MX for Area Detectors

William M. Lavender
Illinois Institute of Technology
Chicago, IL 60616 USA

October 26, 2006

MX has been developed by the Illinois Institute of Technology and is available under the following MIT X11 style license.

Contents

Chapter 1

Introduction

This manual describes how to install, configure, and use the MX beamline control toolkit *http://mx.iit.edu/* for use with area detectors. Currently, the only supported area detector hardware is the PCCD-170170 from AVIEX LLC, but it is anticipated that the list of supported detectors will grow with time.

   MX is capable of controlling a wide variety of other devices such as motors, counter/timers, MCAs, MCSs, and so forth. However, since this is of limited relevance to sites that only use MX for its area detector support, support for these other devices will not be described further in this manual.

1.1 Driver and Platform Support

1.1.1 MX Clients

The *network_area_detector* driver used by MX area detector clients runs on all of the platforms currently supported by MX. As of October 2006, the supported platforms include: *BSD, Cygwin, DJGPP, eCos, HP-UX, Irix, Linux, MacOS X, QNX, RTEMS, Solaris, Tru64, VMS, VxWorks, and Windows (Win32).*

1.1.2 MX Server

The platforms supported by the MX server depend on the drivers used by your area detector.

- **AVIEX PCCD-170170 CCD detector**

   For this detector, the top level area detector driver, *pccd_170170*, is layered on top of the video input driver, *epix_xclib_video_input*, for the imaging board used to read out frames from the AVIEX camera head. The video board itself is a PIXCI E4 frame grabber from EPIX, Inc. which is described here *http://www.epixinc.com/products/pixci_e4.htm*. The PIXCI E4 is a PCI Express board that plugs into an x4 PCI Express slot.

   Please note that the company, EPIX, Inc., has absolutely no relation to the control system called EPICS *http://www.aps.anl.gov/epics/*. The similarity of the names is completely a coincidence.

   The MX *epix_xclib_video_input* driver depends on the XCLIB library *http://www.epixinc.com/products/xclib.htm* provided by EPIX, Inc. for use with their PIXCI line of imaging boards. EPIX, Inc. currently provides versions of XCLIB for Windows XP, Windows 2000, Linux, and 32-bit DOS. They have stated that XCLIB for 64-bit Linux and Windows will be available by the end of 2006.

At present, MX support for the PIXCI E4 has only been tested in the following environments:

- – Fedora Core 5 Linux (x86) with GCC 4.1.1
- – Windows XP SP2 with Visual C++ 2005 Express.

Currently, our recommendation is to use Linux on the detector control computer. The main reason for this is that it is easier to do remote configuration and debugging of a Linux based system. If your firewall allows SSH connections to the detector control system, that is all that is needed to enable remote support.

It might be possible to get the Linux support working on another distribution, but this depends on the portability of the binary Linux module provided by EPIX, Inc. with their imaging boards. However, portability of the EPIX binary module for Linux has not yet been significantly tested by us. The simplest path would be to stick with Fedora Core, although Red Hat Enterprise Linux is quite similar and would probably work as well.

Please note that the EPIX XCLIB programming library is not freely available for download. In order to get it, you must purchase a single developer XCLIB license from EPIX, Inc. As of late 2006, this license costs $495.

- **Software emulated area detector**

  The *soft_area_detector* driver attempts to emulate enough of the behavior of a real area detector that higher level code will mostly not notice the difference. The *soft_area_detector* driver is layered on top of a video input driver in a similar manner to the design of the *pccd_170170* driver mentioned above. The video input driver can either be an emulated video input or a real video input. The following two choices are the most useful ones:

  - – **Software emulated video input**

    The *soft_video_input* driver generates test images and returns them to the caller. The type of test image is, in principle, selectable. However, at the moment the only available option is to return a repeating sequence of four test images, each of which has a maximum in a different corner of the image. This driver is available on all MX platforms.

  - – **Video4Linux 2**

    The *v4l2_input* driver uses any commercial frame grabber or TV capture card that has a Video4Linux version 2 driver. Video4Linux version 1 drivers are not supported. With this driver, it is possible to easily change the test image merely by pointing your video camera at a different target. Unfortunately, Video4Linux 2 only supports 8-bit greyscale images, but the MX area detector support will automatically adjust to compensate for this.

    Obviously, this driver is only available on Linux platforms. Not all Linux distributions include the header file **/usr/include/linux/videodev2.h** in their core *glibc* development package, so you must explicitly enable this driver by making the following definition in the file **mx/libMx/mxconfig.h**:

    ```
    #define HAVE_VIDEO_4_LINUX_2              1
    ```

1.2   Installation from Prebuilt Binaries

Although the server and the client may use different sets of binaries, the process of installation is essentially the same for both computers. Thus, the following description applies to both the server and its clients unless otherwise indicated.

Prebuilt Binaries for MX are generally delivered in the form of .tar.gz archives or .zip archives. The first thing to do is to decide which directory you want to install MX in. Although you are free to install the binaries anywhere

you want, the conventional place to install MX is either in the directory **/opt/mx** on Linux/Unix or the directory **c:\opt\mx** on Windows. On Linux/Unix systems with support for symbolic links, we commonly install to a directory with a name that contains the MX version number such as **/opt/mx-1.4.0** and then use a symbolic link to link this name to **/opt/mx**. In this case, the process looks like this

```
cd /opt
mkdir mx-1.4.0
ln -s mx-1.4.0 mx
```

The last step of this process cannot be done on Windows systems, since they do not support symbolic links.

At this point, you may jump forward to Section 1.4.

1.3   Installation from Source

We describe here the process of building and installing MX from source code on the detector server computer and follow that with comments about the client computer.

1.3.1   Before Building MX

For real area detector hardware, MX controls the image capture card using software libraries provided by the maker of the image capture card.

EPIX PIXCI cards

Prior to building the server side copy of MX, you must download and install the XCAP and XCLIB packages from the EPIX web site and FTP site, which you can find on the web page *http://www.epixinc.com/support/files.htm*. This web page redirects you to various directories on their FTP site underneath the directory *ftp://ftp.epixinc.com/software/*. Occasionally, newer versions may be found in the directory *ftp://ftp.epixinc.com/downloads/* and you may need to use a version from here if directed to by AVIEX or EPIX personnel. However, normally the versions found in *ftp://ftp.epixinc.com/software/* are the preferred versions.

XCAP

The XCAP package contains both the kernel mode driver needed by the EPIX software and a user GUI called *xcap* which can be used to control the imaging board. As of October 2006, you may download this from the directory *ftp://ftp.epixinc.com/software/xcap_v22/*. For Linux, you will want to download the file *xcaplnx_i386.bin*, while for Windows you will want to download the file *xcapwi.exe*.

For the EPIX PIXCI E4 card used by the AVIEX PCCD-170170, you install XCAP using the procedure described in the User's Manual for the card, which may be found at *http://www.epixinc.com/manuals/pixci_e14el/index.htm*. The actual installation is described in chapter 3 of that manual. The PIXCI E4 is bundled with a license for the XCAP-Lite version of the program. XCAP-Lite is what we have used for the development of the MX drivers. On Linux, by default, the software will end up in the directory **/usr/local/xcap**, while on Windows, it will end up in the directory **C:\XCAP**.

We do not recommend that users use the XCAP-Lite version of the program for normal operation, since it has limited functionality. In addition, XCAP-Lite will not start if you have configured the kernel mode image buffer for more than approximately 64 megabytes of memory. At present, we only use XCAP-Lite to generate configuration files that are to be read by XCLIB.

XCLIB

The XCLIB library is used by third-party application packages like MX to control EPIX PIXCI imaging boards. You may download the current version of the library from the EPIX FTP site's directory *ftp://ftp.epixinc.com/software/xclib_v22/*. The license for the development package is *not* included with the purchase price of the EPIX PIXCI E4. Instead, it is an additional charge of $495 as of late 2006.

For Linux, you should download the *xcliblnx* version for Fedora Core 4, which works just fine on Fedora Core 5. For Windows XP, you should download the file *xclibwnt.exe*. Please be careful to download files whose names begin with *xclib* with a **b** and not files whose names begin with *xclip* with a **p**. The *xclip* versions include image processing functionality not used by MX and the license cost for those versions is higher. By default, XCLIB will be installed to **/usr/local/xclib** on Linux and to **C:\XCLIB** on Windows. The procedure for installing XCLIB is not described in any document on the EPIX web site. It is only described in the manual distributed with licensed versions of the XCLIB development kit.

Please note that in the XCLIB download directory, there are versions of the library for Mandrake 9.1 using the older Linux 2.4 kernel, for Windows 95/98/ME and for the 32-bit DOS Watcom compiler. Currently, we have no plans to support these older operating system platforms.

