessary parameters.

In some cases, the actual acquisition value will be slightly different then the value passed in. This routine returns the actual value in the value parameter so that the host software may keep its data synchronized with the data in Handel.

This routine will also accept names that are in all capital letters and interpret them as DSP parameters. Calling this routine with a DSP parameter as the name will cause the parameter to be written to the channel specified by detChan and will also add it to the list of acquisition values to be saved.

Setting acquisition values is the mechanism that Handel provides for persistence of DSP and acquisition value settings. Handel will save this information in the .ini file generated by a call to xiaSaveSystem(). Calling xiaLoadSystem() with the generated .ini file will cause the saved parameter and acquisition values to be loaded into the DSP. This allows for a system to be started up in a state very close to the one it was saved in. See also: Initializing Handel.

Parameters:

detChan detChan to apply the acquisition value to. May be a single detChan or a detChan set.

name The name of the acquisition value supported by the product or a DSP parameter.

value Value to set the corresponding acquisition value to. `double *` cast to `void *`.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel.
XIA_DET_UNKNOWN	Internal Handel error. Contact XIA.
XIA_XERXES	Error reported by Xerxes routine called by Handel.
XIA_PEAKINGTIME_OOR	**(For Saturn/Mercury)** New peaking time is out of range for specified proc
XIA_FILEERR	**(For Saturn/Mercury)** Error getting firmware from FDD file.
XIA_OPEN_FILE	**(For Saturn/Mercury)** Error opening temporary file.
XIA_NOSUPPORT_FIRM	**(For Saturn/Mercury)** The specified type of firmware to download is not
XIA_UNKNOWN_FIRM	**(For Saturn/Mercury)** The specified type of firmware to download is unkı
XIA_BINS_OOR	**(For Saturn/Mercury)** The specified number of bins is out of range for th
XIA_GAIN_OOR	**(For Saturn/Mercury)** The computed gain value is out of range for this b
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
double new_threshold = 1000.0;

/* Set up a valid system here */
status = xiaStartSystem();
if (status != XIA_SUCCESS) {
    /* ERROR starting system */
}

/* Change trigger threshold to 1000 eV */
```

```
status = xiaSetAcquisitionValue(0, "trigger_threshold", (void *)&new_threshold);
if (status != XIA_SUCCESS) {
    /* ERROR setting trigger threshold */
}

printf("Trigger threshold now set to: %lf\n", new_threshold);
```

xiaGetAcquisitionValues()

```
int xiaGetAcquisitionValues(int detChan, char *name, void *value)
```

Retrieves the current setting of an acquisition value. This routine returns the same value as xiaSetAcquisitionValues() in the value parameters.

Parameters:

detChan detChan to retrieve the acquisition value from. Must be a single detChan.

name The name of the acquisition value to retrieve. Reference the product-specific Handel manuals for the complete list by product.

value Variable to return the value. `double *` cast in `void *`.

Return Codes:

Code	Description
XIA_BAD_TYPE	Specified detChan must be a single detChan and not a detChan set.
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel.
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_UNKNOWN_VALUE	Specified name isn't supported by this board type.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
double peaking_time;

/* Setup valid system here */
status = xiaGetAcquisitionValues(0, "peaking_time", (void *)&peaking_time);
if (status != XIA_SUCCESS) {
    /* ERROR getting peaking time */
}
```

```
printf("Peaking time = %lf\n", peaking_time);
```

xiaRemoveAcquisitionValues()

```
int xiaRemoveAcquisitionValues(int detChan, char *name)
```

Removes an acquisition value for the specified channel.

This routine is not generally needed in user programs, which work with built in acquisition values and do not add and remove their own. Handel protects against the removal of any acquisition values that are required for a specific board type.

> The implementation of this routine reapplies the values of other acquisition values to the device. Therefore, as is documented practice with other acquisition values routines, it must only be called during the setup phase of the program, i.e. before the first run or in between runs.

Parameters:

detChan detChan or detChan set to remove the acquisition value from.
name The name of the acquisition value to remove

Return Codes:

Code	Description
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
unsigned short slowgap = 3;

/* Assume valid system already setup */
/* Add optional "slowgap" acquisition value */
status = xiaSetAcquisitionValues(0, "slowgap", (void *)&slowgap);
if (status != XIA_SUCCESS) {
    /* ERROR adding slowgap acquisition value */
}

/* Now, remove it from the acquisition list */
status = xiaRemoveAcquisitionValues(0, "slowgap");
if (status != XIA_SUCCESS) {
    /* ERROR removing slowgap acquisition value */
```

```
}
```

xiaUpdateUserParams()

```
int xiaUpdateUserParam(int detChan)
```

Downloads the user parameters from the list of current acquisition values for the specified channel. In this context, a user parameter is a DSP parameter that has been added to the acquisition values list by a call to xiaSetAcquisitionValues(). This routine checks the acquisition values list for all DSP parameters and then downloads them to the board.

Parameters:

detChan detChan or detChan set to download parameters to.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel.
XIA_XERXES	Error reported by Xerxes routine called by Handel.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;

/* Assume valid system already setup */

/* Set DSP parameters via xiaSetAcquisitionValues uppercase name
 * notation. */

status = xiaUpdateUserParams(0);
if (status != XIA_SUCCESS) {
    /* ERROR updating user parameters */
}
```

xiaGainOperation()

```
int xiaGainOperation(int detChan, char *name, void *value)
```

Performs product-specific special gain operations. This routine supersedes the deprecated xiaGainCalibrate() and `xiaGainChange`.

Gain Operations

All products

calibrate Calibrates the gain using the specified delta. This gain operation supersedes the deprecated routine xiaGainCalibrate().

Adjusts the specified channel's settings in order to scale the energy value by the specified amount. Typically the preamplifier gain is adjusted by the inverse of the delta.

