Handel API

XIA Hardware Description Layer
Revision 0.0.7
June 2002

X-ray Instrumentation Associates
8450 Central Ave
Newark, CA 94560 USA

www.xia.com
software_support@xia.com

1 Introduction

This document is still preliminary and intended only for outlining future software designs. Currently, Handel is in an alpha state and in the midst of an active development cycle. If software is written using this specification, the writer should expect to make changes to both calling structures and routine names in the near future to fully conform to the final API. XIA will make every attempt to only change routines with minimal impact on users, but we cannot guarantee anything at this time. Additionally, since progress is happening rapidly in this phase of development, XIA will only be supporting the most recent version of Handel that has been released.

This document is intended to aid the data collection programmer in developing software that controls and reads out data from all XIA x-ray and gamma ray processors. Handel utilizes the functions defined in the XerXes library, which is provided free of charge from XIA, to build a higher level interface to the hardware, requiring as little knowledge about the hardware as is reasonable by the end-user. This library is written in ANSI C and should work on any platform that has a C compiler available, however the mechanism to communicate with the hardware (e.g. SCSI or EPP drivers) is not guaranteed by XIA and may be left to the end-user to implement.

The organization of this document is as follows: § 2 introduces some terms that are used throughout this document; § 3 describes the calling conventions used by the routines that are included with the host software release; § 4 describes the Handel routines. The final section, § 5, describes all of the files included in the Handel software release.

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

1) Your name
2) Your organization
3) XIA hardware being used
4) Version of the library
5) OS
6) Description of the problem
7) Steps to take to re-create the bug
8) Other relevant information

2 General Terms

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

Hardware interface: This is the method that each XIA module uses to communicate with the host computer. We currently support both EPP and CAMAC on most Linux and Windows operating systems.

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.

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.

FiPPI: This is the field programmable gate array (FPGA) in which the **Fi**lter, **P**eak detection, **P**ileup **I**nspection logic is implemented. Like the DSP, a configuration file must be downloaded to the FiPPI before it can function.

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.

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

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

.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.

XerXes: A companion library distributed free from XIA that deals with lower level interactions between the host software and the XIA processors. Handel is intended to be an intermediary between the user and XerXes.

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 interface between the Handel and Xerxes libraries (PSL driver).

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.

Product Specific Layer (PSL): A set of libraries that provide a method for Handel to interface to Xerxes without having to know the details about every product. These libraries contain the individual logic associated with each product. Host software should **never** call these libraries directly.

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

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.

3 Calling Conventions

Language Interface: Handel is only supported for calls from other C programs or libraries. XIA does not officially support other language interfaces such as Visual Basic or Fortran. We have maintained some effort however, to maintain compatibility with other language interfaces. Please contact XIA for more information.

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 on host computer: We have made no attempt to make the driver routines use "standard" length variables. 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. Development of Handel was done on Windows 98 and Windows 2000 running Pentium 3 and Pentium 4 processors. Some additional work was done using Slackware Linux v8.0 running on a Pentium 3. 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

Currently, 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.

4 Application Programming Interface (API)

Handel uses four global structures to manage a DAQ system: Detector, Firmware, Acquisition Values and Module information. 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 either “Detector”, “Firmware” or “Module”. See the Handel User’s Manual for a complete description of how to use these routines to dynamically configure a system. Alternatively, an .INI file may be used to provide the same information when the library is initialized. See Appendix B for a description of the .INI file format or consult the Handel User’s Manual for a complete description of how to use .INI file.

xiaInit

Syntax:

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

Description:

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 file (in "handel_ini" format) to be loaded.

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 = xiaIni("myFile.ini");

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

xiaInitHandel

Syntax:

```
int xiaInitHandel(void)
```

Description:

Initializes 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 */
}
```

xiaNewDetector

Syntax:

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

Description:

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

Syntax:

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

Description:

Adds information about the detector using name-value pairs.

name	value Type	Description
number_of_channels	unsigned int	The number of detector elements. **Must be specified first**.
channel{n}_gain	double	The preamplifier gain (in mV/keV) for channel n, where n is in the range 0 to (number_of_channels – 1).
channel{n}_polarity	null-terminated string	The polarity of detector channel n, where n is in the range 0 to (number_of_channels – 1). Handel recognizes "+" and "pos" for positive polarity and "-" and "neg" for negative polarity.
type	null-terminated string	The type of preamplifier this detector has. Currently supported values are "reset" and "rc_feedback".
type_value	double	The value associated with the preamplifier type. For instance, "reset" detectors should pass in the value of the reset time in μs, while "rc_feedback" detectors should pass in the 1/e decay time in μs.

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

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 into a void *. See Usage section for examples 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
XIA_BAD_NAME	Specified name is invalid

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

Syntax:

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

Description:

Modify a subset of the total detector information. The user must call xiaStartSystem() again in order to have the change in values reflected in the hardware. The allowed name-value pairs that can be modified are channel{n}_gain, channel{n}_polarity and type_value.

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 */
}
```

xiaGetNumDetectors

Syntax:

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

Description:

Returns the number of detectors currently defined in the system.