Cygwin *(for Windows platforms)*

Although MX can be compiled on Windows with Microsoft Visual C++, Borland C++, MinGW, Cygwin, or DJGPP, the build process is *always* managed using the Cygwin version of *Make*. Microsoft's *Nmake* cannot be used since the MX makefiles make use of features not available in *Nmake*.

Cygwin can be found at the web site *http://www.cygwin.com/*. In brief, the procedure consists of downloading and running Cygwin's **setup.exe** installation program to download and install the Cygwin packages you need. The bare minimum needed are the base packages and the *make* package plus their dependencies. When building MX himself, the author of MX generally installs a variety of other packages including the Cygwin SSH server to create a more congenial development environment, but this is not absolutely necessary.

1.3.2   Downloading MX

The MX home page can be found at *http://mx.iit.edu/*. Official releases can be found on the download page *http://mx.iit.edu/source.html*. In the future, when area detector support has been made part of an official MX 1.4.0 release, this will be the best place to get the source code. However, until this happens, you will need to download a development snapshot or else checkout the code from the MX Subversion repository.

Development snapshots can be downloaded from the directory *http://mx.iit.edu/src/devel/*. In general, you should use the development snapshots rather than code checked out from Subversion, since the development snapshots can generally be counted on to compile and run on the most commonly used MX platforms such as Linux, Windows, MacOS X, and Solaris.

If you really do need the newest possible code, then you must checkout the code from the MX Subversion repository. The various repositories can be browsed with a web browser at *http://mx.iit.edu/svn.html*. The core MX package can be checked out with a command like this:

```
svn checkout http://svn.csrri.iit.edu/mx/trunk  mx
```

This will create a new MX source tree in the subdirectory **mx** which is ready to be configured and built.

Please bear in mind that the Subversion repository is in constant change with commits sometimes taking place several times a day. The most recent commit is not guaranteed to work correctly or even compile correctly at any given moment, so, in general, you are better off using a development snapshot as described above.

1.3.3   Building MX for the Detector Computer

Building MX is a multistep process.

1. The first step is to edit the top level makefile. Relative to the directory you unpacked MX into, the name of this file is **mx/Makefile**.

   In the makefile, the first thing that you will want to do is to change the value of the variable MX_ARCH to match the platform that you are compiling the code. For AVIEX area detectors, the only suitable choices are `linux` and `win32`.

   Then, you will need to change the value of the MX_INSTALL_DIR variable. In general, this can be anywhere you want. Typically, the author generally chooses a name containing the version number of MX such as **/opt/mx-1.4.0** on Linux or **c:/opt/mx-1.4.0** on Windows. On Linux, this directory will generally be symlinked to **/opt/mx**.

   Warning: For Windows, you must use *forward slashes* $(/)$ in the file name, rather than *backslashes* $(\backslash)$. Using backslashes here will cause the build process to fail.

   A more general warning is that you must not use filenames that include spaces anywhere in the MX makefiles, since that will also cause the build to fail. On Windows, this means that you must use **c:/progra~1** rather than **c:/Program Files** and **c:/docume~1** rather than **c:/Documents and Settings**. This is all a consequence of the way that Gnu make processes filenames.

2. The next step is to edit the file **mx/libMx/mxconfig.h**. This file is used to enable or disable the compilation drivers that make use of software libraries external to MX that are not guaranteed to be available. For the AVIEX PCCD-170170 detector, you will need to define the HAVE_EPIX_XCLIB flag as follows:

   ```
   #define HAVE_EPIX_XCLIB                 1
   ```

   If you wish to include the Video4Linux 2 driver for test purposes as well, then you will also need to make the definition

   ```
   #define HAVE_VIDEO_4_LINUX_2           1
   ```

   All of the other defines should be set to 0.

3. The last file to edit is **mx/libMx/Makehead.linux** or **mx/libMx/Makehead.win32**, depending on your platform. What you must do here is make sure that the correct version of the macros INCLUDES, LIB_DIRS, and LIBRARIES is uncommented. On Linux, the correct version to uncomment is

   ```
   INCLUDES  = $(MX_INCLUDES)
   LIB_DIRS  = -L$(MX_LIB_DIR) $(EPIX_XCLIB_LIB_DIRS)
   LIBRARIES = $(EPIX_XCLIB_LIBRARIES) -lpthread -lrt -lm
   ```

   while on Windows, the correct version is

   ```
   INCLUDES  = $(MX_INCLUDES) $(EPIX_XCLIB_INCLUDES)
   LIBRARIES = $(WIN32_LIBS) $(EPIX_XCLIB_LIBS)
   ```

If the EPIX, Inc. XCLIB library was not installed in the default location, you will also need to find the definition of the macro EPIX_XCLIB_DIR and change it to match the location that XCLIB was installed to. Please note that on Windows you must use escaped backslashes (\\) in the filenames mentioned here.

4. Now go to the top level source code directory **mx** and type the command *make depend*. This step constructs files named **Makefile.depend** in each of the **mx/libMx**, **mx/motor**, **mx/server**, **mx/update**, and **mx/util** directories. These are used to determine which files must be recompiled when a given MX file is modified. This step is not strictly necessary, but it does no harm to do it.

5. At this point, we are ready to compile and link MX. To do this, type the command *make* in the top level directory **mx**. If you are building a snapshot or a released version of MX, this should compile without any errors. If you get any errors, this should be reported to the author.

   If you obtained the source code tree you are trying to compile from the *trunk* branch of the MX Subversion repository, not all revisions found there are guaranteed to compile or work correctly. Normally, you should only try the Subversion repository directly if you require some feature that is not already available in a released MX version or a snapshot version from the *http://mx.iit.edu/src/devel/* directory.

6. The final step is to type *make install* in the top level directory **mx**. You must make sure to do this using an account that has permission to write to the installation directory. If this is a system directory, you will need to *su* to root on Linux, or switch to an account with adminstrative privileges on Windows. If you are merely installing a private copy into your own directory, then changing accounts will not be necessary.

1.3.4   Building MX for Client Computers

Building MX on a client computer uses essentially the same process as building it for the detector computer. In fact, if you wish, you may copy the version you built for the detector computer to your client computer. On Linux, this should just work, since the EPIX XCLIB library is statically linked to **libMx.so** as long as the version of *glibc* on the client computer is the same version or newer than the version on the computer you originally compiled MX on. However, on Windows, if you want to do this, you will need to copy the EPIX XCLIB DLL to the client computer.

If you wish to compile MX separately for the client computer, only a few changes need to be made. First, you must make the definition

```
#define HAVE_EPIX_XCLIB                 0
```

in the client's copy of **mx/libMx/mxconfig.h**. You must also make sure that the INCLUDE, LIB_DIRS, and LIBRARIES macros in your platform specific makefile do not include any references to the EPIX libraries. In addition, if you are compiling on a different platform, you will need to change MX_ARCH in the top level Makefile to match your platform. The client side version of MX should compile on any of the platforms mentioned in Section 1.1.1.

1.4   Configuring MX

If you are at a site that is using MX to control the entire beamline, setting up the configuration files can be a fair amount of work and will not be covered in this document. However, if you are only using MX to control your area detector, then you should be able to use the example configuration files shown in this document essentially as is.

1.4.1 Environment Variables

For MX to operate, you must set up two or three environment variables correctly. These are:

MXDIR - This environment variable should contain the name of the top level installation directory for MX. Typically, for Linux/Unix systems, this will be **/opt/mx**.

For Windows systems, this variable typically will be **c:/opt/mx**. Please note that for this case on Windows systems, you must use a forward slash (/).

PATH - You must also add the application binary directory to the PATH environment variable. For Linux/Unix systems, this will generally be **$MXDIR/bin**.

On Windows, if **MXDIR** is **c:/opt/mx**, then the correct string to add to the path will be **c:\opt\mx\bin**. Note that for the **PATH** variable, you use backslashes (\)

For Linux/Unix systems, there is generally one more environment variable to set.

LD_LIBRARY_PATH - For most Linux and Unix platforms, this environment variable tells the operating system where to find shared libraries. Generally, it should be set to the value **$MXDIR/lib**. However, a couple of supported MX platforms use an environment variable with a different name for this purpose. On MacOS X, the correct environment variable to use is **DYLD_LIBRARY_PATH**, while on HP-UX the correct environment variable is **SHLIB_PATH**.

In the MX source code tree, the file **mx/scripts/mxsetup.sh** is a full featured example of what needs to be done on Linux/Unix systems with a shell using the Bourne shell syntax. On most systems, all you will need to do is to define the environment variable MXDIR before sourcing **mxsetup.sh**. Usually MXDIR should be defined to have the same value as the MX_INSTALL_DIR from the top level MX makefile. If you *make install* to one location and then copy the installed binary tree to a different location, you will need to change the MXDIR environment variable to point to the top level directory in the new location.