It may take several iterations of measuring and shifting the energy value in order to achieve the correct energy value due to small variations in gain control sensitivity.

> For Saturn, unlike the gain operation `adjust\_percent\_rule` (formerly the routine `xiaGainChange`), the result of this routine is that the energy value is shifted by deltaGain. `adjust\_percent\_rule` modifies the absolute step size at the ADC but does not change the energy value.

Saturn

adjust_percent_rule Adjusts the acquisition value `adc_percent_rule` by the delta. This gain operation replaces the removed `xiaGainChange` routine.

microDXP

calibrate_gain_trim Applies the delta to the acquisition value `gain_trim`.

STJ

scale_digital_gain Calibrates the digital gain using the specified delta. Adjusts the digital gain by the inverse of the specified delta but doesn't change the associated analog gain. Calling this gain operation is equivalent to calling the special run `scale_digital_gain`.

Parameters:

detChan detChan or detChan set to perform the gain operation on
name Name of the gain operation to perform.

value Value required for the gain operation cast into a void pointer. The underlying value is a double for all gain operations defined to date.

Return Codes:

Code	Description
XIA_MISSING_TYPE	Internal Handel error. Contact XIA.
XIA_INVALID_DETCHAN	Specifed detChan does not exist in Handel.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel.
XIA_MISSING_TYPE	Internal Handel error. Contact XIA.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage:

```
int status;

/* Set up a valid system here */

/* Scale gain by a factor of 2 */
double gainDelta = 2.0;
status = xiaGainOperation(0, "calibrate_gain_trim",  &gainDelta);
if (status != XIA_SUCCESS) {
    /* ERROR Changing gain */
}
```

xiaGainCalibrate()

```
int xiaGainCalibrate(int detChan, double deltaGain)
```

Deprecated. See xiaGainOperation().

Adjusts the specified channel's settings in order to scale the energy value by the specified amount. Typically the preamplifier gain is adjusted by the inverse of the delta.

It may take several iterations of measuring and shifting the energy value in order to achieve the correct energy value due to small variations in gain control sensitivity.

> For Saturn, unlike the gain operation `adjust\_percent\_rule` (formerly the routine `xiaGainChange`), the result of this routine is that the energy value is shifted by deltaGain. `adjust\_percent\_rule` modifies the absolute step size at the ADC but does not change the energy value.

Parameters:

detChan detChan or detChan set to apply the calibration to.
deltaGain Factor by which to scale the gain.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_MISSING_TYPE	Internal Handel error. Contact XIA.
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_GAIN_OOR	**(For Saturn/Mercury)** The calculated gain value (in dB) is out of range.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
double calibEV = 5900.0;
double peakEV = 0.0;
double scaleFactor = 0.0;

/* Set up a valid system here */

/* To calibrate a spectrum peak, get the current
 * peak position and divide the calibration energy
 * by it to get the scale factor. Adjust the gain by
 * the scale factor.
 */

/* Get actual peak position here using whatever method
 * is appropriate. We will assume that peakEV is set
 * somehow.
 */

scaleFactor = calibEV / peakEV;
status = xiaGainCalibrate(0, scaleFactor);
if (status != XIA_SUCCESS) {
    /* ERROR calibrating gain */
}
```

xiaGetParameter()

```
int xiaGetParameter(int detChan, const char *name, unsigned short *value)
```

Gets the current value of a DSP parameter for the specified channel.

CAUTION: Both this routine and xiaSetParameter() work directly with the parameters in the DSP. User programs should generally use xiaGetAcquisitionValues() instead.

Parameters:

detChan detChan to get the value from. Must be a single detChan.
name Name of DSP parameter to retrieve
value Pointer to `unsigned short` variable in which to return the value of the DSP parameter.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_BAD_TYPE	detChan is a set, not a single detChan.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;

unsigned short value;

/* Set up a valid system here */

status = xiaGetParameter(0, "DECIMATION", &value);
if (status != XIA_SUCCESS) {
    /* ERROR getting DECIMATION */
}

printf("Decimation = %u\n", value);
```

xiaSetParameter()

```
int xiaSetParameter(int detChan, const char *name, unsigned short value)
```

Sets a DSP paramter for the specified channel. If the parameter is marked as read-only by the DSP it will not be modified.

CAUTION: Both this routine and xiaGetParameter() work directly with the parameters in the DSP. User programs should generally use xiaSetAcquisitionValues() instead.

Parameters:

detChan detChan or detChan set to set the DSP parameter on.
name Name of DSP parameter to set.
value Pointer in which to return the DSP parameter.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
unsigned short newThresh = 0x1000;

/* Set up a valid system */

status = xiaSetParameter(0, "THRESHOLD", newThresh);
if (status != XIA_SUCCESS) {
    /* ERROR setting THRESHOLD */
}
```

xiaGetNumParams()

```
int xiaGetNumParams(int detChan, unsigned short *numParams)
```

Returns the number of DSP parameters in the DSP code currently loaded on the specified detChan. This routine is typically used in conjunction with [xiaGetParams()] to allocate the proper amount of memory.

Parameters:

detChan detChan to get number of parameters from. Must be a single detChan.
numParams Pointer in which to return the number of parameters.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel.
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_BAD_TYPE	detChan is a set, not a single detChan
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
unsigned short numParams = 0;

/* Assumes that a system has been loaded and
 * that xiaStartSystem() has already been
 * called.

status = xiaGetNumParams(0, &numParams);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of DSP parameters */
}

printf("detChan 0 has %u DSP parameters.\n", numParams);
```

xiaGetParamData()

```
int xiaGetParamData(int detChan, char *name, void *value)
```

Gets the parameter information specified by the name-value pair.

