

XerXes: XIA Host Library Specification

Revision 1.01

X-Ray Instrumentation Associates

8450 Central Ave.
Newark, CA 94560 USA

Tel: (510)494-9020; Fax: (510)494-9040
<http://www.xia.com/>

For additional information Contact
software_support@xia.com

Information furnished by X-ray Instrumentation Associates (XIA) is believed to be accurate and reliable. However, XIA assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of XIA. XIA reserves the right to change specifications at any time without notice. Patents have been applied for to cover various aspects of the design of the DXP Digital X-ray Processor.

Table of Contents

1	Introduction	1
2	General Terms	1
3	Calling Conventions.....	1
4	Common Arguments	2
4.1	General Detector Arguments	2
4.2	DSP Related Arguments	2
4.3	I/O Arguments	2
5	Applications Programming Interface (API).....	2
5.1	How XerXes finds files	2
5.2	File format conventions.....	2
5.3	Supported XIA products.....	2
5.4	Configuration using files	2
5.4.1	XIA_CONFIG and XIASYSTEMS.CFG.....	2
5.4.2	Modules Configuration File	2
5.4.2.1	param and default_param.....	4
5.4.2.2	fippi and default_fippi.....	4
5.4.2.3	dsp and default_dsp.....	5
5.4.3	preamp configuration in the XIA_CONFIG file	5
5.5	Configuration using routines	6
5.6	Return Values	6
5.7	Routines to Initialize the System	8
5.7.1	int dxp_init_library(void)	8
5.7.2	Routines called by dxp_init_library()	8
5.7.2.1	int dxp_install_utils(const char *utilname)	8
5.7.2.2	int dxp_init_ds(void)	8
5.7.2.3	int dxp_init_boards_ds(void).....	8
5.7.3	int dxp_initialize(void).....	8
5.7.4	Routines called by dxp_initialize()	8
5.7.4.1	int dxp_read_config(char *filename)	9
5.7.4.2	int dxp_assign_channel(void)	9
5.7.4.3	int dxp_user_setup(void)	9
5.7.4.4	int dxp_initialize_ASC(float *adcRule, float *energy).....	9
5.7.5	int dxp_add_system_item(char *token, char **values)	9
5.7.6	int dxp_add_board_item(char *token, char **values) 9	9
5.7.7	int dxp_reset_channel(int *detChan)	11
5.7.8	int dxp_locate_system_files(unsigned int *maxlen, char files[][3])	11
5.7.9	int dxp_locate_channel_files(int *detChan, unsigned int *maxlen, char files[][3])	11
5.7.10	void dxp_version(void).....	11
5.8	Routines to Download the DSP	11
5.8.1	int dxp_dspconfig(void).....	11
5.8.2	int dxp_replace_dspconfig(int *detChan,	

	char *filename).....	11
5.9	Routines to Download the FiPPi.....	12
5.9.1	int dxp_fipconfig()	12
5.9.2	int dxp_replace_fipconfig(int *detChan, char *filename).....	12
5.9.3	int dxp_check_decimation(unsigned short *decimation, unsigned short *different).....	12
5.10	Routines to Control the Operation of the DSP.....	12
5.10.1	Routines to Load and Update the DSP Parameter Values	12
5.10.1.1	int dxp_dspdefaults(int *detChan)	12
5.10.1.2	int dxp_replace_dsparams(int *detChan).....	12
5.10.1.3	int dxp_put_dsparams(void).....	12
5.10.1.4	int dxp_symbolname_list(int *detChan, char **list).....	12
5.10.1.5	int dxp_symbolname_limits(int *detChan, unsigned short *access, unsigned short *lbound, unsigned short *ubound)	12
5.10.1.6	int dxp_max_symbols(int *detChan, unsigned short *nsymbols)	13
5.10.1.7	int dxp_set_dspsymbol(char *name, unsigned short *value)	13
5.10.1.8	int dxp_set_one_dspsymbol(int *detChan, char *name, unsigned short *value)	13
5.10.1.9	int dxp_download_params(int *nparam, unsigned short addr[], unsigned short value[])	13
5.10.1.10	int dxp_download_one_params(int *detChan, int *nparam, unsigned short addr[], unsigned short value[])	13
5.10.1.11	int dxp_upload_dsparams(int *detChan)	13
5.10.1.12	int dxp_upload_channel(int *detChan, unsigned short *nparam, unsigned short *addr, unsigned long value[]).....	13
5.10.1.13	int dxp_get_one_dspsymbol(int *detChan, char *name, unsigned short *value)	13
5.10.1.14	int dxp_get_dspsymbol(int *detChan, char *name, unsigned long *value)	13
5.10.1.15	int dxp_get_symbol_index(int *detChan, char *name, unsigned short *symindex).....	13
5.10.1.16	int dxp_symbolname_by_index (int *detChan, unsigned short *symindex, char *name)	13
5.10.1.17	int dxp_mem_dump(int *detChan).....	14
5.10.2	Routines to Change and Observe Other DSP Memory	14
5.10.2.1	int dxp_nspec(int *detChan, unsigned int *len_spec)	14
5.10.2.2	int dxp_nbase(int *detChan, unsigned int *len_base)	14
5.10.2.3	int dxp_nevent(int *detChan, unsigned int *len_event)	14
5.10.2.4	int dxp_readout_detector_run(int *detChan, unsigned short *params, unsigned short *baseline, unsigned long *spectrum)	14
5.11	Routines that Provide Hardware Setup Information	14
5.11.1	int dxp_ndxp(int *numDxp).....	14
5.11.2	int dxp_ndxpchan(int *numDxpChan).....	14
5.11.3	int dxp_get_detectors(int *numDxpChan,	

	int detchan[]).....	14
5.11.4	int dxp_get_board_type(int *detChan, char *name)...	14
5.12	Routines to Control Runs.....	14
5.12.1	int dxp_start_run(unsigned short *gate, unsigned short*resume)	14
5.12.2	int dxp_stop_run(void)	14
5.12.3	int dxp_resume_run(void).....	15
5.12.4	int dxp_start_one_run(int *detChan, unsigned short *gate, unsigned short *resume)	15
5.12.5	int dxp_stop_one_run(int *detChan)	15
5.12.6	int dxp_resume_one_run(int *detChan).....	15
5.12.7	int dxp_isrunning(int *detChan, int *active).....	15
5.12.8	int dxp_isrunning_any(int *detChan, int *active)	15
5.12.9	int dxp_enable_LAM (int *detChan).....	15
5.12.10	int dxp_disable_LAM (int *detChan).....	15
5.12.11	int dxp_reset_LAM (int *detChan).....	15
5.13	Routines to Perform Special Runs	15
5.13.1	int dxp_calibrate(int *calibtask)	15
5.13.2	int dxp_calibrate_one_channel(int *detChan, int *calibtask)	16
5.13.3	int dxp_control_task_info(int *detChan, short *type, int *info).....	16
5.13.4	int dxp_start_control_task(int *detChan, short *type, unsigned int *length, int *info).....	16
5.13.5	int dxp_stop_control_task(int *detChan)	16
5.13.6	int dxp_get_control_task_info(int *detChan, short *type, long *data)	16
5.14	Routines to Obtain Information about a Run	16
5.14.1	int dxp_get_statistics(int *detChan, double *lifetime, double *icr, double *ocr, unsigned long *nevent)	16
5.15	Routines to Change the Gain of the XIA Hardware	16
5.15.1	int dxp_modify_gains(float *gainchange)	16
5.15.2	int dxp_modify_one_gains(int *detChan, float *gainchange)	16
5.16	Routines to Access Files and Save Data.....	17
5.16.1	int dxp_open_file(int *lun, char *filename, int *mode).....	17
5.16.2	int dxp_close_file(int *lun).....	17
5.16.3	int dxp_save_dspparams(int *detChan, int *lun).....	17
5.16.4	int dxp_write_spectra(int *luns, int *lunb).....	17
5.16.5	int dxp_save_config(int *lun)	17
5.16.6	int dxp_restore_config(int *lun).....	17
5.17	Routines to Control Error Logging.....	17
5.17.1	int dxp_enable_log()	17
5.17.2	int dxp_suppress_log().....	17

5.17.3	int dxp_set_log_level(int *level)	17
5.17.4	int dxp_set_log_output(char *filename)	18
5.18	Miscellaneous Utility Routines	18
5.18.1	int dxp_findpeak(long *array, int *nbins, float *thresh, int *lower, int *upper)	18
5.18.2	int dxp_fitgauss0(long array[], int *lower, int *upper, float *pos, float *fwhm)	18
5.18.3	int dxp_lock_resource (int *detChan, short *lock)	18
6	API for the Machine Dependent Library	19
6.1	Machine Dependent Structures	19
6.1.1	struct Xia_Io_Functions	19
6.1.2	struct Xia_Util_Functions	19
6.2	Machine Dependent Initialization	19
6.2.1	int dxp_md_init_util(Xia_Util_Functions funcs, char *type)	19
6.2.2	int dxp_md_init_io(Xia_Io_Functions funcs, char *type)	19
6.2.3	int dxp_md_initialize(int maxMod, char *name)	19
6.2.4	int dxp_md_open(char *name, int *ioChan)	19
6.3	Machine Dependent Memory Allocation	20
6.3.1	void *dxp_md_alloc(size_t length)	20
6.3.2	void dxp_md_free(void *array)	20
6.4	Machine Dependent User Interface	20
6.4.1	<i>***DEPRECATED***</i> void dxp_md_error_control(char *keyword, int *value) <i>***DEPRECATED***</i>	20
6.4.2	int dxp_md_output(char *filename)	20
6.4.3	void dxp_md_error(char *routine, char *message, int *error_code)	20
6.4.4	void dxp_md_warning(char *routine, char *message)	20
6.4.5	void dxp_md_info(char *routine, char *message)	20
6.4.6	void dxp_md_debug(char *routine, char *message) ..	20
6.4.7	int dxp_md_enable_log()	20
6.4.8	int dxp_md_suppress_log()	20
6.4.9	int dxp_md_set_log_level(int level)	20
6.4.10	int dxp_md_log(int level, char *routine, char *message, int error)	21
6.4.11	int dxp_md_puts(char *s)	21
6.5	Machine Dependent External I/O (CAMAC)	21
6.5.1	int dxp_md_io(int *ioChan, int *function, int *address, short *data, int *length)	21
6.5.2	int dxp_md_set_maxblk(int *blksiz)	21
6.5.3	int dxp_md_get_maxblk(void)	21
6.6	Miscellaneous Machine Dependent Routines	21

6.6.1	int dxp_md_wait(float *time).....	21
6.6.2	int dxp_md_lock_resource (int *ioChan, int *modChan, short *lock)	21
7	The host software release	22
7.1	Header Files	22
7.2	Source Files	22
7.3	Library Files	22
8	Control Task Types.....	24
8.1	ALL XIA Modules.....	24
8.2	DXP-4C	25
8.3	DXP-4C2X	26
8.4	DXP-X10P	28
8.5	DXP-G200	30

1 Introduction

This document is intended to aid the data collection programmer in developing software that controls and reads out data from DXP-4C and DXP-4C2X modules. A set of data handlers, which are written in C (currently only C callable¹) are supplied with the DXP which can be incorporated into existing data collection packages or used to develop a stand alone application.