The **mx/scripts** directory also contain briefer examples of what to do for the Linux/Unix C shell (**mxsetup.csh**) and for Windows (**mxsetup.bat**).

1.4.2 Configuring MX for Client Computers

MX client computers generally need to setup one configuration file. This file will have the same contents regardless of whether the server manages a real area detector or a software emulated area detector. Traditionally, this file has the name **$MXDIR/etc/motor.dat**, although you can actually use any filename you want as long as you pass the correct filename to *mx_setup_database()*. A file like the following should work for most installations:

```
adserver server network tcp_server "" "" 0x20000000 192.168.137.3 9727
ad       device area_detector network_area_detector "" "" 8 adserver ad
```

The first field is the *server_flags* field. Currently if you are using the software emulated area detector, you should set this line to 0x20000000, which tells the client to use blocking I/O. This is due to a limitation of the functionality of the software emulated area detector and will not be necessary with a real area detector. For a real area detector, you should set this field to 0x0 so that the client can time out if the server fails.

The 192.168.137.3 field is the IP address of the server. You can also specify a domain name like *mxserver.example.com* here, but I generally recommend using numerical IP addresses here, since this means that you can skip doing a Domain Name Service lookup when your client starts. The last argument 9727 is the port number of the MX server on

the remote computer. Although most installations will use the standard value of 9727, the port number is selectable so that you can run more than one MX server on a given computer.

In the second line, the string *ad* at the start of the line is the name of the MX record that controls your area detector. This name must be passed to *mx_get_record()* after you invoke *mx_setup_database()*. At the end of the line, the number 8 is the maximum number of ROIs configured for this area detector. This value should match the value set in the server's configuration file. *adserver* tells MX that this device is controlled by the MX server defined on the first line, while the trailing *ad* entry is the name of the area detector record on the remote MX server.

1.4.3   Configuring MX for Area Detector Servers

There are four files you may need to configure for an MX server.

1. **$MXDIR/etc/mxserver.acl**

   This file contains a list of IP addresses or domain names that are permitted to connect to the MX server. There should be one address per line. Here is an example.

   ```
   192.168.137.3
   192.168.137.6
   192.168.238.*
   *.example.com
   beamline?.example.net
   ```

   Note that the wildcard character "*" can match any string of characters, while the wildcard character "?" only matches a single character. Thus, for this example, any computer on the 192.168.238 subnet or at example.com can connect to the server. In addition, any computer with a name like beamline1.example.net, beamline2.example.net, and so forth, can also connect.

   Please note that the MX server checks the address of the remote client before reading even a single byte from the client. If the client's host IP is not on the access control list, the connection is dropped without sending any response back.

2. **$MXDIR/etc/mxserver.dat**

   This file configures the devices used by the MX server and is described in detail below.

3. **$MXDIR/etc/mxserver.opt**

   If it exists, this file supplies additional command line arguments to the MX server. The most commonly used argument here is *-t*, which tells the server to print out the name of each record, just before it starts to configure the hardware for that record. This option is useful for cases where the MX server hangs during startup if you want to see which record is causing it to hang. For a system with only a few records, such as an area detector, it is probably simplest to always specify *-t* here.

4. **$MXDIR/etc/mxuser.dat**

   If it exists, this file lists the name of the user account that the MX server should be run under. At present, this feature only works on Linux/Unix systems and only functions if the server startup script **$MXDIR/sbin/mx** is started as root. In general, we recommend that the MX server be run as a non-root user such as *mx* to limit the scope of damage if someone were to break into your computer via the MX server. So far this has never happened, but it is best to be prepared.

1.4.4 Configuring mxserver.dat

The contents of the **$MXDIR/etc/mxserver.dat** depend on which kind of area detector you are using.

AVIEX PCCD-170170

For the AVIEX PCCD-170170 CCD detector, you should use an **mxserver.dat** file that looks like this:

```
xclib interface generic epix_xclib "" "" /opt/mx/etc/pccd_170170.fmt
port  interface camera_link epix_camera_link "" "" 9600 0.5 1
epix  device video_input epix_xclib_video_input "" "" 4096 4096 GREY16 -1 1 port 0x0
ad    device area_detector pccd_170170 "" "" 8 "" "" "" "" epix port 0x1
rs232 interface rs232 camera_link_rs232 "" "" 9600 8 N 1 N 0x0 0x0 -1 0x0 port
```

In general, you should not modify the contents of the first three lines of this file, unless recommended to by AVIEX personnel.

However, you may have reason to modify the fourth line which defines the record *ad* which represents the area detector as a whole. The fields in this record have the following meanings:

Name	Value	Datatype	Description
name	*ad*	1-d string	Name of the record.
mx_superclass	*device*	recordtype	Driver superclass.
mx_class	*area_detector*	recordtype	Driver class.
mx_type	*pccd_170170*	recordtype	Driver type.
label	""	1-d string	Intended for use by GUIs.
acl_description	""	1-d string	ACLs not yet implemented.
maximum_num_rois	*8*	long	Maximum number of ROIs
mask_filename	""	1-d string	Name of the file containing the mask frame.
bias_filename	""	1-d string	Name of the file containing the bias frame.
dark_current_filename	""	1-d string	Name of the file containing the dark current frame.
flood_field_filename	""	1-d string	Name of the file containing the flood field frame.
video_input_record	*epix*	record	The name of the video input record.
camera_link_record	*port*	record	The name of the Camera Link record.
initial_trigger_mode	*0x1*	hex (unsigned long)	Only internal trigger (0x1) is supported at this time.

For arbitrary drivers, you can get a fairly terse list of the field names and their datatypes by running the command *$MXDIR/bin/mxdriverinfo -f pccd_170170*, if you replace *pccd_170170* with the name of the driver you are really interested in. In general, we will not describe the detailed meanings of records that you are not expected to change.

The first modifiable field in the *ad* record is the *maximum_num_rois* field which, as it says, sets the maximum number of regions of interest available to the user. In the example above, it is set to 8. In general, both the client and the server should be configured to use the same maximum number of regions of interest. Each region of interest uses enough memory to hold a copy of the region of interest data. If the region of interest covers the entire frame, then it will use as much memory as an entire frame. However, the memory for a given region of interest is not allocated until the first time it is used.

Next come the *mask_filename*, *bias_filename*, *dark_current_filename*, and *flood_field_filename* fields which specify the names of the initial files to be used for image correction. If they are not set to empty strings, the contents of the specified file will be loaded into the MX server at server startup time. If a given correction filename is set to "", then that correction will not be performed.

The *video_input_record* and *camera_link_record* fields should not be changed. A later version of this driver will provide for external triggers, but for now you should set the *initial_trigger_mode* field to 0x1 meaning internal trigger.

Software Emulated Area Detector with Software Generated Images

In order to test using an area detector emulated entirely in software, you should use the following **mxserver.dat** file:

```
video device video_input soft_vinput "" "" 4096 4096 GREY16 -1 1
ad    device area_detector soft_area_detector "" "" 8 "" "" "" "" video 0x1
```

In the record description for the *soft_area_detector* record, the field containing the number 8 is the *maximum_num_rois* field and the four fields following it are the various correction files, just like for the *pccd_170170* driver discussed above. The last two fields are the *video_input_record* and *initial_trigger_mode* fields as described above.

The definition of the *soft_vinput* record should probably be left alone, unless you want to change the image framesize from 4096 by 4096 to something else. The last field in this record is the *image_type* field. In the future, changing this field will allow you to change the type of image generated. At present, the only supported value is 1, which causes the generated images to repeat with in a cycle of four frames, with the maximum intensity in a different corner for each of the four frames.

Software Emulated Area Detector with Video4Linux 2 Generated Images

This mode of operation replaces the generation of image frames in software with frames read in from a video camera or a TV capture card. For this case, you should use an **mxserver.dat** that looks like this:

```
video device video_input v4l2_input "" "" 640 480 GREY8 -1 /dev/video0 1
ad    device area_detector soft_area_detector "" "" 8 "" "" "" "" video 0x1
```

Although this driver makes many kinds of software testing easier, it is not able to generate images as large as a typical area detector and most consumer cameras do not support square images. In addition, Video4Linux 2 only supports 8-bit greyscale. Thus, it is a less faithful emulation of the real behavior of an area detector.

Software Emulated Area Detector with Images from a Directory of Frames

In the near future, we will add a new video input driver called *file_vinput*. This driver will generate the image frames sent to clients by reading them from a directory of already existing frames, such as from another area detector. The only thing holding this up is that we have not yet written the I/O routines for reading and writing the necessary file formats. In the future, this driver will give the most faithful emulation of a real area detector.