Parameters:

numDet

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

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

Syntax:

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

Description:

Returns a list of the aliases of the detectors currently defined in the system. The proper amount of memory must be allocated for "detectors". Typically this is done by calling xiaGetNumDetectors() and using the number of detectors to initialize the string array. See the Usage section for an example of how this is done.

Parameters:

detectors

A string array of the proper length: 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

Syntax:

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

Description:

This routine serves as a replacement of the routine xiaGetDetectors() 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 xiaGetDetectors() is that xiaGetDetectors() returns a list of all of the detector aliases that are currently defined in the system. xiaGetDetectors_VB() 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 in order to get all of the detector aliases in the system. See the Usage section for an example of how this is done. User must allocate the proper amount of memory for the alias.

Parameters:

index

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

alias

Alias of the detector located at the specified index. User must allocate the proper amount of memory. Typically, the proper amount of memory is MAXALIAS_LEN, located in handel_generic.h.

Return Codes:

Code	Description
XIA_BAD_INDEX	The specified 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

Syntax:

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

Description:

Retrieve current information from detector settings. All of the names that are listed in xiaAddDetectorItem() may be retrieved using this routine.

Parameters:

alias

A valid detector 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 table for xiaAddDetectorItem() for more information. Also, 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	Internal Handel error. Contact XIA.
XIA_BAD_NAME	Specified name is invalid

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);
```

xiaRemoveDetector

Syntax:

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

Description:

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;

/* Create a detector w/ alias detector1 */

status = xiaRemoveDetector(“detector1”);

if (status != XIA_SUCCESS)
{
      /* ERROR removing detector */
}
```

xiaNewFirmware

Syntax:

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

Description:

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

Syntax:

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

Description:

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:

FDD

name	type Value	Description
filename	null-terminated string	Name of FDD file to be used. This file will be searched for using the standard methods. See §3 for more information on how files are found by Handel.
keyword	null-terminated string	Each time keyword is used as a name, another keyword is appended to the list associated with this firmware. CAUTION: Keywords may not be removed from the list once they are added. These keywords will be used by Handel when searching the FDD file for the proper firmware. Handel always adds a keyword associated with the detector type. (Optional)

No FDD

name	type Value	Description
mmu	null-terminated string	The filename of the memory management unit, if present. (Optional)
ptrr	unsigned short	A unique identifier for a Peaking Time Range Reference. Each firmware should have several PTRRs to cover the full peaking time range. All subsequent calls to xiaAddFirmwareItem() will add information to the specified PTRR until another PTRR is added.
min_peaking_time	double	The minimum peaking time value for the current PTRR in μs.
max_peaking_time	double	The maximum peaking time value for the current PTRR in μs.
fippi	null-terminated string	The filename of the FiPPI program to be downloaded for this PTRR.
dsp	null-terminated string	The filename of the DSP program to be downloaded for this PTRR
user_fippi	null-terminated string	The filename of the user-defined FiPPI program to be downloaded for the PTRR. Not available with all products. (Optional)
filter_info	unsigned short	Add another filter information value to the existing information. Currently, there is no way to remove filter information (selectively) after it has been added.

The standard procedure is to use an XIA supplied FDD file, however 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 */
}

/* Etc... */
```

xiaModifyFirmwareItem

Syntax:

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

Description:

Modify the firmware information. The user must call xiaDownloadFirmware() or xiaStartSystem() if they wish to update the firmware that is currently downloaded to the processor. See xiaAddFirmwareItem() for a table of names. However, 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.

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.

value

Value to change current setting to, cast into a void pointer. See for 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_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 */
}
```

xiaGetNumFirmwareSets

Syntax:

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

Description:

Returns the number of firmware sets currently 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

Syntax:

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

Description:

Returns a list of the aliases of the firmware sets currently defined in the system. The proper amount of memory must be allocated for "firmware". Typically this is done by calling xiaGetNumFirmwareSets() and using the number of detectors to initialize the string array. See the Usage section for an example of how this is done.

Parameters:

firmware

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

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] = %s\n", i, firmware[i]);
}

for (i = 0; i < numFirmware; i++) {

      free((void *)firmware[i]);
}

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

xiaGetFirmwareSets_VB

Syntax:

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

Description:

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. User must allocate the proper amount of memory for the 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. User must allocate the proper amount of memory. Typically, the proper amount of memory is MAXALIAS_LEN, located in 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

Syntax:

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

Description:

Returns the number of PTRRs that are currently defined for the firmware 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

Syntax:

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

Description:

Retrieve current information from firmware settings. All of the names that are listed in xiaAddFirmwareItem(), except for "keyword". Two other parameters may be retrieved that are not in the table described in xiaSetFirmwareItem(): **num_filter**, which is 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. If the user tries to get the "filename" item for a firmware alias that uses PTRRs, a blank string will be returned.