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 common arguments used in many of the data handlers (also called primitive routines); § 5 describes the driver routines; and § 6 describes the machine dependent routines. The final section, § 7, describes all of the files included in the host software release.

¹ Routines to make the library Fortran callable have not been implemented as of 2/25/00. All routines involving character strings will require modification, please contact XIA if you require support for Fortran.

2 General Terms

- host:** This is the computer to which the CAMAC crate controller is connected to and on which the software described in this manual runs.
- CAMAC interface:** This is the method that each XIA module uses to communicate with the host computer. It satisfies both standard and fast CAMAC protocol, providing registers for controlling and transferring data to and from the individual DXP channels.
- DSP:** This is the on board digital signal processor, one per channel, 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 DSP will not function until its program has been downloaded to the modules from the host computer.
- DSP program memory:** The program memory is divided into two sections: the DSP program and the DSP spectrum memory. The DSP spectrum memory stores the results of a data taking run. The fundamental word size for program memory is 24 bits.
- DSP data memory:** The DSP memory is referred to by various names throughout the document: parameter memory, event memory and baseline histogram memory. The DSP parameter memory contains the variables used by the DSP program to perform all functions. The DSP event memory stores the output of the on-board ADC in programmable time steps, acting as a digital oscilloscope. The baseline histogram memory contains a histogram of all past baseline measurements. The baseline is a measure of the ADC output assuming no input signal. The fundamental word size in data memory is 16 bits. These particular elements of data memory may or may not exist depending on the XIA device under control, e.g. the gamma ray products do not typically provide the baseline histograms.
- File Pointers:** These are names that can be assigned to any file accessible to the host operating system. On VMS, file pointers are called logical names. On UNIX, they are called symbolic links or environment variables.
- FiPPI:** This is the field programmable gate array (FPGA) in which the **F**ilter, **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.
- read:** Transfer data from the DSP to the host computer. A single word read copies 16 bits of data from a specified location in DSP memory to the 16 least significant bits (LSBs) (this assumes that a short integer type is 32 bits wide) of a word on the host computer. The width of 16-bits is limited by the CAMAC protocol. When reading from program memory, 2 16-bit reads are performed and the data is properly packed by the library.
- DSP parameters:** This is a mapping of DSP parameter memory into symbolic names. This mapping should not be hard coded in the host software, since future releases of the DSP program may use different locations. The mapping is contained in the same file as the DSP program and is read in automatically by XerXes.
- write:** Transfer data from the host computer to the DSP. A single word write copies the 16 LSBs of data from a word on the host computer to a specified location in DSP memory. See the definition of a read for more information.

3 Calling Conventions

- Arguments are “Pass by Reference”** This means that all routines (except those that pass character strings) are FORTRAN or C callable. For FORTRAN programs, wrapper routines need to be written to allow passing of strings between your program and XerXes.
- FORTRAN callable routines that pass character strings:** Some routines (e.g. `dxp_md_open` and `dxp_loc`) are not callable from FORTRAN. FORTRAN programmers require a different set of routines, please contact XIA for more information.
- XIA hardware calls are integer functions.** If successful, all XerXes routines return `DXP_SUCCESS` (as defined in `xerxes_errors.h`), otherwise they return a status code indicating a problem (see `xerxes_errors.h` for error codes). In addition, all routines that sense an error print a message to the output stream. 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 interface to 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.

4 Common Arguments

This section describes some common arguments used to direct the XerXes routines in their actions.

4.1 General Detector Arguments

int *detChan: Detector channel number – The variable **detChan** points to a specific detector channel within the system of XIA modules. The physical detector channel associated with each DXP channel is discussed in the **module configuration file** section.

4.2 DSP Related Arguments

unsigned short *params: DSP parameters – The parameter memory of the DSP is passed using this array of variables. Typically the individual parameters are changed by specific routines and not by direct modification of this array. The number of parameters in the DSP memory is returned by **dxp_max_symbols()**.

unsigned short *baseline: Baseline Histogram – This array contains the baseline histogram filled by the DSP. The length of the baseline histogram is returned by **dxp_nbase()**.

unsigned long *spectrum: Spectrum Memory – This array contains the spectrum memory filled by the DSP. The length of spectrum memory is returned by **dxp_nspec()**.

4.3 I/O Arguments

int *lun: Logical Unit Number – This is a global reference to a file that was opened by the library. It is merely a pointer and should never be used to attempt access to a file directly. The routines **dxp_open_file()** and **dxp_close_file()** should always be used to initiate file access.

5 Applications Programming Interface (API)

The definitions for the XerXes routines are divided by functionality. All calling parameters are passed by reference to accommodate compatibility with Fortran routines. All routines return an integer that can be used to determine if the routine executed successfully.

5.1 How XerXes finds files

When XerXes searches for a file, it starts with an attempt to directly open the file, looking for the file in the current working directory. If the file is not found, then the library will search for the file in the directory pointed to by the environment variable, **XIAHOME**. For backward compatibility with some setups, we also search for the directory pointed to by the environment variable, **DXPHOME**. In addition, XerXes will try to use the file as a pointer to another environment variable, trying to find the indirect file name both directly and with the two environment variables, **XIAHOME** and **DXPHOME**.

5.2 File format conventions

For all files read or written by XerXes, any line starting with an asterisk, *, are considered comments. For some file formats, inline comments are allowed, their form specified for each file format, usually starting with an exclamation point, !.

5.3 Supported XIA products

The following products are currently supported by XerXes. When specifying the product type use these names so that XerXes will recognize the product type.

- **dxp4c** – a DXP-4C XIA Digital X-Ray Processor (DXP) board running at 20Mhz (any number of stuffed channels is supported, i.e. supports the DXP-2C)
- **dxp4c2x** – a DXP-4C2X XIA Digital X-Ray Processor (DXP) board running at 40Mhz (also supports the DXP-2C2X)

- **dpx10p** – a DXP-X10P single channel XIA Digital X-Ray processor running at 40MHz via a parallel port communications protocol.

There are two methods to configure the XerXes library: configuration files and individual configuration routines. In order to use the configuration files, the user first creates a couple of files that contain the system information, then a call to **dxp_initialize()** will read the files and configure the system for operation. The other method involves specifying the same information with repeated calls to **dxp_add_system_item()** and **dxp_add_board_item()**. In either case, the same information must be specified, but the **dxp_add*** routines allow greater flexibility for host programmers. In either case, a later call to **dxp_restore_config()** will wipe the slate clean and configure a previously saved setup.

5.4 Configuration using files

5.4.1 XIA_CONFIG and XIASYSTEMS.CFG

File oriented configuration requires an additional environment variable, **XIA_CONFIG**, which points to the system configuration file. All high-level configuration information is contained in **XIA_CONFIG**. An example file is provided below:

```
*
* This is the system configuration file. The file should contain
* an entry for a preamp configuration file (may be NULL for no
* existing configuration => no default gain settings for hardware).
* A modules entry defines what set of XIA modules the software
* is controlling. A NULL definition for the modules entry implies
* that no modules are in the system. This may be desirable for
* a system that will always use saved configurations.
* A board type is specified followed by a library definition
* Please use lower case for all tag names...to ease confusion later.
* File names, libraries and environment variables may be lower or
* upper case.
*
*
* Sample Configuration
*
preamp      c:\wahl\XIA Software Support\Configuration Files\preamp_ortec.cfg
*preamp     NULL
modules     c:\wahl\XIA Software Support\Configuration Files\dxp.module
*modules    NULL
dxp4c       Unused
dxp4c2x     Unused
```

The preamp and modules configuration files are described later in this document. If either of these tags is defined as NULL, then the initializations involving them are skipped. For example, not defining the preamp specifications will result in the hardware using default gain settings. Each type of board in the system (e.g. DXP-4C, DXP-4C2X, etc...) must have an entry in this file. These are not physical board entries for the system, they merely tell XerXes what type of boards to expect. The types of boards available are described in **Section 5.3**. The "Unused" values associated with the board type entry may be used in the future. Later, when a system is defined, each module in the system must have a board type matching an entry in **XIA_CONFIG**.

5.4.2 Modules Configuration File

The **modules** configuration file specifies all necessary information, both in terms of firmware and physical location, to allow XerXes to communicate with the device. Any line starting with a * represents a comment. The following entries are valid for the **modules** configuration file:

- **board_type**: defines the type of XIA device. The value of this entry becomes the default device type for all future module entries until another **board_type** entry occurs. The specified type must have a corresponding entry in the **XIA_CONFIG** file.

- **interface, iolibrary:** Together, these define the type of communications protocol to use for controlling the device. Both these tags are currently passed to the machine dependent layer and handled there. When using the default machine dependent code distributed with XerXes, the **iolibrary** entry, under windows 9x, specifies the DLL used to communicate with CAMAC or the parallel port base address for EPP protocols. Under linux the **interface** entry is currently unused. The **interface** entry tells the machined dependent layer what communication protocol to employ; CAMAC and EPP are currently supported. Specifically, the tag associated with **iolibrary** is passed to the user defined routine **dxp_md_initialize()** and the **interface** tag is passed to **dxp_md_init_util()** and **dxp_md_init_io()**. In the same fashion as the **board_type** identifier, the values of **iolibrary** and **interface** are persistent for all **module** definitions, until new **iolibrary** and **interface** entries are encountered.
- **module:** The module entry specifies the physical location of a device as well as user defined detector channel numbers that are used to perform operations using XerXes routines. The string (looks like a number in the example below) following the **module** entry contains the information needed by **dxp_md_open()** to establish connection with the device. For the XIA supplied machine dependent code, **0104** specifies that the module is on SCSI bus **0**, crate **1** and in slot **04** of a CAMAC crate. The series of numbers (in this case 4 numbers) on the rest of the **module** entry are user defined detector numbers, referred to as **detChan** numbers from **Section 4.1**. Each entry represents a channel on the XIA device and if the entry is negative, the channel is assumed to be non-functional or non-existent. Any positive value represents a working channel. In this case, a **dxp4c2x** device has 4 channels, which we reference by the numbers **0, 1, 2** and **3**. In all future calls by the user, the detector channels are manipulated using these **detChan** values.
- **dsp, default_dsp:** Specifies a filename for DSP firmware that XerXes will use to configure the previously defined **module** entry. A **default_dsp** entry tells XerXes to apply this filename to all channels of the **module**. Individual channels within a module can have different DSP firmware and are specified by the **dsp** entry, followed by the channel number (e.g. for a 4 channel device, channel numbers are 0-3), then ending with the filename for this specific channel. A **dsp** filename must be specified for each device channel.
- **fippi, default_fippi:** Same as **dsp** except applying to the fippi firmware for the XIA device. A **fippi** filename must be specified for each device channel.
- **param, default_param:** Same as **dsp** except applying to default parameter values for the DSP. These parameters can be used to restore the system to default values. These entries may contain NULL instead of a valid filename.