name	value Type	Description
names	`char \*\*`	The names of all of the DSP parameters for the specified detChan. T
values	`unsigned short \*`	The values of all of the DSP parameters for the specified detChan. T
access	`unsigned short \*`	The access information for all of the DSP parameters for the specifie
lower_bounds	`unsigned short \*`	The lower bounds information for all of the DSP parameters for the s
upper_bounds	`unsigned short \*`	The upper bounds information for all of the DSP parameters for the

Parameters:

detChan detChan to get the DSP parameter names and values from. Must be a single detChan.

name Name from table above corresponding to the desired type of DSP parameter information.

value Value to set the DSP parameter information to, cast to `void \*`. See the table above require data types and memory allocation information for each `name`.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_BAD_TYPE	detChan is set, not a single detChan
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
unsigned short numParams = 0;
unsigned short i;
unsigned short *values = NULL;
unsigned short *access = NULL;
unsigned short *lowBounds = NULL;
unsigned short *highBounds = NULL;
char **names = NULL;

/* Assume that a system has been loaded and
 * that xiaStartSystem() has been called.
 */

status = xiaGetNumParams(0, &numParams);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of DSP parameters */
}

names = (char **)malloc(numParams * sizeof(char *));
if (names == NULL) {
    /* Out of memory trying to create names */
}

for (i = 0; i < numParams, i++) {
    names[i] = (char *)malloc(MAXSYMBOL_LEN * sizeof(char));
    if (names[i] == NULL) {
        /* Out of memory trying to create names[i] */
    }
}
```

```
values = (unsigned short *)malloc(numParams * sizeof(unsigned short));
if (values == NULL) {
    /* Out of memory trying to create values */
}

access = (unsigned short *)malloc(numParams * sizeof(unsigned short));
if (access == NULL) {
    /* Out of memory trying to create access */
}

lowBounds = (unsigned short *)malloc(numParams * sizeof(unsigned short));
if (lowBounds == NULL) {
    /* Out of memory trying to create lowBounds */
}

highBounds = (unsigned short *)malloc(numParams * sizeof(unsigned short));
if (highBounds == NULL) {
    /* Out of memory trying to create high bounds */
}

status = xiaGetParamData(0, "names", (void *)names);
if (status != XIA_SUCCESS) {
    /* ERROR getting DSP parameter names */
}

status = xiaGetParamData(0, "values", (void *)values);
if (status != XIA_SUCCESS) {
    /* ERROR getting DSP parameter values */
}

status = xiaGetParamData(0, "access", (void *)access);
if (status != XIA_SUCCESS) {
    /* ERROR getting DSP parameter access information */
}

status = xiaGetParamData(0, "lower_bounds", (void *)lowBounds);
if (status != XIA_SUCCESS) {
    /* ERROR getting DSP parameter lower bounds information */
}

status = xiaGetParamData(0, "upper_bounds", (void *)highBounds);
if (status != XIA_SUCCESS) {
    /* ERROR getting DSP parameter upper bounds information */
}
```

```
for (i = 0; i < numParams; i++) {
    printf("%s = %u: %u %u %u\n", names[i], values[i], access[i],
        lowBounds[i], highBounds[i]);
}

for (i = 0; i < numParams; i++) {
    free((void *)names[i]);
}

free((void *)names);
names = NULL;

free((void *)values);
values = NULL;

free((void *)access);
access = NULL;

free((void *)lowBounds);
lowBounds = NULL;

free((void *)highBounds);
highBounds = NULL;
```

xiaGetParamName()

```
int xiaGetParamName(int detChan, unsigned short index, char *name)
```

Returns the DSP parameter name located at the specified index. This routine should be used in place of [xiaGetParams()] when interfacing to Handel with a language that doesn't support the passing of string arrays to DLLs, like Visual Basic.

Typical use is to get the number of DSP parameters for the detChan with a call to xiaGetNumParams(). Then loop for 0 to numParams –1 to read in all of the DSP parameter names.

Parameters:

detChan detChan to get the DSP parameter name from. Must be a single detChan.

index Index of the desired DSP parameter name in the complete DSP parameter name list.

name A string of the proper length, MAXSYMBOL_LEN (defined in handel_generic.h)

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_BAD_TYPE	detChan is set, not a single detChan
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
unsigned short numParams = 0;
unsigned short i;
char name[MAXSYMBOL_LEN];

/* Assume that a valid system has been setup here */

status = xiaGetNumParams(0, &numParams);
if (status != XIA_SUCCESS) {
    /* ERROR getting number of DSP params */
}

for (i = 0; i < numParams; i++) {
    status = xiaGetParamName(0, i, name);
    if (status != XIA_SUCCESS) {
        /* ERROR getting DSP parameter name at index i */
    }

    printf("DSP Parameter Name at index = %u: %s\n", i, name);
}
```

Run Control

Routines to start and stop runs and read data from the device.

- xiaStartRun()
- xiaStopRun()
- xiaGetRunData()
- xiaDoSpecialRun()
- xiaGetSpecialRunData()

xiaStartRun()

```
int xiaStartRun(int detChan, unsigned short resume)
```

Starts a run on the specified detChan or detChan set. For some products, e.g. XMAP, even if a single channel is specified, all channels for that module will have a run started. This is an intrinsic property of the hardware and there is no way to circumvent it in the software.

The resume parameter controls whether the MCA memory will be cleared prior to starting the run.

Parameters:

detChan detChan or detChan set on which to start a run.
resume 0 to clear the MCA. 1 to resume without clearing the MCA.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;

/* Set up a valid system here */

status = xiaStartRun(0, 0);
if (status != XIA_SUCCESS) {
    /* ERROR starting a run */
}
```

xiaStopRun()

```
int xiaStopRun(int detChan)
```

Stops a run on the specified channel(s). For some products, this will stop a run on all of the channels in detChan's module.

Parameters:

detChan detChan or detChan set on which to stop the run.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_TIMEOUT	**(For Saturn/Mercury)** Timeout waiting for run to end. (BUSY not equal t
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;

/* Set up a valid system here */

status = xiaStartRun(0, 0);

if (status != XIA_SUCCESS) {
    /* ERROR starting run */

}

/* Wait for data to be collected */

status = xiaStopRun(0);

if (status != XIA_SUCCESS) {
    /* ERROR stopping run */
}
```

xiaGetRunData()

```
int xiaGetRunData(int detChan, char *name, void *value)
```

Returns run data for the channel. This is the primary routine to read out spectra or any other data pertaining to a run.