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 table for xiaAddFirmwareItem() for more information. Also, 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);
```

xiaRemoveFirmware

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

Description:
Removes firmware 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

Syntax:

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

Description:

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

Syntax:

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

Description:

Adds informaton about the module using name-value pairs. Currently, XIA supports modules that run on two interfaces: CAMAC and EPP.

name	**value Type**	**Description**
module_type	null-terminated string	The name corresponding to the type of module. The current supported types are “saturn” and “dxp2x”. **This must be specified first.**
interface	null-terminated string	The type of interface for this module. The current supported interfaces are “genericSCSI’, “j73a”, “genericEPP” and “epp”. It is not necessary to specify the interface as Handel is smart enough to 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 DXP-2X 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	signed int	The “detChan” value for channel n. Each physical channel in the entire 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 created by calling xiaNewDetector().
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 Detector section, nor is it the same as the gain set by 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 next table illustrates the names that apply to the specific interfaces.

name	interface	type Value	Description
scsibus_number	genericSCSI, j73a	unsigned int	The SCSI bus number corresponding to the SCSI card being used.
crate_number	genericSCSI, j73a	unsigned int	The crate number as specified on the front panel of the Jorway 73a controller.
slot	genericSCSI, j73a	unsigned int	The slot number in the CAMAC crate that the module is in. Each module should have a different slot number.
epp_address	genericEPP, epp	unsigned int	The address of the EPP port on the host computer. Typically the address is 0x378 or, ocassionally, 0x278.
daisy_chain_id	genericEPP, epp	unsigned int	The daisy chain ID for this module. Should only be specified if the module specifically implements the daisy chain protocol. (Optional)

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 alias depending on the context of the error message.
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

Syntax:

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

Description:

Modify a subset of the total module information. The user must call xiaStartSystem() again in order to have the change in the values reflected in the hardware. The allowed name-value pairs that can be modified are channel{n}_alias, channel{n}_detector, channel{n}_gain, firmware_set_all and firmware_set_chan{n}.

Parameters:

alias

A valid module alias

name

Name of value to modify

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_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 depending on the context of the error message.
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 */
}
```

xiaGetNumModules

Syntax:

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

Description:

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

Syntax:

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

Description:

Returns a list of the aliases of the modules currently defined in the system. The proper amount of memory must be allocated for "modules". Typically this is done by calling xiaGetNumModules() and using the number of modules to initialize the string array. See the Usage section for an example of how this is done.

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] = %s\n”, i, modules[i]);
}

for (i = 0; i < numModules; i++) {

      free((void *)modules[i]);
}

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

xiaGetModules_VB

Syntax:

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

Description:

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. The difference between this routine and xiaGetModules() is that xiaGetModules() returns a list of all of the modules aliases that are currently defined in the system. xiaGetModules_VB() returns a single modules 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 to then loop from 0 to numModule –1 in order to get all of the module aliases in the system. See the Usage section for an example of how this is done. User must allocate the proper amount of memory for the 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. User must allocate the proper amount of memory. Typically, the proper amount of memory is MAXALIAS_LEN, located 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(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

Syntax:

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

Description:

Retrieve current module information from system. All of the names listed in xiaAddModuleItem() may be retrieved using this routine except for firmware_set_all.

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 tables for xiaAddModuleItem() for more information. Also, 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_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);
```

xiaRemoveModule

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

Description:
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;

/* Create a module with alias module1 */

status = xiaRemoveModule(“module1”);

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

xiaAddChannelSetElem

Syntax:

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

Description:

Adds a detChan to a detector channel set. (See the Handel Users Manual for an explanation of detChan and detector channel sets and how they work.) 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

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

Description:
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

Syntax:

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

Description:

Removes a detector channel set. Does not remove detector channels contained in the set, only dereferences 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
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) */
}
```

xiaStartSystem

Syntax:

```
int xiaStartSystem(void)
```

Description:

Downloads the firmware and acquisition values to all active channels in order to set the system up for data taking. This routine must be called after configuring the system either dynamically or using a configuration file. 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.

Return Codes:

Code	Description
XIA_FIRM_BOTH	Both a FDD file and PTRR information have been specified in one of the firmware aliases. Please report this error to XIA since this check should be performed at the configuration stage and has only been left in as a redundancy check.
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 loop exists. This prevents against situations where a detChan (or Set) refers to another detChan (or Set) that then refers back to itself.
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

Syntax:

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

Description:

Downloads the specified firmware to the specified detChan. Currently, Handel recognizes the following firmware types: “dsp”, “fippi”, “user_dsp”, “user_fippi” and “mmu”. However, it will only download “dsp” and “fippi” at this time. 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 needs to be downloaded to the module.