An example **modules** configuration file follows which defines a single 4 channel DXP-4C2X in slot 4 of a CAMAC crate.

```
*
* Configuration file for modules in the system
* "board_type" type           - this type must match a definition in XIA_CONFIG
* "module" SCSI_Crate_slot channel0 channel1 channel2 channel3
* "default_dsp" filename
* "dsp" channel filename
* "default_fippi" filename
* "fippi" channel filename
* "default_param" filename (NULL)
* "param" channel filename (NULL)
*
* All dsp, fippi and param definitions must follow the module definition
* that they apply to.
*
* the board_type must be defined prior to defining the module information
* it will apply to all future module definitions until the tag is reused.
*
* the iolibrary name must be defined before the interface library is defined
```

** if a new interface is defined, the iolibrary must be named before in all cases.*

```

*
board_type      dxp4c2x
iolibrarycamacdll.dll
interfaceCAMAC
module          0104 0 1 2 3
default_fippi   c:\wahl\dxp 2x\firmware\fxpd0g_v5.fip
fippi           0 c:\wahl\dxp 2x\firmware\fxpd2g_v5.fip
default_dsp     c:\wahl\dxp 2x\dsp\x10p.hex
dsp             3 c:\wahl\dxp 2x\dsp\x10p.hex
default_param   c:\wahl\XIA Software Support\Configuration Files\DXP2x.cfg
param          2 c:\wahl\XIA Software Support\Configuration Files\DXP2x_other.cfg
param          3 NULL

```

5.4.2.1 param and default_param

These symbols point to the file(s) that overrides some or all of the initial parameter values for the DSP. Comments in the file start with a *. All other lines contain a symbol name (in ASCII text) and value pair, delimited by whitespace. Any text after the symbol, value pair is treated as comments as well.

A **param/default_param** filename of **NULL** can be specified. In this case, the default parameters of the DSP are left unchanged.

```

*
* This file lists some default parameter values to be downloaded to the DSP.
*
* Comment lines are indicated with a '*' character at start of line
*
* Non-comment line format: parameter name <space> value [<space> comments]
*
POLARITY 0 ! 0 = negative, 1 = positive
RUNTASKS 122
*
* Here are the FiPPI parameters
*
SLOWLEN 20 ! 1 usec peaking time example
SLOWGAP 6
PEAKINT 26
FASTLEN 4
FASTGAP 0
THRESHOLD 69
MINWIDTH 4
MAXWIDTH 20
PEAKSAMP 21
DECIMATION 0

```

5.4.2.2 fippi and default_fippi

This symbol points to the file that contains the program for the FiPPI (FPGA) on the XIA module. This file must be in ASCII format (typically with the .fip file extension), since this library does not support binary versions of the programming file. It is up to the user to specify the FiPPI configuration file for different decimations. Lines beginning with a * are ignored.

```

* Copyright 1996, X-ray Instrumentation Associates
* All Rights Reserved
* Fippi COMPRESS Version 1.0

```

```

* Xilinx LCA f01c8e0d.lca 4008EPQ160
* File f01c8e0d.rbt
* Fri Mar 21 16:19:00 1997
* Fri Mar 21 16:19:00 1997
* Source
* Version
* Produced by makebits version 5.2.1
FF0424A0F9ECBF7C7DCBBDF7DEF2F5B7B7DCEDDBAB3F996492C93D9B64FE9BFF6FBFBFF6FEFBA
FFD
FEFF6B6FFBFF7DF76DB6DF7DF76CBFFE6FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFEFBFFF
FEFFEF
    
```

5.4.2.3 dsp and default_dsp

This symbol points to the file containing the DSP program to download to the XIA module. This file must contain a compiled DSP program in ASCII format. Lines that begin with * are ignored. The file is divided into two sections. The first section contains the number of symbols followed by the symbol names, delimited by carriage returns. The second section contains the compiled program code in hexadecimal format. XerXes uses bounds information from the symbol definition to restrict user access to DSP memory. For most applications, this file will be generated and distributed unchanged by XIA.

```

*****
* ADSP2181 software for XIA pulse processor
* created Fri Feb 25 12:50:01 2000
*
* PRODUCT=DXP-4C2X
* Hardware variant 0
* Program variant 0
* Program revision 1.00
*
* Following lines are # global symbols followed by symbol table.
* Access codes: -:R/O, *:R/W, with optional bounds
*****
100
PROGNUM      -
CODEREV      -

Etc...for 96 more DSP symbols

USER8        *
TESTUPDN     *
4000009000701871bf000000a001f00000000000000000a001f0000000000000000
1c030f0a001f000000000000a001f0000000000000000a001f0000000000000000
    
```

5.4.3 preamp configuration in the XIA_CONFIG file

This symbol points to a file that configures the analog gains for the XIA module. Lines beginning with a * are ignored. There can be a header line with the key ENERGY that designates the energy of a typical incident event (specified in KeV). Each subsequent line contains the configuration for a single detector channel represented by 6 numbers. The first number is the detector channel number (specified in the **modules configuration file**, also called **detChan**). The second number is the front-end gain factor that is used to fine tune the gain for different detector channels. This gain factor is an overall scale factor for that detector channel. The third number is the polarity (0=negative, 1=positive) of the pulse from the pre-amplifier when an event occurs. The fourth and fifth numbers are the minimum and maximum voltage (in Volts) from the pre-amplifier (i.e. the reset range of the pre-amplifier, not applicable

to RC feedback pre-amplifiers). The last number is the typical step size of the pre-amplifier signal due to an incident X-ray (in milliVolts). A sample table follows, each element of the table must be comma delimited.

A filename of NULL can be specified for this configuration file. In this case, the library does not attempt to set the analog gains for the XIA module. Currently, if the file is specified, ALL detector channels must be specified.

```

*
*   This is the XIA PREAMP configuration file.
*
ENERGY 5.9                               * 5.9KeV X-Rays
1,      1.0,  0,      -2.0,  +2.0,  5      * Detector channel 1 has a modified
* gain of 1.0 (unmodified), is negative polarity, has a voltage range
* of -2.0V to +2.0V and an incident X-Ray of 5.9 KeV steps the voltage
* by 5 mV.
2,      1.0,  1,      -1.0,  +1.0,  2      * Detector channel 2 has a modified
* gain of 1.0 (unmodified), is positive polarity, has a voltage range
* of -1.0V to +1.0V and an incident X-Ray of 5.9 KeV steps the voltage
* by 2 mV.

```

5.5 Configuration using routines

The library is organized such that everything that is defined by the configuration files can be defined using initialization routines. A combination of configuration files and user callable configurations can be performed, but is not recommended unless you have a deep understanding of the library. To create a working configuration using only routines the following steps should be taken.

1. XerXes must first be initialized by a call to **dxp_init_library()**, which will create pointers to the machine dependent routines and initialize the internal data structures to an empty state.
2. The board type(s) being used must be defined with a (several) calls to **dxp_add_system_item()** using a token entry of **dxp4c**, **dxp4c2x** or **dxpx10p** (see Section 5.3 for type definitions). A full discussion of **dxp_add_system_item()** is given later in the document.
3. Now calls can be made to **dxp_add_board_item()** to define an individual board in the system. The complete list of tokens supported by this routine is discussed later. The tokens are very similar to the entries seen in the **modules** configuration file.

This method allows more flexibility in the specification, but clearly requires more knowledge of the library and devices in the system.

5.6 Return Values

Most routines return a value defined in the following list. These values are all defined as compiler directives (**#define**) in the header file **xerxes_errors.h**. Please refer to **xerxes_errors.h** for the most recent list of error codes.

```

/*
* some error codes
*/
#define DXP_SUCCESS      0
/* I/O level error codes 1-100*/
#define DXP_MDOPEN      1
#define DXP_MDIO        2
/* primitive level error codes (due to mdio failures) 101-200*/
#define DXP_WRITE_TSAR  101
#define DXP_WRITE_CSR   102
#define DXP_WRITE_WORD  103
#define DXP_READ_WORD   104
#define DXP_WRITE_BLOCK 105

```

```

#define DXP_READ_BLOCK      106
#define DXP_DISABLE_LAM    107
#define DXP_CLEAR_LAM      108
#define DXP_TEST_LAM       109
#define DXP_READ_CSR       110
#define DXP_WRITE_FIPPI    111
#define DXP_WRITE_DSP      112
#define DXP_WRITE_DATA     113
#define DXP_READ_DATA      114
#define DXP_ENABLE_LAM     115
#define DXP_READ_GSR       116
#define DXP_WRITE_GCR      117
/* DSP/FIPPI level error codes 201-300 */
#define DXP_MEMERROR        201
#define DXP_DSPRUNERROR    202
#define DXP_FIPDOWNLOAD    203
#define DXP_DSPDOWNLOAD    204
#define DXP_INTERNAL_GAIN  205
#define DXP_RESET_WARNING  206
#define DXP_DETECTOR_GAIN  207
#define DXP_NOSYMBOL       208
#define DXP_SPECTRUM       209
#define DXP_DSPLOAD        210
#define DXP_DSPPARAMS      211
#define DXP_DSPACCESS      212
#define DXP_DSPPARAMBOUNDS 213
/* configuration errors 301-400 */
#define DXP_BAD_PARAM       301
#define DXP_NODECIMATION   302
#define DXP_OPEN_FILE      303
#define DXP_NORUNTASKS     304
#define DXP_OUTPUT_UNDEFINED 305
#define DXP_INPUT_UNDEFINED 306
#define DXP_ARRAY_TOO_SMALL 307
#define DXP_NOCHANNELS     308
#define DXP_NODETCHAN      309
#define DXP_NOIOCHAN       310
#define DXP_NOMODCHAN      311
#define DXP_NEGBLOCKSIZE   312
#define DXP_FILENOTFOUND   313
#define DXP_NOFILETABLE    314
#define DXP_INITIALIZE     315
#define DXP_UNKNOWN_BTYPE  316
#define DXP_NOMATCH        317
#define DXP_BADCHANNEL     318
#define DXP_DSPTIMEOUT     319
/* host machine errors codes: 401-500 */
#define DXP_NOMEM          401
#define DXP_CLOSE_FILE     403
#define DXP_INDEXOOB       404
#define DXP_RUNACTIVE      405

```

```

/* Debug support */
#define DXP_DEBUG      1001

```

5.7 Routines to Initialize the System

5.7.1 `int dxp_init_library(void)`

This routine will initialize the library to an empty state and assign routines for handling machine dependent functions, such as memory allocation and error reporting. This is the minimum method to start using the XerXes library. It performs function calls in the following order:

1. `dxp_install_utils("default")`
2. `dxp_init_ds()`.