Parameters:

detChan detChan to get data from. Must be a single detChan.
name Type of data to get. Reference the product-specific Handel manuals for complete lists by product.

value Variable to return the data in, cast into a void *. See the product-specific Handel manuals for the required data type for each name.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_BAD_TYPE	detChan is a set, not a single detChan.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
unsigned long mcaSize = 0;
unsigned long *mca = NULL;

/* Set up a valid system here. */

status = xiaStartRun(0, 0);
if (status != XIA_SUCCESS) {
    /* ERROR starting run */
}

/* Wait for data to collect */

status = xiaStopRun(0);
if (status != XIA_SUCCESS) {
    /* ERROR stopping run */
}

/* Now we can read out the data */
status = xiaGetRunData(0, "mca_length", (void *)&mcaSize);
if (status != XIA_SUCCESS) {
    /* ERROR reading out mca_length */
}

mca = (unsigned long *)malloc(mcaSize * sizeof(unsigned long));
if (mca == NULL) {
    /* Ran out of memory */
}

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

```
if (status != XIA_SUCCESS) {
    /* ERROR reading our mca data */
}

/* Process spectrum data here */
```

xiaDoSpecialRun()

```
int xiaDoSpecialRun(int detChan, char *name, void *info)
```

Starts and stops a special run on the specified channel. Special runs include various diagnostic, setup, and calibration routines, as opposed to MCA and mapping, and list mode runs started with xiaStartRun().

This routine will block program execution until the special run is complete or an internal timeout occurs. (Internal timeouts vary between XIA processors and special run types.)

The types of special runs available for the various products are listed in the Special Run Data tables in the product-specific Handel manuals along with the composition of the info array for each type of run. The info array is used to provide additional parameters for some special run types.

> Some special runs require a call to xiaGetSpecialRunData() in order to properly stop the run. Reference the product-specific Handel manuals for a designation of this behavior by special run type.

Parameters:

detChan detChan or detChan set to start the special run on.
name Type of special run to perform. See Appendix E for a complete list.
info Additional information (if required) for the special run cast to `void \*`.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_TRACE_OOR	**(For Saturn/Mercury)** The specified TRACEWAIT time is out of range.
XIA_TIMEOUT	**(For Saturn/Mercury)** Timeout waiting for the special run to finish. (BUS
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;

/* tracewait of 5 microseconds (in nanoseconds) */
double info[2] = { 0.0, 5000.0 };

/* Set up a valid system here */

/* Acquire an ADC trace */
status = xiaDoSpecialRun(0, "adc_trace", (void *)info);
if (status != XIA_SUCCESS) {
    /* ERROR doing ADC trace run */
}
```

xiaGetSpecialRunData()

```
int xiaGetSpecialRunData(int detChan, char *name, void *value)
```

Returns data associated with a special run. For most special runs this also stops the special run that was started by xiaDoSpecialRun(). This routine must be called after `xiaDoSpecialRun` for some types of special runs.

See the Special Run tables in the product-specific Handel manuals for information on which special runs require the data to be read out and for the names and data types of the special run data to read out.

Parameters:

detChan detChan to get data from. Must be a single detChan.
name Type of data to get.
value Variable to return the data in, cast into a void *.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan does not exist or is not associated with a known module.
XIA_UNKNOWN_BOARD	Board type corresponding to detChan does not exist in Handel
XIA_XERXES	Error reported by Xerxes routine called by Handel.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;
unsigned long adcLen = 0;;
unsigned long *adc = NULL;
```

```
/* tracewait of 5 microseconds (in nanoseconds) */
double info[2] = { 0.0, 5000.0 };

/* Set up a valid system here */

/* Want to acquire an ADC trace */
status = xiaDoSpecialRun(0, "adc_trace", (void *)info);
if (status != XIA_SUCCESS) {
    /* ERROR doing ADC trace run */
}

status = xiaGetSpecialRunData(0, "adc_trace_length", (void *)&adcLen);
if (status != XIA_SUCCESS) {
    /* ERROR getting length of ADC trace */
}

adc = (unsigned long *)malloc(adcLen * sizeof(unsigned long));
if (adc == NULL) {
    /* ERROR allocating memory for adc trace */
}

/* Stops run and gets data */
status = xiaGetSpecialRunData(0, "adc_trace", (void *)adc);
if (status != XIA_SUCCESS) {
    /* ERROR getting ADC trace */
}

/* Post-process ADC trace data */
```

System Configuration

Routines to load and save the entire system configuration from and to files.

- xiaLoadSystem()
- xiaSaveSystem()

xiaLoadSystem()

```
int xiaLoadSystem(char *type, char *filename)
```

Loads a configuration file of the specified type and name. Since only one type is supported (handel_ini), it is typically more convenient to call xiaInit(), which both initializes the library and loads a handel_ini file, instead of xiaInitHandel() and `xiaLoadSystem`.

Parameters:

type Configuration file type to load. Currently only handel_ini is supported. See INI Files for a detailed description of the handel_ini format.

filename Name of file to read configuration from. If the name is specified as NULL, then Handel assumes that the file to be loaded is named xia.ini.

Return Codes

`xiaLoadSystem` calls many internal routines to initialize the various components of the system and may propagate additional error codes not listed here. Specific error codes are useful for debugging during development, but simply checking against `XIA_SUCCESS` provides adequate information for control flow in most user programs.