1.4.5   Running the MX Server

Linux

For Linux and Unix systems, MX comes with a System V style startup which is installed at the location **$MXDIR/sbin/mx**. To manually start, stop, or restart the MX server, you can use the System V style syntax, assuming MXDIR=/opt/mx:

```
/opt/mx/sbin/mx start
/opt/mx/sbin/mx stop
/opt/mx/sbin/mx restart
```

To get automatic startup of the MX server when the machine boots, all you need to do is to arrange that the **$MXDIR/sbin/mx** script be invoked at system startup time. On a Linux computer that uses the System V style init, you must go into the **rc?.d** directories for each run level that you want to run the MX server in and make a symbolic link to the **$MXDIR/sbin/mx** script.

For example, on a Fedora Core system with MXDIR=/opt/mx, you would go into the directory **/etc/rc.d/rc2.d** and make a symbolic link with the following command

```
ln -s /opt/mx/sbin/mx S99mx
```

This tells the System V init system that you want to start the MX server at the end of system startup for run level 2. You would add similar symbolic links to the other run levels that you want it to start in.

For system shutdown, you make similar symbolic links in the run level directories for run levels 0 and 6. For example,

```
ln -s /opt/mx/sbin/mx K00mx
```

tells *init* to shut down the MX server at the start of system shutdown or reboot.

Windows

On Windows, you can start the MX server by running the batch file. **$MXDIR/sbin/mx.bat**. This file is merely a wrapper around the **$MXDIR/sbin/mxserver.exe** binary to set up its command line arguments correctly. The simplest way to get the server to start automatically when the Windows machine boots is to configure the Windows machine to automatically log into the account you will be running the MX server from and then configure the **mx.bat** script as a Startup item. It may be possible to run the MX server as a service using the **srvany.exe** program from the Windows Resource Kit, but we have not tested this.

Chapter 2

Using the MX Area Detector API

2.1 Building an MX Client

Setting up the makefile to build an MX client is quite straightforward. Here is an example for Linux or Unix:

```
#
# Use defines like these if you are linking to an installed version of MX.
#
MXDIR = /opt/mx-1.4.0

MX_LIB_DIR     = $(MXDIR)/lib
MX_INCLUDE_DIR = $(MXDIR)/include

#
# Use defines like these if you are linking to a copy of MX in the original
# build directory.
#

MX_LIB_DIR     = /home/lavender/mxdev/mx-1.4.0/mx/libMx
MX_INCLUDE_DIR = $(MX_LIB_DIR)

CFLAGS = -g -DOS_LINUX -DDEBUG -Wall -Werror -I$(MX_INCLUDE_DIR)

mx_client: mx_client.o
        gcc -g -o mx_client mx_client.o -L$(MX_LIB_DIR) -lMx

mx_client.o: mx_client.c
gcc $(CFLAGS) -c mx_client.c

clean:
-rm *.o mx_client
```

Here is the same example for Windows using Visual C++ 2005 Express:

```
#
# Do not forget that filenames and pathnames supplied to Gnu make
# must _not_ include spaces.  That is the reason for the use of
# progra~1 below rather than "Program Files".
#
MSDEV_DIR = c:\\progra~1\\micros~4

WIN32_LIBS = $(MSDEV_DIR)\\lib\\wsock32.lib $(MSDEV_DIR)\\lib\\winmm.lib \
$(MSDEV_DIR)\\lib\\advapi32.lib

#
# Use defines like these if you are linking to an installed version of MX.
#
MXDIR = c:\\opt\\mx-1.4.0

MX_LIB_DIR     = $(MXDIR)\\lib
MX_INCLUDE_DIR = $MXDIR\\include

#
# Use defines like these if you are linking to a copy of MX in the original
# build directory.
#

MX_LIB_DIR     = c:\\docume~1\\lavender\\mxdev\\mx-1.4.0\\mx\\libMx
MX_INCLUDE_DIR = $(MX_LIB_DIR)

CFLAGS = -DOS_WIN32 -DDEBUG -nologo -Zi -WX -I$(MX_INCLUDE_DIR)

mx_client: mx_client.obj
link /debug /nologo /out:mxserver.exe mx_client.obj \
/nodefaultlib:libc $(MX_LIB_DIR)\\libMx.lib $(WIN32_LIBS)

mx_client.obj: mx_client.c
cl $(CFLAGS) -c mx_client.c

clean:
-rm *.o mx_client
```

2.2 Initializing MX

2.2.1 mx_setup_database()

Most MX client programs should start by calling an MX utility function called *mx_setup_database()*. Calling it looks like this:

```
...

MX_RECORD *mx_database;
char mx_database_filename[] = "/opt/mx/etc/motor.dat";
mx_status_type mx_status;

mx_status = mx_setup_database( &mx_database, mx_database_filename );

if ( mx_status.code != MXE_SUCCESS )
        return mx_status;
...
```

mx_setup_database() encapsulates all of the operations required to use a database file to initialize a running MX database in your client or server process. Once *mx_setup_database()* has returned successfully, you can be sure that you have successfully connected to the remote server(s) and your runtime database is ready to be used.

This example demonstrates two MX data types that you will encounter all of the time in MX programming. The first data type is the **MX_RECORD** structure. **MX_RECORD**s are the most important object type in MX and for the most part have a 1 to 1 relationship to the actual hardware being controlled by the experiment. Thus, an area detector will have an **MX_RECORD** that represents it, and the imaging board that the area detector communicates with will have an **MX_RECORD** that represents it, and so forth.

The **MX_RECORD** structure is designed to be use as a mostly opaque type. The only member of it that you are likely to use is the **name** value. For example, you could print out the name of the *mx_database* record in the example above with code like this:

```
...

printf("The name of the MX database record is '%s'.\n", mx_database->name);

...
```

The output of this line should look like this:

```
The name of the MX database record is 'mx_database'.
```

The MX in-memory database is maintained as a circular linked list of **MX_RECORD** structures. The **MX_RECORD** returned by *mx_setup_database()* is referred to as the "list head" record and is always named "mx_database". The list head record maintains information about the MX database as a whole.

The **MX_RECORD** data type is defined near the top of the header file **$MXDIR/include/mx_record.h**. Do not be alarmed by the complexity of the data structure you find there. You should not have to know about any of it unless you plan to write your own new MX drivers.

The other important data type is the **mx_status_type**. Most, but not all, MX functions have an **mx_status_type** structure as the value returned by the function. The **mx_status_type** structure is a fairly simple structure that has only three element. It is defined as follows:

```
typedef struct {
    long code;              /* The error code. */
    const char *location;   /* Function name where the error occurred. */
    char *message;          /* The specific error message. */
} mx_status_type;
```

Most MX functions will return to their caller either with a line like this

```
return MX_SUCCESSFUL_RESULT;
```

if the function was successful, or they will do something like this

```
...
static const char fname[] = "mx_test_function()";
...
return mx_error( MXE_PERMISSION_DENIED, fname,
    "You do not have sufficient privilege to perform this test." );
...
```

In the latter case, by default the MX program will send to *stderr* a message that looks like this

```
MXE_PERMISSION_DENIED in mx_test_function():
-> You do not have sufficient privilege to perform this test.
```

It is possible to redirect or suppress such messages using the functions described below in Section 2.2.4.
    Note that, by convention, most MX functions will start with a definition of the form

```
static const char fname[] = "mx_function_name()";
```

The C99 standard has now added an equivalent feature to this. However, most of our supported platforms do not yet support the C99 version of the C standard, so we must continue to add our own function name strings for the forseeable future.

2.2.2   mx_setup_database_from_array()

mx_setup_database_from_array() is an alternate to *mx_setup_database()* that reads in the records from an array of character strings instead of a disk file. This function may be used as follows:

```
....
#define NUM_RECORDS    2
....
MX_RECORD *mx_database;
char db_array[NUM_RECORDS][80];
char server_name[] = "192.168.137.3";
long server_port   = 9727;
mx_status_type mx_status;

snprintf( db_array[0], sizeof(db_array[0]),
        "adserver server network tcp_server \"\" \"\" 0x20000000 %s %ld",
                                server_name, server_port );
```

```
strlcpy( db_array[1],
  "ad device area_detector network_area_detector \"\" \"\" 8 adserver ad",
                sizeof(db_array[1]) );

mx_status = mx_setup_database_from_array( &mx_database,
                                          NUM_RECORDS, db_array );

if ( mx_status.code != MXE_SUCCESS )
        return mx_status;
....
```

2.2.3 mx_get_record()

In order to use a device controlled by MX, you must get a pointer to the **MX_RECORD** object describing it. This operation is performed by the function *mx_get_record()*. Here is an example of using *mx_get_record()*:

```
...
MX_RECORD *mx_database;
MX_RECORD *ad_record;
char ad_record_name[] = "ad";

ad_record = mx_get_record( mx_database, ad_record_name );

if ( ad_record == NULL ) {
    return mx_error( MXE_NOT_FOUND, fname,
    "The record '%s' was not found in the running MX database.",
            ad_record_name );
}
...
```

mx_get_record() is a fairly simple function. You hand it a pointer to the MX database and the name of the record you want to find and it either returns a pointer to the new record or a NULL pointer if the record was not found. Note that the record name you specify **must** match the name in your client side database file. Otherwise, the record will not be found.