5.7.2 Routines called by `dxp_init_library()`

The following routines are all executed by a call to `dxp_init_library()`. However there are instances when these routines may be called independent of `dxp_init_library()`.

5.7.2.1 `int dxp_install_utils(const char *utilname)`

A call to this routine configures XerXes to use the machine dependent (**md**) routines described in Section 6. The variable `*utilname` is passed directly to the **md** routines with a call to `dxp_md_init_util()` which returns function pointers to all the **md** routines. In principle, `*utilname` can be used to specify different sets of utility routines for use with different interfaces, e.g. CAMAC or EPP. At any time during use of the library, the user can redefine the **md** routines in use by a call to this routine. Of course, the user would have to write a `dxp_md_init_util()` routine that allows dynamically changing the routine names.

5.7.2.2 `int dxp_init_ds(void)`

This routine will initialize all data structures within XerXes to an empty state and must be called before any calls to `dxp_add_system_item()` and `dxp_add_board_item()`. This is equivalent to having nothing defined in the system. The routine `dxp_init_boards_ds()` is called by this routine to clear all information about board definitions. The full system can be cleared at any time during running, but all calls to `dxp_add_system_item()` and `dxp_add_board_item()` must be redone.

5.7.2.3 `int dxp_init_boards_ds(void)`

This routine will wipe the data structures that contain information about boards in the system and their configuration. This routine is automatically called by `dxp_init_ds()` but can be called independent of `dxp_init_ds()` in order to initialize the board information without losing the system information.

5.7.3 `int dxp_initialize(void)`

If a program desires to use the configuration files method, where all information for the system and boards is contained in files, then this routine will perform all needed initializations. It loads the **XIA_CONFIG** and **module configuration file**. It calls the following user callable routines in this order:

1. `dxp_init_library()`
2. `dxp_read_config("XIA_CONFIG")`
3. `dxp_assign_channel()`
4. `dxp_user_setup()`.

Every time this routine is called, it will reset the configuration to the startup definitions based on the configuration files and all changes to configurations and parameters will be lost.

5.7.4 Routines called by `dxp_initialize()`

These routines are all called by `dxp_initialize()` but can be called independently. This is not recommended unless the user has a deep understanding of the library and the hardware.

5.7.4.1 int dxp_read_config(char *filename)

This routine will read in the system configuration file given by ***filename**. The file must have the format described in Section 5.4.1, the **XIA_CONFIG** file.

5.7.4.2 int dxp_assign_channel(void)

This routine reads in the **module** configuration file that was previously defined in the **XIA_CONFIG** file. Alternatively, the file can be defined using **dxp_add_system_item()**. When a new **interface** is read in from the file, the routine **dxp_md_initialize()** is called and initializes the **md** layer of code. The **module** configuration file associates a detector (e.g. logical channel number) to each module and channels within a module (e.g. electronic channel number). See the discussion in section 5.4.2.

5.7.4.3 int dxp_user_setup(void)

This routine configures (downloads firmware, DSP parameters) each DXP channel in the system according to the information read using **dxp_assign_channel()** or using calls to **dxp_add_board_item()**. The configuration consists four parts:

1. download the FiPPI firmware
2. download the DSP firmware
3. download the default DSP parameters
4. run calibrations appropriate to the board type via **dxp_initialize_ASC()**.

For the DXP-4C2X, calibration runs are automatically performed when the loading of the DSP program completes.

5.7.4.4 int dxp_initialize_ASC(float *adcRule, float *energy)

This routine reads in file pointed to by the **preamp** entry read from the **XIA_CONFIG** file or defined by **dxp_add_system_item()**. The **preamp** file contains channel dependent pre-amplifier parameters which the routine uses to adjust internal parameters that define the gain of the board based on these values. The ***adcRule** is specified as the fraction of the ADC full scale contributed by a single X-Ray pulse (typically set to 0.05). See section 5.4.3 for a more detailed discussion of the **preamp** configuration file. The routine returns the ***energy** that is defined in the **preamp** configuration file.

5.7.5 int dxp_add_system_item(char *token, char **values)

This routine allows the user to dynamically allocate system information. The variable ***token** specifies a system characteristic and the parameter(s) associated with that characteristic are contained in ****values**. The number of entries in the ****values** array depends on the characteristic specified. Below is a list of recognized **tokens** (listed in bold face) followed by a description of **values** associated with that token.

- Each supported XIA module from Section 5.3, e.g. **dxp4c**, **dxpx10p**, represents a valid token and has 1 valid **values** entry, which is currently unused but must be specified, e.g. **values[0]**="NULL".
- **preamp** has 1 valid **values** entry, which is a filename pointing to the preamp configuration file, see Section 5.4.3.
- **modules** has 1 valid **values** entry, which is a filename pointing to the modules configuration file, see Section 5.4.2.

5.7.6 int dxp_add_board_item(char *token, char **values)

This routine allows the user to dynamically allocate board information. The variable ***token** specifies a board characteristic and the parameter(s) associated with that characteristic are contained in ****values**. The number of entries in the ****values** array depends on the characteristic specified. Below is a list of recognized **tokens** (listed in bold face) followed by a description of **values** associated with that token. The calling order of characteristics is very important for **dxp_add_board_item()**. One must first specify a **board_type**, then proceed to define an **interface**, **fippi**, etc... Each characteristic here is given a level number in parentheses, which tells the order of importance. A characteristic of higher level is always applied to all lower level characteristics, e.g. if a **board_type(1)** is defined, then all subsequent calls using a **module(2)** characteristic will be of the type defined by **board_type** until another call of characteristic **board_type** is performed. For each level, all characteristics of the lower levels MUST be defined prior to calling a

characteristic of the current level, e.g. before defining a **module(2)** entry, both **board_type(1)** and **interface(1)** must have been defined. For a similar discussion of this information see Section 5.4.2. After completing calls to **dxp_add_board_item()** a call to **dxp_user_setup()** is required to perform the actual downloading of data to the physical boards.

- **board_type(1)** has 1 valid **values** entry, which is a characteristic used in a previous call to **dxp_add_system_item()**, all subsequent additions to the board item will use this board type.
- **interface(1)** has 2 valid **values** entries, which are the interface type, e.g. CAMAC or EPP in the 0th entry, and the I/O library name in the 1st entry. The I/O library name, which can signify whatever the designer of the **md** routines desires, is passed to the **dxp_md_initialize()** routine.
- **module(2)** has a minimum of 3 **values** entries. The 0th entry contains the **md** associated string that is used by **dxp_md_open()** to perform open an I/O channel, e.g. 2117 represents SCSI bus 2, crate 1 and slot 17 for the default **md** library provided by XIA. The 1st entry in the **values** array is the number of channels that this module contains, e.g. since the dxp4c board has 4 independent detector channels, this number is 4. Following the number of channels is a separate entry for each detector entry that contains the USER defined detector number. If the detector number is negative, the channel is considered broken or un-instrumented and ignored by XerXes. For example, **values**={“2117”,“4”,“0”,“1”,“2”,“-1”} is a valid array for a 4 channel board that has channel 3 broken as indicated by the -1 entry. Future references to reading out data from the channels is done via the USER defined detector numbers, 0,1 and 2.
- **default_dsp(3)** has 1 valid **values** entry, which is the filename of a valid DSP program, typically with filename extension .dsp or .hex (see Section 5.4.2.3). This program is loaded into memory for all channels of the current module. If a new **module(2)** characteristic is defined after a call with **default_dsp(3)**, the new module will default to this DSP program. This default can be overridden with another call with **default_dsp(3)**.
- **dsp(3)** has 2 valid **values** entry, which are the channel number in the 0th entry and the filename of a valid DSP program in the 1st entry, typically with filename extension .dsp or .hex (see Section 5.4.2.3). This program is loaded into memory for ONLY the channel number, specified in the 0th entry of **values**, of the current module. This channel number is relative to 0 and extends to the total number of channels minus 1 in the module (as defined in a previous call with **module(2)**). This channel number is not the same as the USER defined detector number. If a new **module(2)** characteristic is defined after a call with **dsp(3)**, the new module will not inherit the information specified by a call with **dsp(3)**, thus this information must be specified for every module that has a DSP program for a channel different than the other channels.
- **default_fippi(3)** has 1 valid **values** entry, which is the filename of a valid FiPPi firmware program, typically with filename extension .fip or .bin (see Section 5.4.2.2). This program is loaded into memory for all channels of the current module. If a new **module(2)** characteristic is defined after a call with **default_fippi(3)**, the new module will default to this FiPPi program. This default can be overridden with another call with **default_fippi(3)**.
- **fippi(3)** has 2 valid **values** entry, which are the channel number in the 0th entry and the filename of a valid FiPPi program in the 1st entry, typically with filename extension .fip or .bin (see Section 5.4.2.2). This program is loaded into memory for ONLY the channel number, specified in the 0th entry of **values**, of the current module. This channel number is relative to 0 and extends to the total number of channels minus 1 in the module (as defined in a previous call with **module(2)**). This channel number is not the same as the USER defined detector number. If a new **module(2)** characteristic is defined after a call with **fippi(3)**, the new module will not inherit the information specified by a call with **fippi(3)**, thus this information must be specified for every module that has a FiPPi program for a channel different than the other channels.
- **default_params(3)** has 1 valid **values** entry, which is the filename of a valid DSP parameter file (see Section 5.4.2.1). The information from this file is loaded into memory for all channels of the current module and provides default changes to the DSP parameters which are applied after downloading the DSP program to the channel. If a new **module(2)** characteristic is defined after a call with **default_params(3)**, the new module will default to this DSP program. This default can be overridden with another call with **default_params(3)**.