Code	Description
XIA_FILE_TYPE	Specified file type is not a supported or valid format to load.
XIA_OPEN_FILE	Error opening file
XIA_NOSECTION	Section missing in file
XIA_FORMAT_ERROR	File is improperly formatted
XIA_FILE_RA	File is missing required information

Usage

```
int status;
status = xiaLoadSystem("handel_ini", "my_config.ini");
if (status != XIA_SUCCESS) {
    /* ERROR loading configuration file */
}
```

xiaSaveSystem()

```
int xiaSaveSystem(char *type, char *filename)
```

Saves the current system configuration to the specified file and with the specified format.

Parameters:

type Configuration file type to save. handel_ini is the only supported format. See INI Files for a detailed description of the format.

filename Name of file to save to.

Return Codes:

Code	Description
XIA_FILE_TYPE	Specified file type is not a supported or valid format to load.
XIA_OPEN_FILE	Error opening file
XIA_MISSING_TYPE	Unknown detector type
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage

```
int status;

/* Set up a valid system */

status = xiaSaveSystem("handel_ini", "my_config.ini");
if (status != XIA_SUCCESS) {
    /* ERROR saving system configuration */
}
```

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.

By default Handel logs errors to stdout. Routines in this section allow the user to configure the output stream and increase verbosity to record additional internal information.

- xiaEnableLogOutput()
- xiaSuppressLogOutput()
- xiaSetLogLevel()
- xiaSetLogOutput()
- xiaCloseLog()

xiaEnableLogOutput()

```
int xiaEnableLogOutput(void)
```

Enables logging to the output stream as configured by a call to xiaSetLogOutput(). By default, logging is enabled and is directed to standard out, so you would only need to call this routine if you previously called xiaSuppressLogOutput() and wanted to re-enable logging.

> If Handel has not been initialized then it will be initialized silently by this routine.

Return Codes:

Code	Description
XIA_MD	Error reported by MD routine called by Handel.

Usage

```
int status;

status = xiaInitHandel();
if (status != XIA_SUCCESS) {
    /* ERROR initializing Handel */
}

/* This call is redundant since logging is enabled by default. */
status = xiaEnableLogOutput();
if (status != XIA_SUCCESS) {
    /* ERROR enabling log output */
}
```

xiaSuppressLogOutput()

```
int xiaSuppressLogOutput(void)
```

Stops log output from being written to the current stream.

> If Handel has not been initialized then it will be initialized silently by this routine.

Return Codes:

Code	Description
XIA_MD	Error reported by MD routine called by Handel

Usage

```
int status;
status = xiaInitHandel();
if (status != XIA_SUCCESS) {
    /* ERROR initializing Handel */
}

status = xiaSuppressLogOutput();
```

```
if (status != XIA_SUCCESS) {
    /* ERROR suppressing log output */
}
```

xiaSetLogLevel()

```
int xiaSetLogLevel(int level)
```

Sets the level of logging that will be reported to the log output stream. The levels are defined as constants in the file md_generic.h.

MD_DEBUG All messages, including information only relevant to the developers at XIA. This level generates significantly more output and can should typically be used only if needed in debugging an issue with XIA.

MD_INFO All messages except for debug level messages.

MD_WARNING All warning and error messages. Warning message indicate conditions where a routine will keep executing but the user should probably fix the condition warned about. This is the most useful level of debugging since warning messages often indicate subtle user errors that aren't catastrophic but may lead to an unexpected state.

MD_ERROR Only messages that cause a routine to end its execution early. An error must be fixed before the routine that caused the original error is called again.

If Handel has not been initialized then it will be initialized silently by this routine.

Parameters:

level Level of logging desired. See above for discussion of allowed values.

Return Codes:

Code	Description
XIA_MD	Error reported by MD routine called by Handel

Usage

```
int status;
status = xiaInitHandel();
if (status != XIA_SUCCESS) {
    /* ERROR initializing Handel */
}
```

```
status = xiaSetLogLevel(MD_INFO);
if (status != XIA_SUCCESS) {
    /* ERROR setting log level to MD_INFO */
}
```

xiaSetLogOutput()

```
int xiaSetLogOutput(char *filename)
```

Sets the output stream for the logging routines. Defaults to stdout.

The literal strings "stdout" and "stderr" redirect to the corresponding console device. All other names are opened as files.

Parameters:

filename Name of file to redirect logging messages, or "stdout" or "stderr".

Usage

```
int status;
status = xiaInitHandel();
if (status != XIA_SUCCESS) {
    /* ERROR initializing Handel */
}

xiaSetLogOutput("my_log.txt");
```

xiaCloseLog()

```
int xiaCloseLog(void)
```

Closes the logging stream. This effectively closes any open log file and redirects to stdout. Call this routine when cleaning up program resources if you set log output to a file with xiaSetLogOutput().

Dynamically setting up the system

> The dynamic configuration methods could potentially make the system unstable if used after starting up. Modification to system configuration can be safely done by loading a new configuration file with xiaLoadSystem(), then calling xiaStartSystem().

For advanced or dynamic system configuration, all of the information associated with these structures is modified through a uniform set of routines: `xiaNew{NAME}`, `xiaAdd{NAME}Item`, `xiaModify{NAME}Item`, `xiaGet{NAME}Item` and `xiaRemove{NAME}`, where {NAME} is one of "Detector", "Firmware", or "Module".

Routines for dynamically configuring information about physical detectors.