2.2.4 Redirecting Output

MX has four different classes of output that it sends to the user. These include *debug*, *info*, *warning*, and *error* messages. By default, all of these messages are sent to the *stderr* stream. MX provides four functions that can potentially redirect each of these types of output to a different location. These are: *mx_set_debug_output_function()*, *mx_set_info_output_function()*, *mx_set_warning_output_function()*, and *mx_set_error_output_function()*.

Each one of these functions takes a single argument which is a pointer to a function that takes a *char ** as its single argument and which returns void. For example, if we define the following function

```
static void my_info_output( char *string )
{
```

```
        printf( "MY_INFO_OUTPUT: %s\n", string );
    }
```

and then make the call

```
    mx_set_info_output_function( my_info_output_function );
```

then subsequent calls like this

```
    mx_info("This is a test.");
```

will generate output like this

```
    MY_INFO_OUTPUT: This is a test.
```

2.2.5   Example for mx_setup_database()

Here we show a complete example that loads an MX database, gets a pointer to the area detector record, and finishes by checking that it is indeed an *network_area_detector* record.

```
/*
 * Name:      example1a.c
 *
 * Purpose: This program demonstrates how to initialize MX
 *          using a database file.
 *
 * Author:   William Lavender
 *
 *-------------------------------------------------------------------------
 *
 * Copyright 2006 Illinois Institute of Technology
 *
 * See the file "LICENSE" for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL WARRANTIES.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include "mx_util.h"
#include "mx_record.h"
#include "mx_driver.h"

int
main( int argc, char *argv[] )
{
        MX_RECORD *mx_database, *area_detector_record;
        char mx_database_name[] = "./mx_client.dat";
```

```
        char area_detector_name[] = "ad";
        mx_status_type mx_status;

        mx_status = mx_setup_database( &mx_database, mx_database_name );

        if ( mx_status.code != MXE_SUCCESS ) {
            fprintf( stderr, "Did not successfully open MX database '%s'.\n",
                mx_database_name );
            exit(1);
        }

        area_detector_record = mx_get_record( mx_database, area_detector_name );

        if ( area_detector_record == NULL ) {
            fprintf( stderr, "Did not find record '%s' in MX database '%s'.\n",
                area_detector_name, mx_database_name );
            exit(1);
        }

        if ( area_detector_record->mx_type != MXT_AD_NETWORK ) {
            fprintf( stderr,
                "MX record '%s' is not a network area detector record.\n",
                    area_detector_record->name );
            fprintf( stderr, "Instead, it is of type '%s'.\n",
                    mx_get_driver_name( area_detector_record ) );
            exit(1);
        }

        printf( "Successfully found network area detector '%s'.\n",
            area_detector_record->name );

        exit(0);
}
```

This example introduces a few new features. First are the MX include file definitions

```
    #include "mx_util.h"
    #include "mx_record.h"
    #include "mx_driver.h"
```

The first include file **$MXDIR/include/mx_util.h** contains a large number of utility definitions and functions. In particular, it is the header file that defines **mx_status_type**. The second header file **$MXDIR/include/mx_record.h** defines the **MX_RECORD** structure as well as a variety of other structures and generic functions to manipulate records. All of the MX functions in the above example are defined in **mx_record.h**. The last include file **$MXDIR/include/mx_driver.h** contains a list of the numerical driver type codes and is included in this program in order to get the definition of the MXT_AD_NETWORK macro.

Two other additions of note are the the use of the **mx_type** field of the **MX_RECORD** structure in the statement

```
            if ( area_detector_record->mx_type != MXT_AD_NETWORK ) {
```

and the use of the function *mx_get_driver_name()* which returns the character string name of the driver for this record. It will always be the same text which appears in the fourth field of the MX record definition in the MX database file.

2.2.6   Example for **mx_setup_database_from_array()**

Here is the same example as in the previous section rewritten to use *mx_setup_database_from_array()*:

```
/*
 * Name:     example1b.c
 *
 * Purpose: This program demonstrates how to initialize MX
 *          using a database stored in an array.
 *
 * Author:  William Lavender
 *
 *--------------------------------------------------------------------------
 *
 * Copyright 2006 Illinois Institute of Technology
 *
 * See the file "LICENSE" for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL WARRANTIES.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include "mx_util.h"
#include "mx_record.h"
#include "mx_driver.h"

#define NUM_RECORDS 2
#define LINE_LENGTH 80

int
main( int argc, char *argv[] )
{
        MX_RECORD *mx_database, *area_detector_record;
        char server_host[] = "192.168.137.3";
        long server_port = 9827;
        char area_detector_name[] = "ad";
        char **db_array;
        mx_status_type mx_status;

        db_array = malloc( 2 * sizeof(char *) );
```

```
if ( db_array == NULL ) {
    fprintf( stderr,
        "Could not allocate row pointer for db_array.\n" );
    exit(1);
}

db_array[0] = malloc( LINE_LENGTH * sizeof(char) );

if ( db_array[0] == NULL ) {
    fprintf( stderr,
        "Could not allocate row 0 of db_array.\n" );
    exit(1);
}

db_array[1] = malloc( LINE_LENGTH * sizeof(char) );

if ( db_array[1] == NULL ) {
    fprintf( stderr,
        "Could not allocate row 1 of db_array.\n" );
    exit(1);
}

snprintf( db_array[0], LINE_LENGTH,
    "adserver server network tcp_server \"\" \"\" 0x0 %s %ld",
        server_host, server_port );

snprintf( db_array[1], LINE_LENGTH,
"%s device area_detector network_area_detector \"\" \"\" 8 adserver ad",
    area_detector_name );

mx_status = mx_setup_database_from_array( &mx_database,
                                    NUM_RECORDS, db_array );

if ( mx_status.code != MXE_SUCCESS ) {
    fprintf( stderr, "Did not successfully open MX database array.\n");
    exit(1);
}

area_detector_record = mx_get_record( mx_database, area_detector_name );

if ( area_detector_record == NULL ) {
    fprintf( stderr, "Did not find record '%s' in the MX database.\n",
        area_detector_name );
    exit(1);
}
```

```
        if ( area_detector_record->mx_type != MXT_AD_NETWORK ) {
            fprintf( stderr,
                "MX record '%s' is not a network area detector record.\n",
                    area_detector_record->name );
            fprintf( stderr, "Instead, it is of type '%s'.\n",
                    mx_get_driver_name( area_detector_record ) );
            exit(1);
        }

        printf( "Successfully found network area detector '%s'.\n",
            area_detector_record->name );

        exit(0);
}
```

2.3   Reading and Writing Area Detector Settings

MX has a variety of functions for reading and writing internal area detector settings. The following list summarizes
the functions available of this type.

mx_area_detector_get_maximum_framesize()
     This function reports the resolution that the detector has if it is in unbinned mode.

mx_area_detector_get_framesize()
     This function reports the current resolution of the detector.

mx_area_detector_set_framesize()
     This function sets the the resolution of the detector to the nearest size that is actually supported by the detector.

mx_area_detector_get_image_format()
     Returns a constant that describes the image format.

mx_area_detector_get_bytes_per_pixel()
     Returns the number of bytes that correspond to a pixel. There exist image formats for which this ratio is not an
     integer, so *mx_area_detector_get_bytes_per_pixel()* reports back the bytes per pixel as a C *double* rather than as
     an integer.

mx_area_detector_get_bytes_per_frame()
     Returns the number of bytes in an image frame using the current image format and the current framesize.

mx_area_detector_get_trigger_mode()
     Reports whether the area detector is using internal or external triggering.

mx_area_detector_set_trigger_mode()
     Specifies whether the area detector should use internal or external triggering.

2.4 Detector Properties

Each kind of detector has a variety of internal parameters that are specific to that model. Those internal parameters, which MX calls *properties* are read and written using the following set of property functions. Each property is referred to by an ASCII name. The following functions are provided for reading and writing the properties.

mx_area_detector_get_property_double()
mx_area_detector_set_property_double()
These two functions read and write property values as doubles.

mx_area_detector_get_property_long()
mx_area_detector_set_property_long()
These two functions read and write property values as longs.

mx_area_detector_get_property_string()
mx_area_detector_set_property_string()
These two functions read and write property values as strings.

2.5 Sequences

For an area detector, a *sequence* is a series of one or more image frames that are taken after a single trigger is sent to the area detector. In the most general case, each frame can have a different exposure time and each pair of frames can be separated by a different separation time.