- **param(3)** has 2 valid **values** entry, which are the channel number in the 0th entry and the filename of a valid DSP parameter file in the 1st entry (see Section 5.4.2.1). This program is loaded into memory for ONLY the channel number specified in the 0th entry of **values** the current module. This channel number is relative to 0 and extends to the total number of channels minus 1 in the module (as defined in a previous call with **module(2)**). This channel number is not the same as the USER defined detector number. If a new **module(2)** characteristic is defined after a call with **param(3)**, the new module will not inherit the information specified by a call with **param(3)**, thus this information must be specified for every module that has default parameters for a channel different than the other channels.

5.7.7 int dxp_reset_channel(int *detChan)

This routine resets a single channel to its starting state. It performs the same actions as **dxp_user_setup()**, but limited to a single XIA module channel.

5.7.8 int dxp_locate_system_files(unsigned int *maxlen, char files[[3])

This routine returns 3 filenames dealing with system configuration. The user must supply the maximum length, **maxlen**, of a filename allocated by the calling routine, preventing XerXes from overwriting memory space. The filenames returned have any internal environment variables (e.g. **XIA_CONFIG**, **XIAHOME** or **DXPHOME**) translated to remove ambiguities related to what file is actually in use. If the file contains no path information, but is simply a filename, then the file was accessed from the current working directory, typically the directory in which the program was executed. The array **files** contains the following file names:

- **files[0]** = **XIA_CONFIG** file, the global system configuration file (section 5.4.1)
- **files[1]** = The modules configuration file (section 5.4.2).
- **files[2]** = The preamp configuration file (section 5.4.3).

5.7.9 int dxp_locate_channel_files(int *detChan, unsigned int *maxlen, char files[[3])

This routine returns 3 filenames associated with the channel configuration of **detChan**. The user must supply the maximum length, **maxlen**, of a filename allocated by the calling routine, preventing XerXes from overwriting memory space. The filenames returned have any internal environment variables (e.g. **XIAHOME** or **DXPHOME**) translated to remove ambiguities related to what file is actually in use. If the file contains no path information, but is simply a filename, then the file was accessed from the current working directory, usually the directory in which the program was executed. The array **files** contains the following file names:

- **files[0]** = The FIPPI firmware file (section 5.4.2.2)
- **files[1]** = The DSP firmware file (section 5.4.2.3).
- **files[2]** = The default DSP parameters file (section 5.4.2.1).

5.7.10 void dxp_version(void)

This routine prints the library version currently in use.

5.8 Routines to Download the DSP

5.8.1 int dxp_dspconfig(void)

This routine will download DSP firmware to all modules. This process will overwrite the current DSP parameter table and the user is responsible for loading local parameter values from the XIA modules.

5.8.2 int dxp_replace_dspconfig(int *detChan, char *filename)

This routine replaces the DSP program of a single detector channel. The new DSP program from the file named **filename** is read into memory and downloaded to the DSP in detector **detChan**. The DSP parameter memory will be set to the default values.

5.9 Routines to Download the FiPPi

5.9.1 int dxp_fipconfig()

This routine will download FiPPi firmware to all modules. The FiPPi file must be downloaded to the detector channel prior to loading of the DSP program in order for the DSP calibration routines to run properly. In addition, the DSP program should be downloaded after any change is made to the FiPPi program.

5.9.2 int dxp_replace_fipconfig(int *detChan, char *filename)

This routine replaces the FiPPi program of a single detector channel. The new FiPPi configuration stored in the file named **filename** is read into memory and downloaded to the FiPPi in detector **detChan**. The DSP program should be re-downloaded after any change to the FiPPi program.

5.9.3 int dxp_check_decimation(unsigned short *decimation, unsigned short *different)

This routine confirms that all detector channels have the same value for the DSP parameter DECIMATION, passed as **decimation**. If any channel returns a DECIMATION value different from **decimation**, the return parameter **different** will be non-zero. This routine is intended to cross-check the downloading of the FiPPi configurations after DSP code has been downloaded. This routine does not currently support FiPPi programs of differing DECIMATION between detector channels.

5.10 Routines to Control the Operation of the DSP

These routines control the general operation of the DSP and are divided into two sections, one related to the manipulation of the parameter memory and another for other aspects of DSP control.

5.10.1 Routines to Load and Update the DSP Parameter Values

5.10.1.1 int dxp_dspdefaults(int *detChan)

This routine transfers the default parameter values contained in memory into the internal parameter memory. Only the default values, which were previously read from a DSP parameters file (see Section 5.4.2.1), are replaced. The parameters read into memory are not downloaded to the module and a call to **dxp_put_dspparams()** or **dxp_replace_dspparams()** must be called to perform the actual downloading to the module.

5.10.1.2 int dxp_replace_dspparams(int *detChan)

This routine writes the DSP parameters stored in internal parameter memory to the detector specified by **detChan**. The parameter array can be set to default values using **dxp_dspdefaults()**.

5.10.1.3 int dxp_put_dspparams(void)

This routine writes the DSP parameters stored in the internal parameter memory to all detectors in the system. The parameter array can be set to default values using **dxp_dspdefaults()**.

5.10.1.4 int dxp_symbolname_list(int *detChan, char **list)

This routine returns the **list** of DSP parameter names previously read from the DSP program file for detector number **detChan**. The memory for **list** must be allocated by the calling routine. Use the **dxp_max_symbols()** routine to determine the number of parameters in **list**, and each entry in the list should be of length **MAXSYMBOL_LEN**.

5.10.1.5 int dxp_symbolname_limits(int *detChan, unsigned short *access, unsigned short *lbound, unsigned short *ubound)

This routine returns lists of access type, and lower and upper bound limit information for the DSP parameters used by detector number **detChan**. The memory for **access**, **lbound** and **ubound** must be allocated by the calling routine. Use the **dxp_max_symbols()** routine to determine the number of parameters in each array. The offset location of each parameter in these arrays matches the location of the names returned by **dxp_symbolname_list()**, i.e. the symbol name contained in **list[20]** from **dxp_symbolname_list()** has access type stored in **access[20]**. The **access** array information is interpreted as follows:

0. Read/Write
1. Read Only
2. Write Only.

If there are no imposed bounds on a parameter, both the **lbound** and **ubound** value are set to 0.

5.10.1.6 **int dxp_max_symbols(int *detChan, unsigned short *nsymbols)**

This routine returns the number of parameters, **nsymbols**, associated with the DSP program loaded for detector number **detChan**.

5.10.1.7 **int dxp_set_dspsymbol(char *name, unsigned short *value)**

This routine sets a single DSP parameter pointed to by **name** to **value**. The routine sets the parameter in the all modules configured in the system.

5.10.1.8 **int dxp_set_one_dspsymbol(int *detChan, char *name, unsigned short *value)**

This routine sets a single DSP parameter pointed to by **name** to **value**. The routine only sets the parameter in detector channel **detChan**.

5.10.1.9 **int dxp_download_params(int *nparam, unsigned short addr[], unsigned short value[])**

This routine changes a list of addresses pointed to by the **addr[]** array to the value in the **value[]** array, where the number of addresses to change is specified by **nparam**. The routine sets the list of values for all channels configured in the system. This routine is for experts; it can easily lead to unintended writing of DSP memory.

5.10.1.10 **int dxp_download_one_params(int *detChan, int *nparam, unsigned short addr[], unsigned short value[])**

This routine changes a list of addresses pointed to by the **addr[]** array to the value in the **value[]** array, where the number of addresses to change is specified by **nparam**. The routine only changes the channel pointed to by **detChan**. This routine is for experts; it can easily lead to unintended writing of DSP memory.

5.10.1.11 **int dxp_upload_dspparams(int *detChan)**

This routine reads all DSP parameters from the detector channel pointed to by **detChan**. The values of the parameters are stored in internal memory for each channel and can be downloaded with **dxp_replace_dspparams()**.

5.10.1.12 **int dxp_upload_channel(int *detChan, unsigned short *nparam, unsigned short *addr, unsigned long value[])**

This routine reads **nparam** parameters starting at an address offset of **addr** and returns the values using the **value[]** array. The routine only reads the values from one detector channel pointed to by **detChan**.

5.10.1.13 **int dxp_get_one_dspsymbol(int *detChan, char *name, unsigned short *value)**

This routine reads a single DSP parameter pointed to by **name** and returns its **value**. The routine reads the parameter from the detector channel pointed to by **detChan**.

5.10.1.14 **int dxp_get_dspsymbol(int *detChan, char *name, unsigned long *value)**

This routine reads a single DSP parameter pointed to by **name** and returns its **value**. The routine reads the parameter from the detector channel pointed to by **detChan**. The **name** does not have to be an exact match to a DSP parameter for this routine, rather the base name of a multi 16-bit can be specified. E.g. the **name** "FASTPEAKS" can be used to read out the total number of fast peaks from the DSP, where it is stored as FASTPEAKS0 and FASTPEAKS1, each of which is 16-bits, **dxp_get_dspsymbol()** returns the proper 32-bit version using the variable **value**.

5.10.1.15 **int dxp_get_symbol_index(int *detChan, char *name, unsigned short *symindex)**

This routine determines the index, **symindex**, of a single DSP parameter pointed to by **name** within the parameters array returned by other XerXes routines. The routine determines the index only for the detector channel pointed to by **detChan**.

5.10.1.16 **int dxp_symbolname_by_index(int *detChan, unsigned short *symindex, char *name)**

This routine returns the name of the DSP symbol at location **symindex**, as determined by **dxp_get_symbol_index()**. The calling routine must allocate memory (length is **MAXSYMBOL_LEN**, given in **xerxes_generic.h**) for **name**.

5.10.1.17 int dxp_mem_dump(int *detChan)

This routine dumps the full parameter memory of the DSP pointed to by **detChan** using the routine **dxp_md_puts()** which defaults to the screen, but is user definable in the machine dependent libraries. This routine reads the parameter values directly from the hardware, but does NOT modify the local copy of the parameter memory. Use **dxp_upload_dspparams()** or **dxp_upload_channel()** to perform local memory updating.

5.10.2 Routines to Change and Observe Other DSP Memory

5.10.2.1 int dxp_nspec(int *detChan, unsigned int *len_spec)

This routine returns the length of the spectrum memory as **len_spec** in 24 bit words for detector number **detChan**.