Parameters:

detChan

detChan to download firmware to. May be either a single detChan or detChan set.

type

The type of firmware to be downloaded. Must be “dsp”, “fippi”, “user_dsp”, “user_fippi” or “mmu”.

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 type.
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 */
}
```

xiaSetAcquisitionValues

Syntax:

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

Description:

Translates a high-level acquisition value into the appropriate DSP parameter(s). Appendix A lists 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.

xiaSetAcquisitionValues() is the only mechanism that Handel provides for persistence of the current 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.

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, from Appendix A, to set or a DSP parameter.

value

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

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 product.
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 supported for this board type.
XIA_UNKNOWN_FIRM	**(For Saturn/Mercury)** The specified type of firmware to download is unknown.
XIA_BINS_OOR	**(For Saturn/Mercury)** The specified number of bins is out of range for this board type.
XIA_GAIN_OOR	**(For Saturn/Mercury)** The computed gain value is out of range for this board type.
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage:

```
int status;
double new_threshold = 1000.0;

/* Setup 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

Syntax:

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

Description:

Retrieve the current setting of an acquisition value listed in Appendix A. 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, from Appendix A, to retrieve.

value

Variable to retrieve the corresponding acquisition value into cast into a void pointer. See Usage section for an example of using void pointers in this context.

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

Syntax:

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

Description:

Removes an acquisition value for the specified channel. Handel protects against the removal of any acquisition values that are required for a specific board type.

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 = xiaAddAcquisitionValues(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

Syntax:

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

Description:

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 */

status = xiaUpdateUserParams(0);

if (status != XIA_SUCCESS) {

      /* ERROR updating user parameters */
}
```

xiaGainOperation

Syntax:

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

Description:

This routine supports special gain operations that are usually board-specific. Currently, only the Gamma200 board type implements this routine in a meaningful way.

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. For more information on using void pointers in this context see the Usage section.

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:

```
/* Currently, this routine is not
 * implemented for products other than
 * the Gamma200.
 */
```

xiaGainChange

Syntax:
```
int xiaGainChange(int detChan, double deltaGain)
```

Description:
Scales the dynamic range by a constant factor. For the x-ray products, this routine scales the ADC percent rule by the delta gain amount and then adjusts the other relevant DSP parameters.

NOTE: For calibrating the energy scale use the routine xiaGainCalibrate().

Parameters:
detChan
detChan or detChan set to apply the gain change to
deltaGain
Factor to scale gain by

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_THRESH_OOR	**(For SATURN/MERCURY)** Calculated THRESHOLD is out of range
XIA_UNKNOWN	Internal Handel error. Contact XIA

Usage:
```
int status;

/* Setup a valid system here */

/* Scale gain by a factor of 2 */
status = xiaGainChange(0, 2.0);

if (status != XIA_SUCCESS)
{
      /* ERROR Changing gain */
}
```

xiaGainCalibrate

Syntax:

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

Description:

Adjusts the specified channel's settings in order to scale the energy value by the specified amount Unlike xiaGainChange(), the result of this routine is that the energy value is shifted by deltaGain. xiaGainChange() modifies the absolute step size at the ADC but does not change the energy value.

Note: 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.

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;

/* Setup 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 */
}
```

xiaStartRun

Syntax:

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

Description:

Starts a run on the specified channel(s). For some products, 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. If the resume parameter is set to 0, the MCA memory will be cleared prior to starting the run.

Parameters:

detChan

detChan or detChan set to start a run on.

resume

0 indicates that the run should be started with the MCA cleared, a 1 indicates that the run will 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;

/* Setup a valid system here */

status = xiaStartRun(0, 0);

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

xiaStopRun

Syntax:
```
int xiaStopRun(int detChan)
```

Description:
Stops a run on the specified channel(s). For some products, this may stop a run on all of the channels.

Parameters:
detChan
detChan or detChan set to stop the run on.

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 to 0)
XIA_XERXES	Error reported by Xerxes routine called by Handel
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage:
```
int status;

/* Setup 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

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

Description:
Returns information for the specified channel. The most common idiom associated with this routine is to read out the size of the data, allocate the proper amount of memory and then read out the data itself. This technique is used to minimize the amount of memory that the library allocates. The Usage section illustrates how to use this idiom to read out the spectrum data.

See Appendix D for a listing of the allowed names on a per product basis.

Parameters:

detChan
: detChan to get data from. Must be a single detChan.

name
: Type of data to get. See Appendix D for a complete list.

value
: Variable to return the data in, cast into a void *. See the Usage section for an example of using void pointers in this context.

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;

/* Setup a valid DXP-X10P 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

Syntax:

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

Description:

Starts and stops a special run on the specified channel. This routine will stop 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 Appendix E 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.

CAUTION: Some special runs require a call to xiaGetSpecialRunData() in order to properly stop the run. Please consult Appendix E for a list of the special runs that must call xiaGetSpecialRunData().

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 into a void *. See the Usage section for an example of using void pointers in this context.

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. (BUSY not equal to 0).
XIA_UNKNOWN	Internal Handel error. Contact XIA.

Usage:

```
int status;

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

/* Setup 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 */
}
```

xiaGetSpecialRunData

Syntax:

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

Description:

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 Appendix E for information on which special runs require the data to be read out and for the names of the special run data to read out.

The standard idiom associated with this routine is to read out the size of the data, allocate the proper amount of memory and then read out the data itself. This technique is used to minimize the amount of memory that the library allocates. The Usage section illustrates how to use this idiom to read out the ADC trace data.

Parameters:

detChan

detChan to get data from. Must be a single detChan.

name

Type of data to get. See Appendix E for a complete list.

value

Variable to return the data in, cast into a void *. See the Usage section for an example of using void pointers in this context.

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 };

/* Setup 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 */
```

xiaLoadSystem

Syntax:

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

Description:

Loads a configuration file of the specified type and name. Currently, the only supported type is handel_ini, but other types will be supported in the near future. If the name is specified as NULL, then Handel assumes that the file to be loaded is named xia.ini.

Parameters:

type

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

filename

Name of file to read configuration from.

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_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 */
}
```

* Since Handel is building the configuration from the file using the dynamic configuration routines, all of their error values may be returned as well.

xiaSaveSystem

Syntax:

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

Description:

Saves the current system configuration to the specified file and with the specified format. Currently, Handel only supports writing to the handel_ini format. See Appendix B for a detailed description of the format.

Parameters:

type

Configuration file type to save. Currently only handel_ini is supported. See Appendix B for a detailed description of the handel_ini 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;

/* Setup a valid system */

status = xiaSaveSystem(“handel_ini”, “my_config.ini”);

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

xiaGetParameter

Syntax:

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

Description:

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 and shoule be used with caution.

Parameters:

detChan

detChan to get the value from. Must be a single detChan.

name

Name of DSP parameter to retrieve

value

Pointer to variable to store the value of the DSP parameter in.

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;

/* Setup a valid system here */
status = xiaGetParameter(0, “DECIMATION”, &value);

if (status != XIA_SUCCESS)
{
      /* ERROR getting DECIMATION */
}

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

xiaSetParameter

Syntax:

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

Description:

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 and shoule be used with caution.

Parameters:

detChan

detChan or detChan set to set the DSP parameter on.

name

Name of DSP parameter to set.

value

Value to set the DSP parameter 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;

unsigned short newThresh = 0x1000;

/* Setup a valid system */
status = xiaSetParameter(0, “THRESHOLD”, newThresh);

if (status != XIA_SUCCESS)
{
      /* ERROR setting THRESHOLD */
}
```

xiaGetNumParams

Syntax:

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

Description:

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 to a variable to store the number of parameters in

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

Syntax:

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

Description:

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. The proper amount of memory must be allocated for the string array passed in. The standard size is the number of parameters (retrieved using xiaGetNumParams()) by MAXSYMBOL_LEN, which is defined in handel_generic.h.
values	unsigned short *	The values of all of the DSP parameters for the specified detChan. The proper amount of memory must be allocated for the array passed in. Typically, the number of parameters is retrieved from xiaGetNumParams() and then used to allocate an array of the proper length.
access	unsigned short *	The access information for all of the DSP parameters for the specified detChan. The proper amount of memory must be allocated for the array passed in. Typically, the number of parameters is retrieved from xiaGetNumParams() and then used to allocate an array of the proper length. The access values are interpreted as follows: 0 = Read/Write, 1 = Read Only and 2 = WriteOnly.
lower_bounds	unsigned short *	The lower bounds information for all of the DSP parameters for the specified detChan. The proper amount of memory must be allocated for the array passed in. Typically, the number of parameters is retrieved from xiaGetNumParams() and then used to allocate an array of the proper length. If both the lower bounds and upper bounds information for a DSP parameter are equal to 0, then that DSP parameter doesn't have any defined bounds.
upper_bounds	unsigned short *	The upper bounds information for all of the DSP parameters for the specified detChan. The proper amount of memory must be allocated for the array passed in. Typically, the number of parameters is retrieved from xiaGetNumParams() and then used to allocate an array of the proper length. If both the lower bounds and upper bounds information for a DSP parameter are equal to 0, then that DSP parameter doesn't have any defined bounds.

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 into a void *. See the Usage section for examples of using void pointers in this context.