- xiaNewDetector()
- xiaAddDetectorItem()
- xiaModifyDetectorItem()
- xiaRemoveDetector()
- xiaNewFirmware()
- xiaAddFirmwareItem()
- xiaModifyFirmwareItem()
- xiaRemoveFirmware()
- xiaNewModule()
- xiaAddModuleItem()
- xiaModifyModuleItem()
- xiaRemoveModule()
- xiaAddChannelSetElem()
- xiaRemoveChannelSetElem()
- xiaRemoveChannelSet()

xiaNewDetector()

```
int xiaNewDetector(char *alias)
```

Creates a new detector with the name alias that can be referenced by other routines such as xiaAddDetectorItem(), xiaGetDetectorItem(), xiaModifyDetectorItem() and xiaRemoveDetector().

Parameters:

alias Name of new detector to be added to system.

Return Codes:

Code	Description
XIA_ALIAS_SIZE	Length of alias exceeds the maximum allowed length
XIA_ALIAS_EXISTS	A detector with the specified alias already exists
XIA_NOMEM	Ran out of memory trying to create new detector

Usage

```
int status;
status = xiaNewDetector("detector1");
if (status != XIA_SUCCESS) {
    /* ERROR Creating new detector */
}
```

xiaAddDetectorItem()

```
int xiaAddDetectorItem(char *alias, char *name, void *value)
```

Adds information about the detector using name-value pairs. Reference the Detector Items list for valid names and required value types.

An error is returned if the specified alias has not been created with a previous call to xiaNewDetector().

Parameters:

alias A valid detector alias.
name Name from table above corresponding to the information the user wishes to set.
value Value to set the corresponding detector information to, cast to a void *. See Usage for examples of using void pointers in this context.

Return Codes:

Code	Description
XIA_BAD_VALUE	`value` could not be converted to the required type.
XIA_NO_ALIAS	Specified alias does not exist.
XIA_BAD_NAME	`name` is not a valid detector item.

Usage

```
int status;
unsigned int number_of_channels = 1;
double gain = 5.6;

/* Assume that detector already created with alias = detector1 */
status = xiaAddDetectorItem("detector1", "number_of_channels",
                            (void *)&number_of_channels);
if (status != XIA_SUCCESS) {
```

```
    /* ERROR Adding number_of_channels */
}

status = xiaAddDetectorItem("detector1", "channel0_gain", (void *)&gain);
if (status != XIA_SUCCESS) {
    /* ERROR adding gain */
}

status = xiaAddDetectorItem("detector1", "channel0_polarity", (void *)"pos");
if (status != XIA_SUCCESS) {
    /* ERROR adding polarity */
}
```

xiaModifyDetectorItem()

```
int xiaModifyDetectorItem(char *alias, char *name, void *value)
```

Modifies the value of a detector item. See Detector Items for the table of allowed names and value types.

> You must call xiaStartSystem() in order for the changed values to be reflected in the hardware.

Parameters:

alias A valid detector alias
name Name of value to modify. See description for allowed names.
value Value to change current setting to, cast into a void pointer. See the Usage section for an example of using a void pointer in this context.

Return Codes:

Code	Description
XIA_BAD_VALUE	Value is NULL or there is an error with it
XIA_BAD_NAME	Specified name is not allowed to be modified or is invalid
XIA_NO_ALIAS	Specified alias does not exist

Usage

```
int status;
double new_gain = 5.7;

/* Assume a detector with alias "detector1" has already been created.
 */
```

```
status = xiaModifyDetectorItem("detector1", "channel0_gain", (void *)&new_gain);
if (status != XIA_SUCCESS) {
    /* ERROR Modifying channel 0 gain */
}

status = xiaStartSystem();
if (status != XIA_SUCCESS) {
    /* ERROR starting system */
}
```

xiaRemoveDetector()

```
int xiaRemoveDetector(char *alias)
```

Removes a detector from the system.

Parameters:

alias A valid detector alias

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist

Usage

```
int status;

/* Remove detector w/ alias detector1 */
status = xiaRemoveDetector("detector1");
if (status != XIA_SUCCESS) {
    /* ERROR removing detector */
}
```

xiaNewFirmware()

```
int xiaNewFirmware(char *alias)
```

Creates a new firmware with the name alias that can be referenced by other routines such as xiaAddFirmwareItem(), xiaGetFirmwareItem(), xiaModifyFirmwareItem() and xiaRemoveFirmware().

Parameters:

alias Name of new firmware to be added to system

Return Codes:

Code	Description
XIA_ALIAS_SIZE	Length of alias exceeds the maximum allowed length
XIA_ALIAS_EXISTS	Firmware with the specified alias already exists
XIA_NOMEM	Ran out of memory trying to create new firmware

Usage

```
int status;
status = xiaNewFirmware("firmware1");
if (status != XIA_SUCCESS) {
    /* ERROR Creating new firmware */
}
```

xiaAddFirmwareItem()

```
int xiaAddFirmwareItem(char *alias, char *name, void *value)
```

Adds information about the firmware using name-value pairs. Firmware can be divided into two categories: those that use the FDD and those that don't. Each category has its own set of name-value pairs. Reference the Firmware Items lists for details.

The standard procedure is to use an XIA supplied FDD file. If one is defining custom firmware there are some key points to remember:

- The PTRRs may not overlap
- Once a call to xiaAddFirmwareItem() has been made with the name "ptrr", all calls to xiaAddFirmwareItem() using names listed below "ptrr" in the table above will be added to the most recently added "ptrr". The implication of this is that you must add all of the PTRR information before switching to the next PTRR. However, any information that is omitted may be added using xiaModifyDetectorItem().

Parameters:

alias A valid firmware alias
name Name from table above corresponding to the information the user wishes to set

value Value to set the corresponding firmware information to, cast into a void *. See Usage section for examples of using void pointers in this context.