5.10.2.2 int dxp_nbase(int *detChan, unsigned int *len_base)

This routine returns the length of the baseline memory as **len_base** in 16 bit words for detector number **detChan**.

5.10.2.3 int dxp_nevent(int *detChan, unsigned int *len_event)

This routine returns the length of the event memory as **len_event** in 16 bit words for detector number **detChan**.

5.10.2.4 int dxp_readout_detector_run(int *detChan, unsigned short *params, unsigned short *baseline, unsigned long *spectrum)

This routine reads out the parameter memory, **params[]**, baseline histogram memory, **baseline[]**, and spectrum memory, **spectrum[]**, for the detector pointed to by **detChan**. The routine checks the error word in the parameter memory and clears it if set, reporting any error encountered. This routine is used after a run has been performed to readout the results. For each array, If any of the arrays, **params**, **baseline** or **spectrum**, are **NULL** (C standard definition), then the readout is not performed and no memory need be allocated by the host program for that array. This allows a user to only readout one set of information, e.g. the spectrum memory, by setting **params=baseline=NULL**.

5.11 Routines that Provide Hardware Setup Information

5.11.1 int dxp_ndxp(int *numDxp)

This routine returns the number of modules, **numDxp**, in the current system configuration.

5.11.2 int dxp_ndxpchan(int *numDxpChan)

This routine returns the number of detector channels, **numDxpChan**, in the current system configuration.

5.11.3 int dxp_get_detectors(int *numDxpChan, int detchan[])

This routine returns the number of DXP channels, **numDxpChan**, in the current system configuration and a list of the detector channel numbers, **detchan[]**. The values returned by this routine are defined by either using the **dxp_assign_channel()** routine that reads in the **module configuration file** or with calls to the **dxp_add_board_item()** routine.

5.11.4 int dxp_get_board_type(int *detChan, char *name)

This routine returns a string, **name**, that is the board type for **detChan**. This board type returned in **name** will match one of the types listed in section 5.3. The user is responsible for allocating enough memory to hold **name**, prior to calling this routine, the maximum length is specified in **xerxes_generic.h** as **MAXBOARDNAME_LEN** (currently set to 20).

5.12 Routines to Control Runs

5.12.1 int dxp_start_run(unsigned short *gate, unsigned short*resume)

This routine starts data taking for all channels in the current system configuration. If **gate** is set to 0, then the system will wait for the external gate signal to begin the run. If **resume** is set to 0, then the MCA memory is cleared at the start of a run, else the MCA memory is left untouched and the next run is added to the previous run.

5.12.2 int dxp_stop_run(void)

Stops data taking for all channels in the current system configuration.

5.12.3 **int dxp_resume_run(void)**

Resumes a previously stopped data taking run. All modules must have the run stopped for resume run to function properly. The previous usage of the gate signal is maintained, i.e. the user must start a new run with `dxp_start_run()` to change the usage of gate.

5.12.4 **int dxp_start_one_run(int *detChan, unsigned short *gate, unsigned short *resume)**

This routine starts a run in a single module. Because of hardware limitations, a run must be started on the whole module that contains `detChan`. Please see `dxp_start_run()` for further details.

5.12.5 **int dxp_stop_one_run(int *detChan)**

This routine stops the run on a single module. Because of hardware limitations, the run must be stopped for all channels of the module that contains `detChan`.

5.12.6 **int dxp_resume_one_run(int *detChan)**

This routine resumes a run for a single channel. Because of hardware limitations, a run must be resumed on the whole module that contains `detChan`. Please see `dxp_resume_run()` for further details.

5.12.7 **int dxp_isrunning(int *detChan, int *active)**

This routine determines if a run is active for `detChan`, returning the state in `active`. If set, bit one of `active` indicates that the hardware module itself thinks a run is active, bit 2 indicates that the XerXes library believes the run is active. If `active` equals zero, then no run is active for this channel.

5.12.8 **int dxp_isrunning_any(int *detChan, int *active)**

This routine determines if a run is active for any channel, returning the first active channel in `deChan` and run state in `active`. If set, bit one of `active` indicates that the hardware module itself thinks a run is active, bit 2 indicates that the XerXes library believes the run is active. If `active` equals zero, then no run is active for this channel.

5.12.9 **int dxp_enable_LAM (int *detChan)**

This routine will enable the LAM (Look At Me) signal for this detector channel. The definition of a LAM varies from device to device. For a CAMAC device it is the standard definition and is the origin of the term, however for an EPP device it has no meaning and will be ignored by XerXes. For future use by PCI devices, the LAM will generate a PCI interrupt. Please refer to the hardware manuals for proper use of the LAM.

5.12.10 **int dxp_disable_LAM (int *detChan)**

This routine will disable the LAM (Look At Me) signal for this detector channel. The definition of a LAM varies from device to device. For a CAMAC device it is the standard definition and is the origin of the term, however for an EPP device it has no meaning and will be ignored by XerXes. For future use by PCI devices, the LAM will generate a PCI interrupt. Please refer to the hardware manuals for proper use of the LAM.

5.12.11 **int dxp_reset_LAM (int *detChan)**

This routine will reset the LAM (Look At Me) signal to an off state for this detector channel. The definition of a LAM varies from device to device. For a CAMAC device it is the standard definition and is the origin of the term, however for an EPP device it has no meaning and will be ignored by XerXes. For future use by PCI devices, the LAM will generate a PCI interrupt. Please refer to the hardware manuals for proper use of the LAM.

5.13 **Routines to Perform Special Runs**

5.13.1 **int dxp_calibrate(int *calibtask)**

This routine performs an internal calibration function for all channels: e.g. start and stop runs and check results. Currently no calibration tasks are supported by this library for the DXP-4C2X modules for which all needed calibrations are performed automatically upon loading of the DSP program. For the DXP-4C modules, the following values of `calibtask` are supported,

1. Tracking DAC calibration (whichtest=7)
2. Reset Calibration (whichtest=2).

5.13.2 `int dxp_calibrate_one_channel(int *detChan, int *calibtask)`

This routine performs an internal calibration function only for **detChan**: e.g. start and stop runs and check results. Currently no calibration tasks are supported by this library for the DXP-4C2X modules for which all needed calibrations are performed automatically upon loading of the DSP program. For the DXP-4C modules, the following values of **calibtask** are supported,

1. Tracking DAC calibration (whichtest=7)
2. Reset Calibration (whichtest=2).

5.13.3 `int dxp_control_task_info(int *detChan, short *type, int *info)`

This routine returns information about a control task that will/was run on **detChan**. The **type** must be one of the choices defined in section 8. The **info** array is of length 3 and contains the following information:

- `info[0]` = length of the array that the user must allocate before calling `dxp_get_control_task_data()`.
- `info[1]` = recommended time in milliseconds to wait before attempting to stop the control run (for some modules, this involves polling the **BUSY** state).
- `info[2]` = recommended time to wait between polls to determine if the control run is finished. Thus the user is recommended to wait `info[1]` ms, then begin polling for a end run state every `info[2]` ms.

5.13.4 `int dxp_start_control_task(int *detChan, short *type, unsigned int *length, int *info)`

This routine starts a control task of **type** (a full list is in section 8) on channel **detChan**. Since some control tasks require information prior to starting the run, the user is required to pass that information when the control task is begun in the array **info** of length **length** (control task **types** that require special data are listed in section 8). The first **info** array entry (`info[0]`) always contains the number of iterations for the control task, typically related to the **LOOPCOUNT** DSP parameter.

5.13.5 `int dxp_stop_control_task(int *detChan)`

This routine stops a control task on channel **detChan**.

5.13.6 `int dxp_get_control_task_info(int *detChan, short *type, long *data)`

This routine returns information after a control task of **type** has been run on channel **detChan**. The data are returned in an array **data**, the length of which must be obtained from `dxp_control_task_info()` prior to calling `dxp_get_control_task_info()` (see section 8 for control task **types** that return data).

5.14 Routines to Obtain Information about a Run

5.14.1 `int dxp_get_statistics(int *detChan, double *lifetime, double *icr, double *ocr, unsigned long *nevent)`

This routine returns information about the previous run contained in the DSP parameter memory pointed to by **detChan**. The **lifetime** is the total time (in seconds) the channel was “live” and taking data and is used to determine the input count rate, **icr**, and output count rate, **ocr** (both returned in kHz). The **icr** is defined as the number of fast peaks detected by the DXP divided by the **lifetime**. The **ocr** is defined as the number of events in the MCA window + overflows + underflows, also referred to as the total number of accepted events, **nevent**, divided by the **lifetime**.

5.15 Routines to Change the Gain of the XIA Hardware

5.15.1 `int dxp_modify_gains(float *gainchange)`

This routine scales the DXP gain by a factor of **gainchange** and adjusts the threshold for triggering based on the new gains.

5.15.2 `int dxp_modify_one_gains(int *detChan, float *gainchange)`

Same behavior as `dxp_modify_gains()`, but only affecting detector channel **detChan**.

5.16 Routines to Access Files and Save Data

5.16.1 `int dxp_open_file(int *lun, char *filename, int *mode)`

This routine opens the file specified by **filename** and returns a number **lun** that points at the file for all future operations. The user is responsible for keeping track of the value of **lun** for future use and closing of the file. Supported values of **mode** are

0. open file for output
1. append data to an existing file
2. open a file for input.

5.16.2 `int dxp_close_file(int *lun)`

This routine closes the file pointed to by **lun** which was previously opened with `dxp_open_file()`. If **lun** is -1, then all files that have been opened with `dxp_open_file()` are closed.

5.16.3 `int dxp_save_dspparams(int *detChan, int *lun)`

This routine reads the DSP parameter memory of the detector pointed to by **detChan** and writes the data to a file pointed to by **lun**, previously opened with `dxp_open_file()`. This routine intended for diagnostic purposes, if you wish to save the state of an experiment, use `dxp_save_config()`.

5.16.4 `int dxp_write_spectra(int *luns, int *lunb)`

This routine reads out every detector channel in the current system configuration, displays some run statistics, and writes out all of the spectra to one file (**luns**) and all of the baseline histograms to a different file (**lunb**). Calls to `dxp_open_file()` prior to calling this routine to open files associated with **luns** and **lunb**. If **luns** or **lunb** are negative, then the writing of the file is skipped for the spectrum (**luns**) or baseline (**lunb**).

5.16.5 `int dxp_save_config(int *lun)`

This routine writes all configuration data to the file specified by **lun** and then reads the parameter memory of all detector channels and writes the memory in hexadecimal format to the same file. The file pointed to by **lun** must have been previously opened by a call to `dxp_open_file()`.