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;
```

xiaGetParamName

Syntax:

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

Description:

Returns the DSP parameter name located at the specified index. This routine should be used in place of the xiaGetParams() routine when interfacing to Handel with a language that doesn't support the passing of string arrays to DLLs, like Visual Basic. The idiom generally associated with this routine 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. See the Usage section for an example of how to do this.

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, typically 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);
}
```

xiaEnableLogOutput

Syntax:

```
int xiaEnableLogOutput(void)
```

Description:

Enables logging information to be output to the output stream set by a call to xiaSetLogOutput(). By default, logging is enabled and is directed to standard out. Note: 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 is really a redundant call since logging is
 * enabled by default.
 */
status = xiaEnableLogOutput();

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

xiaSuppressLogOutput

Syntax:
```
int xiaSuppressLogOutput(void)
```

Description:
Stops log output from being written to the current stream. Note: 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

Syntax:
```
int xiaSetLogLevel(int level)
```

Description:
Sets the level of logging that should be reported to log 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. Should only set this level if instructed to by XIA since it generates a **lot** of output, much of which is repetitive and may obscure other more important messages.

MD_INFO: All messages except for debug level messages. Currently, this level is almost equivalent to MD_WARNING as not all of the info messages have been added to the library yet.

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 buy may still produce incorrect results.

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.

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

Parameters:
level
Level of logging desired. See discussion in Description section for 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

Syntax:

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

Description:

Re-direct the logging output to the specified stream. The output stream may be set to one of three choices: an actual file, stdout or stderr. Handel interprets stdout and stderr to mean the console display and interprets any other name to be a file.

Parameters:

filename

Name of file to re-direct logging messages to or stdout or stderr

Usage:

```
int status;

status = xiaInitHandel();

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

xiaSetLogOutput(“my_log.txt”);
```

5 Files

Handel is really a framework of several libraries that all interact to create the interface that Handel provides. Figure 1 illustrates how all of the separate pieces fit together.