In MX, the instructions for taking a sequence are encoded in an **MX_SEQUENCE_PARAMETERS** structure. This structure is found in the MX header file **$MXDIR/include/mx_image.h** and is defined like this:

```
typedef struct {
        long sequence_type;
        long num_parameters;
        double parameter_array[MXU_MAX_SEQUENCE_PARAMETERS];
} MX_SEQUENCE_PARAMETERS;
```

The *sequence_type* structure member specifies the type of sequence to be executed. The *parameter_array* member contains all of the parameters used by the particular type of sequence, while *num_parameters* tells you how many parameters are in the array. Bear in mind that not all types of sequences supported by MX are available for all types of area detectors.

The *mx_area_detector_set_sequence_parameters()* function can be used to specify the parameters for the next sequence to be run and *mx_area_detector_get_sequence_parameters()* can be used to report the current sequence parameter settings. Here is an example of how to use *mx_area_detector_set_sequence_parameters()* for One-shot mode, which is used to take one frame and then stop:

```
...
MX_RECORD *ad_record;
MX_SEQUENCE_PARAMETERS seq_params;
...
seq_params.sequence_type = MXT_SQ_ONE_SHOT;
seq_params.num_parameters = 1;
seq_params.parameter_array[0] = 0.5;  /* Exposure time of 0.5 seconds. */
```

```
    mx_status = mx_area_detector_set_sequence_parameters( ad_record,
                                                  &seq_params );
if ( mx_status.code != MXE_SUCCESS )
    return mx_status;
....
```

However, each sequence type has a wrapper function that can be used as a simplified interface to that type of sequence. Thus, for the one-shot case above, it would be more common to use this method instead:

```
...
MX_RECORD *ad_record;
...
/* Request a single frame with a 0.5 second exposure time. */

mx_status = mx_area_detector_set_one_shot_mode( ad_record, 0.5 );

if ( mx_status.code != MXE_SUCCESS )
    return mx_status;
....
```

A variety of convenience functions have been defined for setting up particular kinds of sequences:

mx_area_detector_set_one_shot_mode()

    A one-shot sequence takes a single frame for the requested exposure time and then stops.

mx_area_detector_set_continuous_mode()

    A continuous sequence repeatedly takes frames for the requested exposure time, without any explicit gaps between the frames until commanded to stop. Each new frame overwrites the previous frame. This mode is useful for live display of images from the detector.

mx_area_detector_set_multiframe_mode()

    This type of sequence takes a specified number of frames with each frame going into a separate frame buffer. All of the frames have the same exposure times and the gaps between the frames are all the same as well.

mx_area_detector_set_circular_multiframe_mode()

    This mode is almost identical to the multiframe mode. However, after the detector takes the last requested frame, it loops back to the start of the frame sequence and starts overwriting old frames.

mx_area_detector_set_strobe_mode()

    This type of sequence takes a specified number of frames with each frame going into a separate frame buffer. The start of exposure for each frame is triggered by an external trigger. This mode can be thought of as edge-triggered. Each frame has the same exposure time.

mx_area_detector_set_bulb_mode()

    This type of sequence takes a specified number of frames with each frame going into a separate frame buffer. For each frame, the external trigger signal controls the start and the end of the exposure time for each frame. The exposure starts when the external trigger goes high and ends when the external trigger goes low. This mode can be thought of as level-triggered.

mx_area_detector_set_geometrical_mode()

>   This type of sequence is specific to the PCCD-170170 area detector. In this mode, the first frame and first gap are taken for the specified exposure and gap times. For each subsequent frame, the exposure time and gap times are multiplied by another factor of the exposure and gap multipliers.

Please note that not all types of sequences are available for all types of detectors.

2.6 Actions

The functions in this section command the area detector to start a data acquisition sequence and then report on the status of this sequence.

mx_area_detector_arm()

>   Tells the area detector to perform all the preliminary setup required to be ready to start an image sequence. If external trigger is enabled, the first trigger after the arm finishes will start the imaging sequence.

mx_area_detector_trigger()

>   Sends a software trigger to the area detector that tells it to start the imaging sequence.

mx_area_detector_start()

>   This is a utility function that invokes *mx_area_detector_arm()* followed by *mx_area_detector_trigger()*.

mx_area_detector_stop()

>   This tells the area detector to stop an in-progress imaging sequence after the current frame completes.

mx_area_detector_abort()

>   This tells the area detector to stop an in-progress imaging sequence as quickly as possible. In general, the frame being taken at the time of the abort will be lost.

mx_area_detector_get_last_frame_number()

>   Reports the frame number of the most recently acquired frame. Before the first frame (frame 0) is taken, it will return the value -1.

mx_area_detector_get_status()

>   Returns a status flags value that describes the current state of the area detector. The long integer returned is a bitmap in which each bit stands for a different status flag.

mx_area_detector_get_extended_status()

>   This function returns both the last frame number and the status flags.

mx_area_detector_is_busy()

>   This function reads out the status flag from the detector and returns the value of the *busy* bit (MXSF_AD_IS_BUSY).

2.7 Frame Transfer

mx_area_detector_setup_frame()

>   This function creates a local **MX_IMAGE_FRAME** structure that can be used to hold a frame from the area detector.

mx_area_detector_readout_frame()

    This function tells the area detector to readout the contents of the specified frame number into the image buffer of the MX server.

mx_area_detector_transfer_frame()

    This function transfers the contents of one of the frame buffers described in Section 3.1.2 into the frame buffer that was configured by *mx_area_detector_setup_frame()*.

mx_area_detector_load_frame()

    This function loads an image frame from the specified file into one of the frame buffers described in Section 3.1.2. The filename specified must refer to a file found on the detector computer itself and must use the image format configured for this area detector.

mx_area_detector_save_frame()

    This function saves one of the frame buffers described in Section 3.1.2 to the specified file. The filename specified must refer to a file found on the detector computer itself and will use the image format configured for this area detector.

mx_area_detector_save_frame()

    This function copies an image frame from one of the frame buffers described in Section 3.1.2 to another of the frame buffers.

Utility Functions

mx_area_detector_get_frame()

    This is a utility function that returns the contents of the specified image frame number to the specified **MX_IMAGE_FRAME** structure. This function invokes the following functions in order: *mx_area_detector_setup_frame()* followed by *mx_area_detector_readout_frame()* followed by *mx_area_detector_correct_frame()* and finishing with *mx_area_detector_transfer_fr*

mx_area_detector_get_sequence()

    This is a utility function that fills in the frames in an **MX_IMAGE_SEQUENCE** structure by repeated calls to *mx_area_detector_get_frame()*.

mx_area_detector_get_frame_from_sequence()

    This is a utility function that returns a pointer to the specified frame number in the specified **MX_IMAGE_SEQUENCE** structure. It is expected that the contents of the **MX_IMAGE_SEQUENCE** structure will already have been filled in by a previous call to a function like *mx_area_detector_get_sequence()*.

2.8 Frame Correction

A hexadecimal *correction_flags* field is used for determining which corrections are made. The bits in the correction flags use the bit definitions made in Section 3.1.2.

mx_area_detector_get_correction_flags()

    Gets the current correction flag settings.

mx_area_detector_set_correction_flags()

    Sets the correction flags.

mx_area_detector_correct_frame()
> This function causes all of the requested corrections to be performed on the frame in the image frame buffer of the detector computer. Normal MX client programs should use this function rather than the following function.

mx_area_detector_frame_correction()
> This function supplies pointers to the image frames to be used for image corrections. Since these frames are normally located on the detector computer, this function should normally only be invoked on the detector computer.

mx_area_detector_measure_correction_frame()
> This function can be used to create new dark current and flood field frames.

mx_area_detector_measure_dark_current_frame()
> This is a macro wrapper for *mx_area_detector_measure_correction_frame()* that is only used for dark current measurements.

mx_area_detector_measure_flood_field_frame()
> This is a macro wrapper for *mx_area_detector_measure_correction_frame()* that is only used for flood field measurements.

2.9 Region of Interest (ROI) Functions

mx_area_detector_get_roi()
> Reports the X and Y dimensions in binned coordinates for the requested ROI number.

mx_area_detector_set_roi()
> Sets the X and Y dimensions in binned coordinates for the requested ROI number.

mx_area_detector_get_roi_frame()
> This function reads out the contents of the requested ROI number from the supplied original **MX_IMAGE_FRAME** structure to another **MX_IMAGE_FRAME** structure that is only large enough to hold the ROI contents.

mx_area_detector_get_subframe_size()
> For the AVIEX PCCD-170170 detector, this reads out the number of columns in the subframe.

mx_area_detector_set_subframe_size()
> This configures the AVIEX PCCD-170170 detector to only readout a restricted number of columns from the center of a detector element. When used, it disables the operation of the normal ROIs and returns the data in ROI 0.