5.16.6 `int dxp_restore_config(int *lun)`

This routine reads a configuration file (previously saved with `dxp_save_config()`) and restores the system parameters contained in the file. The file pointed to by **lun** must have been previously opened by a call to `dxp_open_file()`.

5.17 Routines to Control Error Logging

5.17.1 `int dxp_enable_log()`

This routine enables error logging; the default state is enabled.

5.17.2 `int dxp_suppress_log()`

This routine suppresses all error logging; the default state is enabled.

5.17.3 `int dxp_set_log_level(int *level)`

This routine sets the level of error logging as defined by `md_generic.h`. The levels are

1. MD_ERROR
2. MD_WARNING
3. MD_INFO
4. MD_DEBUG.

Any errors that occur at a higher level will not be reported, i.e. setting **level** to MD_WARNING will report all MD_ERROR and MD_WARNING level errors, but not MD_INFO and MD_DEBUG. The default level is MD_ERROR.

5.17.4 int dxp_set_log_output(char *filename)

This routine directs all error logging to the file specified by **filename**. Each time this routine is called, any previously used file will be closed and the new one opened; using the same name will overwrite the current file destroying information contained in it. If **filename** is set to stdout or stderr, then the logging is redirected to the appropriate stream; a NULL pointer will redirect output to stdout.

5.18 Miscellaneous Utility Routines

5.18.1 int dxp_findpeak(long *array, int *nbins, float *thresh, int *lower, int *upper)

This routine implements a simple peak finding method. It looks for a maximum in **array[]** which has **nbins**, then search on either side of the maximum till the bin contents fall below **thresh** percent of the peak bin contents. The routine returns the peak limits as **lower** and **upper**, representing the upper and lower bin limits of the peak.

5.18.2 int dxp_fitgauss0(long array[], int *lower, int *upper, float *pos, float *fwhm)

This routine implements a simple Gaussian "fitter" based on the linearization method; it only computes peak position and FWHM and was taken from Knoll, "Radiation Detection and Measurement" 2nd Edition, John Wiley & Sons, 1989, p677-8. Input arguments are **array[]** which is an array of values representing histogram contents and bin limits, **lower** and **upper**, within which to perform the fit. The routine returns the central fitted position, **pos**, and full width at half maximum, **fwhm**, of the assumed Gaussian peak.

5.18.3 int dxp_lock_resource (int *detChan, short *lock)

This routine is used to lock a hardware resource from access by a different thread, allowing multiple threads to access the same hardware without conflicts. The actual locking is performed by the MD (Machine Dependent) layer via the routine **dxp_md_lock_resource()**.

6 API for the Machine Dependent Library

This library is used to allow XerXes to work under different operating systems and different hardware controllers. A default set of routines is provided that work under the 32-bit Microsoft Windows operating system. The libraries assume communication with the DXP-4C(2X) is performed with a Jorway CAMAC crate controller. If these conditions are not met, then changes in the machine dependent library will be necessary. The library also contains support for EPP communications with the DXPX10P modules.

6.1 Machine Dependent Structures

6.1.1 struct Xia_Io_Functions

Contains function pointers to the following routines: **dxp_md_io()**, **dxp_md_initialize()**, **dxp_md_open()**, **dxp_md_get_maxblk()** and **dxp_md_set_maxblk()**. The routine **dxp_md_init_io()** creates pointers in this structure to all the I/O functions in the **md** library.

6.1.2 struct Xia_Util_Functions

Contains function pointers to the following routines: **dxp_md_error_control()**, **dxp_md_error()**, **dxp_md_alloc()**, **dxp_md_free()**, **dxp_md_puts()** and **dxp_md_wait()**. The routine **dxp_md_init_util()** creates pointers in this structure to all the utility functions in the **md** library.

6.2 Machine Dependent Initialization

6.2.1 int dxp_md_init_util(Xia_Util_Functions funcs, char *type)

This routine will create a set of pointers to the **md** library utility routines. The pointers are all contained in the **Xia_Util_Functions** structure that is defined in **xerxes_structures.h**. The variable **type** specifies the hardware control mechanism, allowing a different set of utility routines for different types of CAMAC controllers, for example.

6.2.2 int dxp_md_init_io(Xia_Io_Functions funcs, char *type)

This routine will create a set of pointers to the **md** library I/O routines. These routines are not needed by most user applications but are used by the XerXes libraries extensively. The pointers are all contained in the **Xia_Io_Functions** structure that is defined in **xerxes_structures.h**. The variable **type** specifies the hardware control mechanism, allowing a different set of I/O routines for different types of CAMAC controllers, for example. The variable **type** is passed via the XerXes libraries as the **interface** entry of the **module configuration file** or the appropriate call to **dxp_add_board_item()**.

6.2.3 int dxp_md_initialize(int maxMod, char *name)

This routine performs global initialization of the machine dependent layer and is called when either the **module configuration file** is parsed or calls to **dxp_add_board_item()** are made by XerXes. The parameter **maxMod** is the maximum number of XIA modules allowed in the system. The variable **name** is a user defined parameter obtained from the **iolibrary** entry in the **module configuration file** or a call to **dxp_add_board_item()**. The variable **name** is used to specify any additional information needed to initialize the MD libraries. For example, the default libraries use **name** to point to the dynamic library used to communicate with the Jorway 73a CAMAC controller.

6.2.4 int dxp_md_open(char *name, int *ioChan)

This routine “opens” an I/O channel to the CAMAC crate. The routine really allocates a pointer to the location of an XIA module in the CAMAC crate. The default format of the string **name** is defined in the section describing the **module configuration file**, where **name** is identical to the first element of a following the tag **module**. The default format for **name** contains a branch number (representing the SCSI bus number), a crate number (a single SCSI bus can control multiple crates) and a station number (better known as a slot number within the CAMAC crate). The routine returns a status code indicating success.

6.3 Machine Dependent Memory Allocation

6.3.1 void *dxp_md_alloc(size_t length)

All memory allocation for XerXes is performed using this routine. The calling structure for this routine is the same as the ANSI C standard routine `malloc()`.

6.3.2 void dxp_md_free(void *array)

XerXes uses this routine to free all memory previously allocated with `dxp_md_alloc()`. The routine has the same calling structure as the ANSI C routine `free()`.

6.4 Machine Dependent User Interface

6.4.1 *****DEPRECATED***** void dxp_md_error_control(char *keyword, int *value) *****DEPRECATED*****

*This routine controls the level of error/information reporting. Currently the only **keyword** available is "print_debug" which assigns **value** to the internal variable `print_debug`. If the value of `print_debug` is not zero, whenever `md_error()` is called with an **error_code**=`DXP_DEBUG`, the message from the higher level routine is printed via `dxp_md_puts()`. This routine can be expanded to handle reporting different error states in a user defined fashion.*

6.4.2 int dxp_md_output(char *filename)

This routine lets the user define where to send all messages from the library. If `filename` equal "stderr" or "stdout", then messages are routed to the appropriate stream, however if `filename` is NULL or "stdin", then messages are defaulted back to stdout. If `filename` is none of these, then a write-only file is opened with name **filename**. Any file opened prior to calling this routine will be closed before the message stream is redirected to the currently requested destination. The default destination is stdout.

6.4.3 void dxp_md_error(char *routine, char *message, int *error_code)

This routine handles error reporting for XerXes. The name of the routine reporting the error is specified by **routine**. The error message and code are specified by **message** and **error_code** (code values are discussed in Section 5.6).

6.4.4 void dxp_md_warning(char *routine, char *message)

This routine handles warning reporting for XerXes. The name of the routine issuing the warning is specified by **routine**, while the warning message is contained in **message**. This routine is used to tell the user that something went wrong, but it is not catastrophic.

6.4.5 void dxp_md_info(char *routine, char *message)

This routine handles informational message reporting for XerXes. The name of the routine reporting the information is specified by **routine**, while the message itself is contained in **message**.

6.4.6 void dxp_md_debug(char *routine, char *message)

This routine handles special debug message reporting for XerXes. The name of the routine reporting the debugging information is specified by **routine**, while the message itself is contained in **message**.

6.4.7 int dxp_md_enable_log()

This routine turns on logging of messages from all libraries, but does not change the level or the destination of such messages.

6.4.8 int dxp_md_suppress_log()

This routine turns off logging of messages from all libraries.

6.4.9 int dxp_md_set_log_level(int level)

This routine sets the level of message logging, the higher the level the more messages reported from the library. The following constants are defined in `md_generic.h`, and are intended to be used in conjunction with this routine (as well as the corresponding wrapper in `xerxes.dll`, `dxp_set_log_level()`):

1. MD_ERROR
2. MD_WARNING

3. MD_INFO
4. MD_DEBUG.

The default setting is MD_ERROR, which only reports errors.

6.4.10 int dxp_md_log(int level, char *routine, char *message, int error)

This routine routes messages to the appropriate error, warning, info or debug routine based on the value of **level** (see definitions in **md_generic.h**). The parameter **error** is only used if the level is set to MD_ERROR, else only the values of **routine** and **message** are passed to the lower routines **dxp_md_error()**, **dxp_md_warning()**, **dxp_md_info()** and **dxp_md_debug()**.

6.4.11 int dxp_md_puts(char *s)

This routine prints the string(s) to the screen or other destination; the default destination is the screen. No direct calls to **printf()** are performed by XerXes, instead all output is directed to either the **md_error()** or **md_puts()** routines. The routine returns a status integer, the default return value is the return value of **printf()**.

6.5 Machine Dependent External I/O (CAMAC)

6.5.1 int dxp_md_io(int *ioChan, int *function, int *address, short *data, int *length)

This routine performs the IO call to read or write data via the CAMAC bus. The pointer to the desired IO channel (previously defined with a call to **dxp_md_open()**) is passed as **ioChan**. The address to write to is specified by **function** and **address**. The number of data words (16-bit words) and the data are pointed to by **length** and **data**, respectively. The return value is **DXP_SUCCESS** if no error is encountered, else **DXP_MDIO** is returned.

6.5.2 int dxp_md_set_maxblk(int *blksiz)

This routine sets the maximum number of words that can be transferred in one block by **md_io()**. This can change from system to system and from controller to controller. A **blksiz** size of 0 means to write all data at once, regardless of transfer size, and is the default. The routine returns **DXP_SUCCESS** if a non-negative **blksiz** is passed, else **DXP_NEGBLOCKSIZE** is returned.

6.5.3 int dxp_md_get_maxblk(void)

This routine returns the maximum number of words that can be transferred at once via **md_io()**. The routines in XerXea call **dxp_md_get_maxblk()** to determine the maximum size of a block transfer of data that an interface controller can handle. The return value of the routine is the maximum block size.