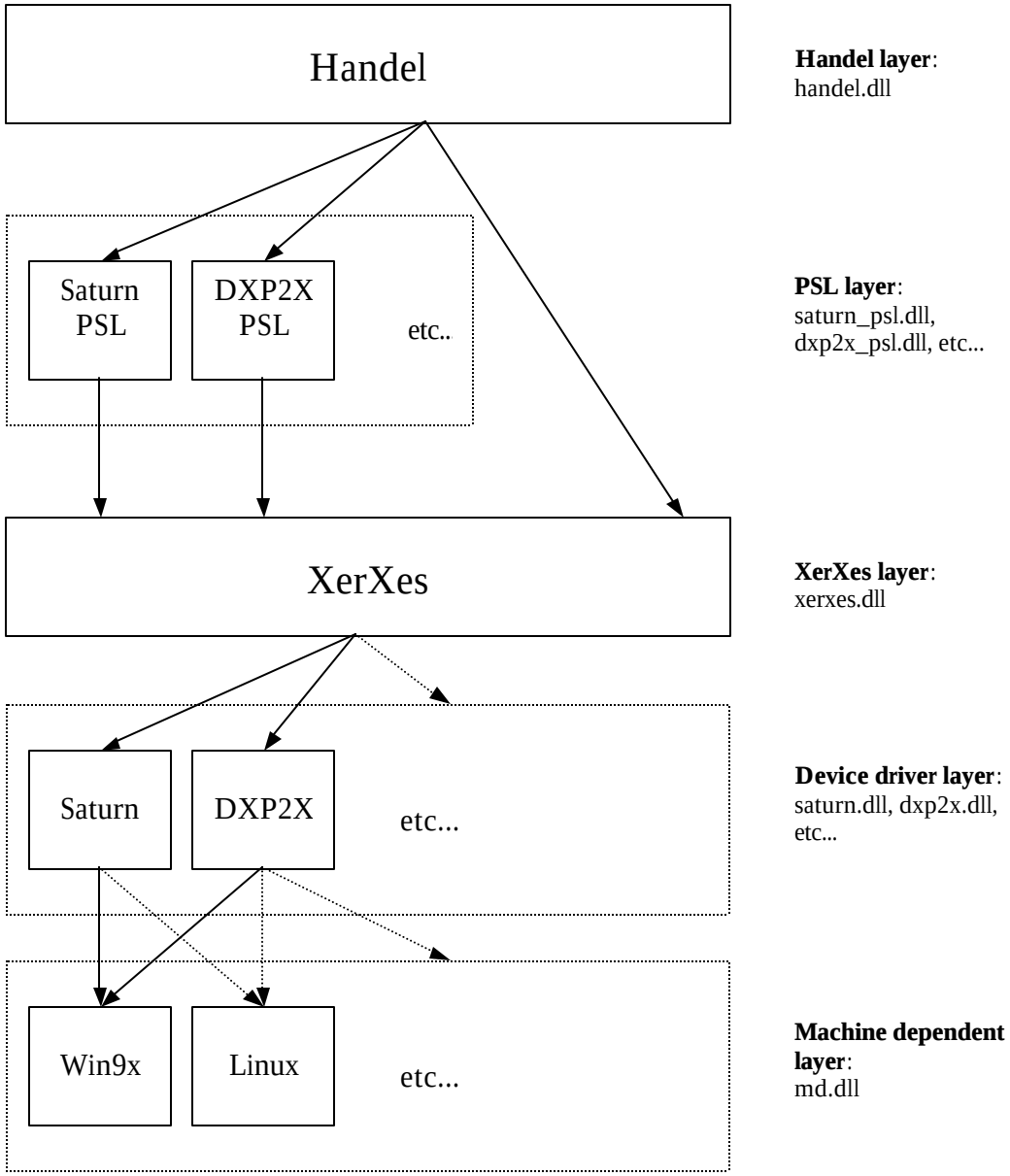

Figure 1

Below is a list of the Handel source files:

Handel

Source
handel.c
handel_dbg.c
handel_detchan.c
handel_dyn_default.c
handel_dyn_detector.c
handel_dyn_firmware.c
handel_dyn_module.c
handel_file.c
handel_log.c
handel_memdbg_win32.c
handel_run_control.c
handel_run_params.c
handel_sort.c
handel_system.c
handel_xerxes.c
saturn_psl.c

Includes
handel.h
handel_errors.h
handel_generic.h
handel_xerxes.h
handeldef.h
xia_file.h
xia_handel.h
xia_handel_structures.h
xia_module.h
xia_psl.h
xia_sort.h
xia_system.h
xia_system.h

The other libraries in the framework are explained in the Xerxes API.

Appendix A: Acquisition Values by Product

This appendix will attempt to provide a comprehensive list of all of the acquisition values by product. All of these values are stored as type double by Handel, however they will be cast into the proper type when appropriate.

X-ray Products (Saturn, Mercury, DXP-4C, DXP-2X)

Name	Product	Description
peaking_time	**All**	Peaking time in µs. Peaking time is roughly equal to twice the shaping time, which is commonly used in analog systems.
trigger_threshold	**All**	Value of the trigger threshold in eV. The trigger threshold is sometimes referred to as the "threshold".
mca_bin_width	**All**	Width of a single MCA bin in eV.
adc_percent_rule	**All**	Percent of the ADC corresponding to a calibration energy x-ray step.
calibration_energy	**All**	Expected energy of the calibration peak in eV.
number_mca_channels	**Saturn, Mercury, DXP-2X**	Total number of MCA ch annels.
mca_low_limit	**Saturn, Mercury, DXP-2X**	Energy cut-off for the lowest MCA bin in eV
energy_threshold	**Saturn, Mercury, DXP-2X**	Value of the energy threshold in eV. The energy threshold is sometimes referred to as the "slow threshold".
gap_time	**All**	The length of the slow filter gap (at decimation 0) in µs. SLOWGAP will be determined by calculating the minimum of $\left(\frac{gap\_time}{clock\_speed \cdot 2^{DECIMATION}}, \quad 3 \right)$
trigger_peaking_time	**All**	The peaking time of the trigger filter in µs. This value will be used to calculate FASTLEN.
trigger_gap_time	**All**	The gap time of the trigger filter in µs. FASTGAP will be determined by the following equation: $FASTGAP = \frac{trigger\_gap\_time}{clock\_speed}$
preset_runtime	**Saturn, Mercury, DXP-2X**	If this value is present then Handel will acquire data until the specified amount of runtime has passed. Runtime is expressed in seconds. (Optional)
preset_livetime	**Saturn, Mercury, DXP-2X**	If this value is present then Handel will acquire data until the specified amount of livetime has passed. Livetime is expressed in seconds. (Optional)
preset_output	**Saturn, Mercury, DXP-2X**	If this value is present then Handel will take data until the specified amount of output counts has been acquired. Output counts are expressed in counts and may not be larger then $2^{32} - 1$. (Optional)
preset_input	**Saturn, Mercury, DXP-2X**	If this value is present then Handel will take data until the specified amount of input counts has been acquired. Input counts are expressed in counts and may not be larger then $2^{32} - 1$. (Optional)
preset_standard	**Saturn, Mercury, DXP-2X**	If this value is present then Handel will take data until the user calls xiaStopRun().

peakint_offset_ptrr{n}	**All**	The constant from the PEAKINT recipe. (See Appendix C for more details.) If an FDD file is being used then the "_ptrr{n}" part of the name can be omitted. (Optional)
peaksam_offer_ptrr{n}	**All**	The constant from the PEAKSAM recipe. (See Appendix C for more details.) If an FDD file is being used then the "_ptrr{n}" part of the name can be omitted. (Optional)

Appendix B: .INI File Format

Handel uses the standard .ini file format of bracketed section headings ([Section]) followed by name-value pairs that define information for that section. Handel also supplements this format by allowing multiple aliases to be specifed under a single section heading. The only caveat is that 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.