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist
XIA_BAD_VALUE	Error with value passed in
XIA_BAD_NAME	Specified name is invalid
XIA_BAD_PTRR	The PTRR to be added already exists

Usage

```
int status;
unsigned short ptrr = 0;
double min_ptime = 0.25;
double max_ptime = 1.25;

/* Only illustrate how to create firmware using PTRRs since using the
 * FDD files is trivial. Assume firmware "firmware1" already
 * created.
 */

status = xiaAddFirmwareItem("firmware1", "ptrr", (void *)&ptrr);
if (status != XIA_SUCCESS) {
    /* ERROR Adding PTRR */
}

status = xiaAddFirmwareItem("firmware1", "min_peaking_time", (void *)&min_ptime);
if (status != XIA_SUCCESS) {
    /* Error Adding minimum peaking time to PTRR 0 */
}

status = xiaAddFirmwareItem("firmware1", "max_peaking_time", (void *)&max_ptime);
if (status != XIA_SUCCESS) {
    /* ERROR Adding maximum peaking time to PTRR 0 */
}

status = xiaAddFirmwareItem("firmware1", "fippi", (void *)"fxpd0g.fip");
if (status != XIA_SUCCESS) {
    /* ERROR Adding FiPPI file to PTRR 0 */
}

status = xiaAddFirmwareItem("firmware1", "dsp", (void *)"x10p.hex");
```

```
if (status != XIA_SUCCESS) {
    /* ERROR Adding DSP file to PTRR 0 */
}

ptrr = 1;
min_ptime = 1.25;
max_ptime = 5.0;
status = xiaAddFirmwareItem("firmware1", "ptrr", (void *)&ptrr);

if (status != XIA_SUCCESS) {
    /* ERROR Adding new PTRR */
}

status = xiaAddFirmwareItem("firmware1", "min_peaking_time", (void *)&min_ptime);
if (status != XIA_SUCCESS) {
    /* ERROR Adding minimum peaking time to PTRR 1 */
}
```

xiaModifyFirmwareItem()

```
int xiaModifyFirmwareItem(char *alias, unsigned short ptrr, char *name, void *value)
```

Modifies a firmware set item. See Firmware Items for a table of names. The overall disposition of the firmware may not be modified, i.e., if a firmware is already using the FDD, it may not use PTRRs; a new firmware alias should be created for the PTRRs.

> You must call xiaDownloadFirmware() or xiaStartSystem() if you wish to update the firmware that is currently downloaded to the processor.

Parameters:

alias A valid firmware alias

ptrr The PTRR that corresponds to the information to be modified. Not all names to be modified require a PTRR, in which case it may be set to NULL.

name Name of value to modify, which must come from the tables in Firmware Items.

value Value to change current setting to, cast to void *. Reference the tables for the data type to use for the actual variable.

Return Codes:

Code	Description
XIA_BAD_VALUE	Value is NULL or there is an error with it
XIA_NO_ALIAS	Specified alias does not exist
XIA_BAD_NAME	Specified name is invalid

Usage

```
int status;

/* Add firmware called "firmware1" here that uses PTRRs. Assume that
 * there is a single PTRR with the value of 0.
 */

status = xiaModifyFirmwareItem("firmware1", 0, "dsp", (void *)"d2xr0105.hex");
if (status != XIA_SUCCESS) {
    /* ERROR Modifying PTRR 0 DSP file name */
}
```

xiaRemoveFirmware()

```
int xiaRemoveFirmware(char *alias)
```

Removes a firmware set from the system.

Parameters:

alias A valid firmware alias

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist

Usage

```
int status;

/* Assume firmware w/ alias "firmware1" already exists */
status = xiaRemoveFirmware("firmware1");
if (status != XIA_SUCCESS) {
    /* ERROR Removing firmware */
}
```

xiaNewModule()

```
int xiaNewModule(char *alias)
```

Creates a new module with the name alias that can be referenced by other routines such as xiaAddModuleItem(), xiaGetModuleItem(), xiaModifyModuleItem() and xiaRemoveModule().

Parameters:

alias Name of new module to be added to system

Return Codes:

Code	Description
XIA_ALIAS_SIZE	Length of alias exceeds the maximum allowed length
XIA_ALIAS_EXISTS	A module with the specified alias already exists
XIA_NOMEM	Ran out of memory trying to create a new module

Usage

```
int status;
status = xiaNewModule("module1");
if (status != XIA_SUCCESS) {
    /* ERROR Creating new module */
}
```

xiaAddModuleItem()

```
int xiaAddModuleItem(char *alias, char *name, void *value)
```

Adds informaton about the module using name-value pairs. Reference the Module Items list for valid names and required value types.

When setting up a new module, always add the `module_type` first, then add the other details. Certain items in this table apply only to certain module types. An error is returned if you attempt to add an item that does not apply.

Parameters:

alias A valid module alias
name Name from the tables above corresponding to the information that the user wants to set

value Value to set the corresponding module information to, cast into a void pointer. See Usage section for an example of using void pointers in this context.

Return Codes:

Code	Description
XIA_BAD_VALUE	Error with value passed in
XIA_NO_ALIAS	Specified alias does not exist. May refer to module, detector or firmware alia
XIA_BAD_INTERFACE	Specified interface is invalid
XIA_WRONG_INTERFACE	Specified name is not a valid element of the current interface
XIA_INVALID_DETCHAN	Specified detChan does not exist or is invalid
XIA_BAD_TYPE	Internal error. Contact XIA
XIA_BAD_CHANNEL	Specified physical detector channel (see channel_detector{n}) is invalid