2.10 Image Functions

MX comes with a number of functions that operate directly on **MX_IMAGE_FRAME** structures and which do not involve the area detector at all. These functions are defined in the header file **$MXDIR/include/mx_image.h**.

mx_image_alloc()
> This function creates a new **MX_IMAGE_FRAME** structure. You must specify the image type, the frame size, the image format, the pixel order, the bytes per pixel, the header length, and the image length. If you

want an **MX IMAGE FRAME** structure that is compatible with your area detector, you are better off letting *mx_area_detector_setup_frame()* or *mx_area_detector_get_frame()* do it for you, instead of invoking this function directly.

mx image free()

This function frees all of the memory in use by an existing **MX IMAGE FRAME** structure.

mx image copy frame()

This copies the contents of one **MX IMAGE FRAME** to another.

mx image get exposure time()

This returns the exposure time in seconds of the exposure that was used to take the original image data. This information is generally used to perform scaled dark current corrections. If the image has been read from an image file whose header does not contain the exposure time, the exposure time is set to 1.

mx image get image data pointer()

This returns a pointer to a 1-dimensional buffer containing the image data contained in an **MX IMAGE FRAME** structure. If the image size or format of a new frame read into an existing **MX IMAGE FRAME** structure by *mx_area_detector_get_frame* or *mx_area_detector_transfer_frame* is different than that of the image data that was already in the structure, then MX may replace the existing image data buffer with a new one. If you need a pointer to the 1-dimensional image data buffer, it is safest to reinvoke *mx_image_get_image_data_pointer()* after each new frame is read into the **MX IMAGE FRAME** structure.

mx image copy 1d pixel array()

This function copies the pixel data from an **MX IMAGE FRAME** structure to a 1-dimensional buffer supplied by the caller.

mx image read file()

This function reads the contents of the requested image file in the requested format into the supplied **MX IMAGE FRAME** structure. The specified filename must exist on the computer that is invoking this function.

mx image write file()

This function writes the contents of the supplied **MX IMAGE FRAME** structure in the requested format into the requested image file. The file created will be on the computer that is invoking this function.

mx image get format type from name()

mx image get format name from type()

These are two utility functions that convert back and forth between ASCII names for the formats such as *GREY16* and the corresponding numerical format values.

2.11 Example Programs

Having shown an overview of the API used to control area detectors with MX, we now show some complete programs that use the API.

2.11.1 Acquiring and Saving Images - example2.c

This program takes a sequence of image frames from an area detector. The program demonstrates both saving them to a file on the detector computer and transferring them to the client and then saving them on the client computer.

```
/*
 * Name:      example2.c
 *
 * Purpose: This program demonstrates running a multiple frame sequence.
 *           The program has one command line argument 'use_client_disk'
 *           which if set to a non-zero value will make the client program
 *           transfer the frames to its local disk and save them there.
 *
 * Author:  William Lavender
 *
 *-----------------------------------------------------------------------------
 *
 * Copyright 2006 Illinois Institute of Technology
 *
 * See the file "LICENSE" for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL WARRANTIES.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include "mx_util.h"
#include "mx_record.h"
#include "mx_image.h"
#include "mx_area_detector.h"

static mx_status_type
save_frame_on_detector_computer( MX_RECORD *ad_record,
                                 long last_frame_number );

static mx_status_type
save_frame_on_client_computer( MX_RECORD *ad_record,
                               MX_IMAGE_FRAME **image_frame,
                               long last_frame_number );

int
main( int argc, char *argv[] )
{
        MX_RECORD *mx_database, *ad_record;
        MX_IMAGE_FRAME *image_frame;
        char mx_database_name[] = "./mx_client.dat";
```

```
        char area_detector_name[] = "ad";
        double exposure_time, gap_time;
        long num_frames, last_frame_number;
        long current_frame_number;
        unsigned long ad_status_flags;
        int use_client_disk;
        mx_status_type mx_status;

        if ( argc != 2 ) {
                fprintf(stderr, "Usage: %s 'use_client_disk_flag'\n", argv[0]);
                exit(1);
        }

        image_frame = NULL;

        /* Do we save the frames on the client computer or do we save
         * the frames on the detector computer?
         */

        use_client_disk = atoi( argv[1] );

        /* Initialize the MX database. */

        mx_status = mx_setup_database( &mx_database, mx_database_name );

        if ( mx_status.code != MXE_SUCCESS ) {
                fprintf( stderr,
                        "Did not successfully initialize MX database '%s'.\n",
                        mx_database_name );
                exit(1);
        }

        /* Find the area detector record in the database. */

        ad_record = mx_get_record( mx_database, area_detector_name );

        if ( ad_record == NULL ) {
                fprintf( stderr,
                "Did not find area detector record '%s' in MX database '%s'.\n",
                        area_detector_name, mx_database_name );
                exit(1);
        }

        /* Configure a sequence for the area detector that tells it to
         * take 10 image frames with an exposure time of 1.0 second
         * for each frame and a gap of 0.5 seconds between the frames.
```

```
 */

num_frames    = 10;
exposure_time = 1.0;
gap_time      = 0.5;

fprintf( stderr,
"Taking a multiframe sequence of %ld frames using area detector '%s' with "
"an exposure time of %g seconds and a gap time of %g seconds.\n",
            num_frames, ad_record->name,
            exposure_time, gap_time );

mx_status = mx_area_detector_set_multiframe_mode( ad_record,
                                            num_frames,
                                            exposure_time,
                                            gap_time );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* Arm the detector. */

fprintf( stderr, "Arming the detector.\n" );

mx_status = mx_area_detector_arm( ad_record );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* We assume for this example that the area detector is configured
 * for internal trigger mode.  If so, then we will need to explicitly
 * trigger the data acquisition sequence.  If we were using an
 * external trigger, then this step would not be necessary.
 */

fprintf( stderr, "Triggering the detector.\n" );

mx_status = mx_area_detector_trigger( ad_record );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* Monitor the progress of the data acquisition sequence. */

current_frame_number = -1;
```

```
while (1) {
        /* Ask for the status of the detector. */

        mx_status = mx_area_detector_get_extended_status( ad_record,
                                        &last_frame_number,
                                        &ad_status_flags );

        if ( mx_status.code != MXE_SUCCESS )
                exit(1);

        /* If the reported last frame number has changed, then there
         * are one or more frames available to be read out.
         */

        if ( last_frame_number != current_frame_number ) {

                current_frame_number++;

                if ( use_client_disk ) {
                        mx_status = save_frame_on_client_computer(
                                ad_record, &image_frame,
                                current_frame_number );
                } else {
                        mx_status = save_frame_on_detector_computer(
                                ad_record, current_frame_number );
                }

                if ( mx_status.code != MXE_SUCCESS )
                        exit(1);
        }

        /* If the status flags say that the area detector is
         * no longer busy, see if there are any frames left
         * to read out.
         */

        if ( ( ad_status_flags & MXSF_AD_IS_BUSY ) == 0 ) {

                if ( current_frame_number >= last_frame_number ) {

                        /* If not, then exit. */

                        fprintf( stderr,
                        "The data acquisition sequence has completed.\n" );
                        fprintf( stderr, "Exiting now...\n" );
```

```
                                exit(0);
                        }
                }

                /* Sleep for a millisecond so that we do not use up
                 * all of the CPU time on the computer.
                 */

                mx_msleep(1);
        }

        fprintf( stderr, "Should never get here.\n" );

        exit(1);
}

static mx_status_type
save_frame_on_detector_computer( MX_RECORD *ad_record,
                                long last_frame_number )
{
        char savefile_name[80];
        mx_status_type mx_status;

        fprintf( stderr,
    "Saving area detector '%s' frame number %ld on the detector computer.\n",
                ad_record->name, last_frame_number );

        /* Tell the detector computer to readout the requested frame from
         * the area detector hardware into the detector computer's primary
         * image frame buffer.
         */

        mx_status = mx_area_detector_readout_frame( ad_record,
                                                last_frame_number );

        if ( mx_status.code != MXE_SUCCESS )
                return mx_status;

        /* Tell the detector computer to correct the frame that was
         * just read out.
         */

        mx_status = mx_area_detector_correct_frame( ad_record );

        if ( mx_status.code != MXE_SUCCESS )
                return mx_status;
```

```
        /* Tell the detector computer to save the frame on its own
         * hard disk.
         *
         * If we specify a full pathname, the MX server will save
         * the file at the requested location.
         *
         * If we give the server only the filename, it will save the
         * file in the default save file directory on the detector
         * computer.
         *
         * If we give it an empty or NULL filename, the MX server
         * will choose the next filename on its own.
         */

        snprintf( savefile_name, sizeof(savefile_name),
                "example2_%04ld.pnm", last_frame_number );

        mx_status = mx_area_detector_save_frame( ad_record,
                                                MXFT_AD_IMAGE_FRAME,
                                                savefile_name );

        if ( mx_status.code != MXE_SUCCESS )
                return mx_status;

        fprintf(stderr, "Successfully wrote image file '%s'.\n", savefile_name);

        return MX_SUCCESSFUL_RESULT;
}

static mx_status_type
save_frame_on_client_computer( MX_RECORD *ad_record,
                               MX_IMAGE_FRAME **image_frame,
                               long last_frame_number )
{
        MX_IMAGE_FRAME *local_image_frame;
        char savefile_name[80];
        size_t image_length;
        void *image_data_pointer;
        uint16_t *uint16_array;
        int i;
        mx_status_type mx_status;

        fprintf( stderr,
    "Saving area detector '%s' frame number %ld on the client computer.\n",
                ad_record->name, last_frame_number );
```

```
/* The first time that mx_area_detector_setup_frame() is invoked,
 * it will allocate memory for the image frame data structures.
 * On subsequent calls, it checks to see if the already allocated
 * image frame data structures are too small to hold the new
 * image frame.  If they are already big enough, the image frame
 * is left alone.
 *
 * In addition, mx_area_detector_setup_frame() saves a pointer
 * to the image frame in the area detector record data structure.
 * Later in this routine, mx_area_detector_transfer_frame() will
 * read the image sent by the detector computer into that frame.
 */

mx_status = mx_area_detector_setup_frame( ad_record, image_frame );

if ( mx_status.code != MXE_SUCCESS )
        return mx_status;

/* Tell the detector computer to readout the requested frame from
 * the area detector hardware into the detector computer's primary
 * image frame buffer.
 */

mx_status = mx_area_detector_readout_frame( ad_record,
                                            last_frame_number );

if ( mx_status.code != MXE_SUCCESS )
        return mx_status;

/* Tell the detector computer to correct the frame that was
 * just read out.
 */

mx_status = mx_area_detector_correct_frame( ad_record );

if ( mx_status.code != MXE_SUCCESS )
        return mx_status;

/* Tell the detector computer to send the primary image frame
 * across the network to the client computer.  If this call
 * completes successfully, the image will be accessible via
 * the *image_frame pointer.
 */

mx_status = mx_area_detector_transfer_frame( ad_record,
```

```
                                                    MXFT_AD_IMAGE_FRAME );

        if ( mx_status.code != MXE_SUCCESS )
                return mx_status;

        local_image_frame = *image_frame;

        /* Display the first 10 pixels in the frame.  You should always
         * reinvoke mx_image_get_image_data_pointer() after each time
         * you read in a frame.
         */

        mx_status = mx_image_get_image_data_pointer( local_image_frame,
                                                &image_length,
                                                &image_data_pointer );

        if ( mx_status.code != MXE_SUCCESS )
                return mx_status;

        fprintf( stderr, "The transferred image is %lu bytes long.\n",
                        (unsigned long) image_length );

        uint16_array = image_data_pointer;

        for ( i = 0; i < 10; i++ ) {
                fprintf( stderr, "image[%d] = %hu\n",
                        i, (unsigned short) uint16_array[i] );
        }

        /* We finish by writing the local image to a disk file in PNM format.
         *
         * FIXME: For the real release, we need to convert this to SMV format.
         */

        snprintf( savefile_name, sizeof(savefile_name),
                "example2_%04ld.pnm", last_frame_number );

        mx_status = mx_image_write_file( local_image_frame,
                                        MXT_IMAGE_FILE_PNM,
                                        savefile_name );

        if ( mx_status.code != MXE_SUCCESS )
                return mx_status;

        fprintf(stderr, "Successfully wrote image file '%s'.\n", savefile_name);
```

```
              return MX_SUCCESSFUL_RESULT;
}
```