Each section heading has a different set of allowed name-value pairs, which are similar to the allowed name-value pairs for the corresponding dynamic configuration routine.

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
```

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

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

etc...

Appendix C: Filter Parameters By Product

Each product potentially interprets the filter information stored in the Firmware structure differently. Handel, since it is product agnostic, only knows that it needs to store some filter data, but it makes no assumption about what the data means. The burden is on the individual user/programmer to verify that the information is entered correctly.

The index is the order in which it should be added as an item to a Firmware alias using xiaAddFirmwareItem().

X-ray Products (Saturn, Mercury, DXP-2X, DXP-4C)

Name	Index	Product	Description
peakint_offset	0	All	The constant in the PEAKINT recipe: PEAKINT = SLOWLEN + SLOWGAP + peakint_offset
peaksam_offset	1	All	The constant in the PEAKSAM recipe: PEAKSAM = PEAKINT – peaksam_offset

Appendix D: Run Data Types by Product

Each product has a different selection of data that may be read from its channels. In some cases, there are also special types associated with various firmware configurations.

X-ray Products (Saturn, Mercury, DXP-2X, DXP-4C)

Name	Type	Product	Firmware	Description
mca_length	unsigned long	All	Standard	The length of the MCA spectrum in unsigned long words.
mca	unsigned long *	All	Standard	An array containing the MCA spectrum. The user is expected to pass in an array of size "mca_length" to Handel.
livetime	double	All	Standard	The total time that the processor was able to acquire new events in seconds.
runtime	double	Saturn, Mercury, DXP-2X	Standard	The total time that the processor was taking data in seconds. Some XIA manuals refer to this as the "realtime".
input_count_rate	double	All	Standard	This is the total number of triggers divided by the runtime in counts per second.
output_count_rate	double	All	Standard	This is the total number of events in the run divided by the runtime in counts per second.
events_in_run	unsigned long	All	Standard	The total number of events in a run that are written into the MCA spectrum.
triggers	unsigned long	All	Standard	The total number of triggers obtained in a run. This quantity includes pile-up inspection.
baseline_length	unsigned long	All	Standard	The length of the baseline histogram.
baseline	unsigned long *	All	Standard	An array containing the baseline histogram. The user is expected to pass in an array of size "baseline_length" to Handel.
run_active	unsigned long	All	Standard	Returns run active status of a channel. The following constants, defined in handel.h, are returned: XIA_RUN_HARDWARE – The hardware thinks that a run is active. XIA_RUN_HANDEL – Handel thinks that a run is active. XIA_RUN_CT – Handel thinks that a control task run is active.

Appendix E: Special Run Types by Product

Each product has a different set of special runs associated with it. Additionally, each special run has a different set of parameters that can be passed into it via. the info array. However, the first element of the info array is the same for all of the special runs and should be set to the number of times the special run will execute. (For most special runs, this should be set to 1.) The Read Data column indicates if the appropriate xiaGetSpecialRunData() routine must be called in order to stop execution of the special run.

X-ray Products (Saturn, Mercury, DXP-2X, DXP-4C)

Name	Product	Read Data?	Type	Info	Description
adc_trace	All	Yes	double	info[1]: Amount of time to wait between ADC samples in nanoseconds. (Ignored for the DXP-4C.)	Acquire an ADC trace. An ADC trace is the digitzed output of the preamplifier (after being processed by the DXP's Analog Signal Conditioner). This is also known as the Digital Oscilloscope mode.
baseline_history	Saturn, Mercury, DXP-2X	Yes	int	N/A	Disable updating the baseline history buffer in preparation for reading it out. (If the history buffer was allowed to keep updating we wouldn't be able to get a "snapshot" of the history buffer at the time of the function call.)
open_input_relay	Saturn, Mercury, DXP-2X	No	int	N/A	This will open the input relay to the signal coming from the detector.
close_input_relay	Saturn, Mercury, DXP-2X	No	int	N/A	This will close the input relay and isolate the processor from the external signal from the detector.

Below is the table of special run data that can be read:
X-ray Products (Saturn, Mercury, DXP-2X, DXP-4C)

Name	Type	Product	Description
adc_trace_length	unsigned long	All	The length of the ADC trace to be read from the processor in unsigned long words.
adc_trace	unsigned long *	All	An array containing the ADC trace data. The user is expected to pass in an array of size "adc_trace_info" to Handel.
baseline_history_length	unsigned long	Saturn, Mercury, DXP-2X	The length of the baseline history buffer to be read from the processor in unsigned long words.
baseline_history	unsigned long *	Saturn, Mercury, DXP-2X	An array containing the baseline history data. The user is expected to pass in an array of size "baseline_history_length" to Handel.