6.6 Miscellaneous Machine Dependent Routines

6.6.1 int dxp_md_wait(float *time)

This routine waits a specified **time** (in seconds) and then returns, performing no other actions. This allows the user to call routines that are native to different operating systems to wait a well-defined minimum time. This routine is not intended to be a precise method of timing data taking runs since the overhead in a computer program is typically too unpredictable. The routine returns **DXP_SUCCESS** if successful, no failed state is currently defined.

6.6.2 int dxp_md_lock_resource (int *ioChan, int *modChan, short *lock)

This routine allows the MD layer to lock a hardware resource so that another thread is unable to access the same hardware until the thread that gained access first has completed its operations. This is a user-defined operation and due to its nature will need to interact with global library variables to maintain what hardware is being accessed.

7 The host software release

This section describes the contents of the host software release. This release has been tested on a PC running Windows NT, using an Adaptec SCSI controller that was connected to a Jorway 73A CAMAC controller. The following files are provided in this driver distribution.

7.1 Header Files

The following file are used only in user routine compilations

xerxes.h – Header file containing host routine definitions for use by user.

The following files are used in compilation of both user routines and XerXes libraries

xerxes_errors.h – Header file containing XIA error definitions.

xerxes_generic.h – Header file containing XIA definitions for memory limits and other miscellaneous data.

xerxes_structures.h – Header file containing XIA structure definitions.

xerxesdef.h – Header file containing internal compile and linking definitions. Use the **XERXES_USE_DLL** preprocessor definition to obtain prototypes to link to sharable libraries. To link to static libraries, no preprocessor definitions are needed.

md_generic.h – used to define constants used by the MD layer.

The following files are used for compilation of the libraries only

dxp4c2x.h – Header file containing all public DXP4C2X driver definitions.

dxp4c.h – Header file containing all public DXP4C driver definitions.

x10p.h – Header file containing all public DXPX10P driver definitions.

g200.h – Header file containing all public DGF200 driver definitions.

xia_xerxes.h – Header file containing host routine definitions used when compiling XerXes.

xia_xerxes_structures.h – Header file containing private XIA structure definitions.

xia_dxp4c2x.h – Header file containing private definitions for DXP4C2X driver library.

xia_mddef.h – Header file containing internal compile and linking definitions. Use the **XIA_MD_USE_DLL** preprocessor definition to obtain prototypes to link to the sharable MD libraries. To link to static libraries, no preprocessor definitions are needed. NOTE: to use the MD libraries, the user still needs to call **dxp_md_init_util()** and **dxp_md_init_io()** to initialize the function pointers.

xia_dxp4c.h – Header file containing private definitions for DXP4C driver library.

xia_x10p.h – Header file containing private definitions for DXPX10P driver library.

xia_g200.h – Header file containing private definitions for DGFG200 driver library.

machine_dependent.h – Header file containing definitions for the machine dependent layer.

7.2 Source Files

xerxes.c – Source file for the XIA host software library. This should never be modified.

dxp4c.c – Source file for the DXP4C driver library. This should never be modified.

dxp4c2x.c – Source file for the DXP4C2X driver library. This should never be modified.

x10p.c – Source file for the DXPX10P driver library. This should never be modified.

machine_dependent_win95.c – Source file for the default machine dependent code. This should be used as a template for future changes to the machine dependent library.

7.3 Library Files

The following libraries are dynamic/sharable libraries. The *.lib files are used to link to the dynamic libraries.

xerxes.dll –DLL for the XerXes library.

xerxes_dll.lib –Lib file used to link to the DLL for the XerXes library.

dxp4c.dll –DLL for the DXP4C module.

dxp4c_dll.lib –Lib file used to link to the DLL for the DXP4C module.

dxp4c2x.dll –DLL for the DXP4C2X module.

dxp4c2x_dll.lib – Lib file used to link to the DLL for the DXP4C2X module.

dpx10p.dll –DLL for the DXPX10P module.

dpx10p_dll.lib –Lib file used to link to the DLL for the DXPX10P module.

md.dll – DLL for the machine dependent code. This is the compiled version of **machine_dependent_win95.c**.

md_dll.lib – Lib file used to link to the DLL for the machine dependent code. This is the compiled version of **machine_dependent_win95.c**.

camacdll.dll – Dynamic Link Library (DLL) for the Jorway 73A SCSI CAMAC controller under windows 9x.

The following libraries are static link libraries.

xerxes.lib – Static library for the XerXes library.

dxp4c.lib – Static driver library for the DXP4C module.

dxp4c2x.lib – Static driver library for the DXP4C2X module.

dpx10p.lib – Static driver library for the DXPX10P module.

md.lib – Static link library for the machine dependent code. This is the compiled version of **machine_dependent_win95.c**.

Appendix

8 Control Task Types

The following lists show all the control task type available for each board type supported by XerXes. This set of constants is found in `xerxes_generic.h`. For all input lists, the first entry (the number of iterations) is ignored in this section, all descriptions start with `info[1]` (see description of `dxp_start_control_task()` in section 5.13.4).

8.1 ALL XIA Modules

The following control tasks are valid for all XIA modules currently in production. Each task is given a brief explanation, for more detail the DSP programmer's manual should be consulted for the particular module.

- `CT_ADC` - Acquire an ADC trace (Oscilloscope mode)
 - `INPUT:` `info[1] = TRACEWAIT` – delay between each ADC sample
 - `OUTPUT:` array containing the ADC trace readout

8.2 DXP-4C

The following control tasks are valid for the DXP-4C modules. Each task is given a brief explanation, for more detail the DSP programmer's manual should be consulted for the DXP-4C.

- CT_DXP4C_ADC_TRACE - Acquire an ADC trace
See description of CT_ADC in section 8.1
- CT_DXP4C_RESETASC - Reset Slope Generator
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP4C_TRKDAC - TrackDAC calibration (CALIB_TRKDAC also acceptable)
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP4C_DACSET - Set the tracking, slope and gain DACs
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP4C_ADCTEST - Test the ADC Linearity
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP4C_ASCMON - Monitor ASC interrupts
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP4C_RESET - Reset calibration (CALIB_RESET also acceptable)
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP4C_FIPTRACE - Acquire Decimated Trace data
 - INPUT: NONE
 - OUTPUT: NONE

8.3 DXP-4C2X

The following control tasks are valid for the DXP-4C2X modules. Each task is given a brief explanation, for more detail the DSP programmer's manual should be consulted for the DXP-4C2X.

- CT_DXP2X_SET_ASCDAC -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_ADC_TRACE - Acquire an ADC trace
See description of CT_ADC in section 8.1
- CT_DXP2X_TRKDAC - TrackDAC calibration
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_SLOPE_CALIB -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_SLEEP_DSP - Idle the DSP allowing the downloading of firmware
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_PROGRAM_FIPPI - Download new values to the FIPPI from the DSP
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_SET_POLARITY -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_CLOSE_INPUT_RELAY - Close the input relay, isolating the module from external signals
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_OPEN_INPUT_RELAY - Open the input relay, allowing signals to pass to the ADC
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_RC_BASELINE -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_RC_EVENT -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXP2X_BASELINE_HIST - Acquire a history of the baseline averages
 - INPUT: NONE
 - OUTPUT: array containing the history of baseline averages
- CT_DXP2X_RESET -

- INPUT: NONE
- OUTPUT: NONE

8.4 DXP-X10P

The following control tasks are valid for the DXP-X10P modules. Each task is given a brief explanation, for more detail the DSP programmer's manual should be consulted for the DXP-X10P.

- CT_DXPX10P_SET_ASCDAC -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_ADC_TRACE - Acquire an ADC trace
See description of CT_ADC in section 8.1
- CT_DXPX10P_TRKDAC - TrackDAC calibration
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_SLOPE_CALIB -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_SLEEP_DSP - Idle the DSP allowing the downloading of firmware
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_PROGRAM_FIPPI - Download new values to the FIPPI from the DSP
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_SET_POLARITY -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_CLOSE_INPUT_RELAY - Close the input relay, isolating the module from external signals
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_OPEN_INPUT_RELAY - Open the input relay, allowing signals to pass to the ADC
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_RC_BASELINE -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_RC_EVENT -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DXPX10P_BASELINE_HIST - Acquire a history of the baseline averages
 - INPUT: NONE
 - OUTPUT: array containing the history of baseline averages
- CT_DXPX10P_RESET -

- INPUT: NONE
- OUTPUT: NONE

8.5 DXP-G200

The following control tasks are valid for the DXP-G200 modules. Each task is given a brief explanation, for more detail the DSP programmer's manual should be consulted for the DXP-G200.

- CT_DGFG200_SETDACS - Download values to the DACs on the module
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DGFG200_CLOSE_INPUT_RELAY - Close the input relay, isolating the module from external signals
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DGFG200_OPEN_INPUT_RELAY - Open the input relay, allowing signals to pass to the ADC
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DGFG200_RAMP_OFFSET_DAC - Ramp the offset DAC to determine a good baseline location
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DGFG200_ADC_TRACE - Acquire an ADC trace
See description of CT_ADC in section 8.1
- CT_DGFG200_PROGRAM_FIPPI - Download new values to the FIPPI from the DSP
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DGFG200_READ_MEMORY_FIRST - Read data from the on-module memory for the first time
 - INPUT: NONE
 - OUTPUT: data read from the memory
- CT_DGFG200_READ_MEMORY_NEXT - Read data from the on-module memory on subsequent reads
 - INPUT: NONE
 - OUTPUT: data read from the memory
- CT_DGFG200_WRITE_MEMORY_FIRST - Writedata to the on-module memory for the first time
 - INPUT: info[1:length] = data to be written into memory (can not exceed event buffer length)
 - OUTPUT: NONE
- CT_DGFG200_WRITE_MEMORY_NEXT - Write data to the on-module memory on subsequent writes
 - INPUT: info[1:length] = data to be written into memory (can not exceed event buffer length)
 - OUTPUT: NONE
- CT_DGFG200_MEASURE_NOISE -
 - INPUT: NONE
 - OUTPUT: NONE
- CT_DGFG200_ADC_CALIB_FIRST -
 - INPUT: NONE
 - OUTPUT: NONE

- CT_DGFG200_BASELINE_HIST - Acquire a history of the baseline averages
 - INPUT: NONE
 - OUTPUT: array containing the history of baseline averages
- CT_DGFG200_ADC_CALIB_NEXT -
 - INPUT: NONE
 - OUTPUT: NONE