Usage

```
int status;
int chan0alias = 0;
unsigned int epp_address = 0x378;
unsigned int num_channels = 1;
double chan0gain = 1.0;

/* Assume that module already created with alias "module1". This
 * example will show how to add a DXP-X10P box to the system.
 */

status = xiaAddModuleItem("module1", "module_type", (void *)"dxpx10p");

if (status != XIA_SUCCESS) {
    /* ERROR Adding module_type */
}

status = xiaAddModuleItem("module1", "interface", (void *)"epp");
if (status != XIA_SUCCESS) {
    /* ERROR Adding interface */
}

status = xiaAddModuleItem("module1", "epp_address", (void *)&epp_address);
if (status != XIA_SUCCESS) {
    /* ERROR Adding epp_address */
}

status = xiaAddModuleItem("module1", "number_of_channels", (void *)&num_channels);
```

```
if (status != XIA_SUCCESS) {
    /* ERROR Adding number_of_channels */
}

/* Here, assume that we have the following aliases defined:
 * detector1, firmware1.
 */
status = xiaAddModuleItem("module1", "channel0_alias", (void *)&chan0alias);
if (status != XIA_SUCCESS) {
    /* ERROR Adding channel0_alias */
}

status = xiaAddModuleItem("module1", "channel0_detector", (void *)"detector1:0");
if (status != XIA_SUCCESS) {
    /* ERROR Adding channel0_detector */
}

status = xiaAddModuleItem("module1", "channel0_gain", (void *)&chan0gain);
if (status != XIA_SUCCESS) {
    /* ERROR Adding channel0_gain */
}

status = xiaAddModuleItem("module1", "firmware_set_chan0", (void *)"firmware1");
if (status != XIA_SUCCESS) {
    /* ERROR Adding firmware_set_chan0 */
}
```

xiaModifyModuleItem()

```
int xiaModifyModuleItem(char *alias, char *name, void *value)
```

Modifies a module item. Allowed names are channel{n}_alias, channel{n}_detector, channel{n}_gain, firmware_set_all and firmware_set_chan{n}. See Module Items for the data types to use for `value`.

> You must call xiaStartSystem() in order for the changed values to be reflected in the hardware.

Parameters:

alias A valid module alias
name Name of value to modify
value Value to change current setting to, cast into a void pointer.

Return Codes:

Code	Description
XIA_NO_MODIFY	Specified name can not be modified
XIA_BAD_VALUE	Error with value passed in
XIA_NO_ALIAS	Specified alias does not exist. May refer to module, detector or firmware alias
XIA_INVALID_DETCHAN	Specified detChan does not exist or is invalid
XIA_BAD_TYPE	Internal error. Contact XIA
XIA_BAD_CHANNEL	Specified physical detector channel (see channel_detector{n}) is invalid

Usage

```
int status;

/* Add a module called "module1" here. See xiaAddModuleItem() for an
 * example of the information in "module1".
 */
status = xiaModifyModuleItem("module1", "firmware_set_all", (void *)"new_firmware");
if (status != XIA_SUCCESS) {
    /* ERROR Modifying firmware_set_all */
}
```

xiaRemoveModule()

```
int xiaRemoveModule(char *alias)
```

Removes a module from the system.

Parameters:

alias A valid module alias

Return Codes:

Code	Description
XIA_NO_ALIAS	Specified alias does not exist

Usage

```
int status;

/* Remove the module with alias module1 */
status = xiaRemoveModule("module1");
if (status != XIA_SUCCESS) {
```

```
    /* ERROR Removing module */
}
```

xiaAddChannelSetElem()

```
int xiaAddChannelSetElem(unsigned int detChan, unsigned int newChan)
```

Adds a detChan to a detector channel set. If the detector channel set doesn't already exist, it will be created. If it already exists, newChan will be added to it.

Parameters:

detChan Detector channel set to, if necessary, create and add newChan to

newChan Existing detector channel (or detector channel set) to add to detChan. CAUTION: This must be a previously created detector channel set or detector channel.

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	newChan doesn't exist yet
XIA_BAD_VALUE	Internal Handel error. Contact XIA.
XIA_BAD_TYPE	Internal Handel error. Contact XIA.

Usage

```
int status;

/* Assume that a module with channel 0 set to detChan = 0 exists.
 * Now, we want to create a detector channel set with detChan = 1
 * that contains detChan = 0. (If this is confusing, consult the
 * Handel Users Manual for a detailed explanation of how detector
 * channels works.
 */
status = xiaAddChannelSetElem(1, 0);
if (status != XIA_SUCCESS) {
    /* ERROR Adding detChan 0 to detector channel set 1 */
}
```

xiaRemoveChannelSetElem()

```
int xiaRemoveChannelSetElem(unsigned int detChan, unsigned int chan)
```

Remove a channel from a detector channel set.

Parameters:

detChan Detector channel number of the set that contains chan
chan Detector channel to remove from detChan

Return Codes:

Code	Description
XIA_INVALID_DETCHAN	Specified detChan(s) are invalid

Usage

```
int status;

/* Assume that a detector channel set (detChan = 0) has been created
 * with detChans 1 & 2 as elements.
 */

/* Remove detChan = 1 from detChan = 0 */
status = xiaRemoveChannelSetElem(0, 1);
if (status != XIA_SUCCESS) {
    /* ERROR Removing detChan */
}
```

xiaRemoveChannelSet()

```
int xiaRemoveChannelSet(unsigned int detChan)
```

Removes a detector channel set. Detector channels contained in the set are not removed; they are only dereferenced them from the specified set.

Parameters:

detChan A valid detector channel set to be removed.

Return Codes:

Code	Description
XIA_WRONG_TYPE	Specified detChan is not a detector channel set
XIA_INVALID_DETCHAN	Specified detChan is invalid

Code	Description
XIA_BAD_TYPE	Internal Handel error. Contact XIA.

Usage

```
int status;

/* Assume that a detector channel set (detChan = 0) has been created
 * with detChans 1 & 2.
 */

status = xiaRemoveChannelSet(0);
if (status != XIA_SUCCESS) {
    /* ERROR Removing detector channel set (detChan = 0) */
}
```

Legal

Copyright 2005-2016 XIA LLC

All rights reserved

All trademarks and brands are property of their respective owners.

Licenses

Handel

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of XIA LLC nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED

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

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