2.11.2 Measuring Detector Dark Currents - example3.c

```
/*
 * Name:    example3.c
 *
 * Purpose: This program shows how to create a new dark current image frame
 *          for an area detector.
 *
 * Author:  William Lavender
 *
 *-------------------------------------------------------------------------
 *
 * Copyright 2006 Illinois Institute of Technology
 *
 * See the file "LICENSE" for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL WARRANTIES.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include "mx_util.h"
#include "mx_record.h"
#include "mx_image.h"
#include "mx_area_detector.h"

int
main( int argc, char *argv[] )
{
        MX_RECORD *mx_database, *ad_record;
        MX_IMAGE_FRAME *image_frame;
        char mx_database_name[] = "./mx_client.dat";
        char area_detector_name[] = "ad";
        double exposure_time;
        long num_exposures;
        char *savefile_name;
        mx_bool_type busy;
        mx_status_type mx_status;

        if ( argc != 4 ) {
                fprintf(stderr,
```

```
                "Usage: %s 'exposure time' 'num exposures' 'savefile name'\n",
                            argv[0]);
            exit(1);
    }

    image_frame = NULL;

    exposure_time = atof( argv[1] );
    num_exposures = atoi( argv[2] );
    savefile_name = argv[3];

    /* Initialize the MX database. */

    mx_status = mx_setup_database( &mx_database, mx_database_name );

    if ( mx_status.code != MXE_SUCCESS ) {
            fprintf( stderr,
                    "Did not successfully initialize MX database '%s'.\n",
                    mx_database_name );
            exit(1);
    }

    /* Find the area detector record in the database. */

    ad_record = mx_get_record( mx_database, area_detector_name );

    if ( ad_record == NULL ) {
            fprintf( stderr,
            "Did not find area detector record '%s' in MX database '%s'.\n",
                    area_detector_name, mx_database_name );
            exit(1);
    }

    /* Tell the area detector to start acquiring images with which
     * to make a dark current correction frame.
     *
     * There should be no photons hitting the detector while this
     * measurement is in progress.
     */

    if ( num_exposures == 1 ) {
            fprintf( stderr,
            "Starting the dark current measurement using "
            "1 exposure of %g seconds.\n",
                    exposure_time );
    } else {
```

```
            fprintf( stderr,
            "Starting the dark current measurement using "
            "%ld exposures of %g seconds each.\n",
                    num_exposures, exposure_time );
    }

    mx_status = mx_area_detector_measure_dark_current_frame( ad_record,
                                        exposure_time, num_exposures );

    if ( mx_status.code != MXE_SUCCESS ) {
            fprintf( stderr,
            "Unable to start the dark current measurement.\n" );
            exit(1);
    }

    /* Wait for the dark current measurement to complete. */

    while (1) {
            /* See if the measurement is still in progress. */

            mx_status = mx_area_detector_is_busy( ad_record, &busy );

            if ( mx_status.code != MXE_SUCCESS ) {
                    fprintf( stderr,
                "An attempt to check the status of the detector failed.\n");
                    exit(1);
            }

            if ( busy == FALSE ) {
                    /* The measurement is complete, so we can now
                     * break out of the while() loop.
                     */

                    break;
            }

            /* Sleep for a millisecond so that we do not use up
             * all of the CPU time on the computer.
             */

            mx_msleep(1);
    }

    fprintf( stderr, "The dark current measurement is complete.\n" );

    /* The new dark current frame will already be in the correct
```

```
         * image buffer on the detector computer for the purpose of
         * automatic image correction.  However, we will also save a
         * copy of the new dark current frame to disk.  Please note
         * that the filename for the save file refers to the disk
         * on the detector computer.
         */

        mx_status = mx_area_detector_save_frame( ad_record,
                                                 MXFT_AD_DARK_CURRENT_FRAME,
                                                 savefile_name );

        if ( mx_status.code != MXE_SUCCESS ) {
                fprintf( stderr,
                "The attempt to save the new dark current frame\n"
                "to the file '%s' on the detector computer failed.\n",
                        savefile_name );
                exit(1);
        }

        fprintf( stderr,
        "The dark current frame was successfully saved to the file '%s' "
        "on the detector computer.\n",
                savefile_name );

        fprintf( stderr, "Program complete.\n" );

        exit(0);
}
```

2.11.3 Reading Out a Region Of Interest (ROI) - example4.c

```
/*
 * Name:    example4.c
 *
 * Purpose: This program acquires a single image frame, reads out a region
 *          of interest and then writes the ROI to a disk file on the client.
 *
 * Note:    If programmed in this manner, only the pixels contained within
 *          the ROI will be transferred across the network.
 *
 * Author:  William Lavender
 *
 *-----------------------------------------------------------------------------
 *
 * Copyright 2006 Illinois Institute of Technology
```

```
 *
 * See the file "LICENSE" for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL WARRANTIES.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include "mx_util.h"
#include "mx_record.h"
#include "mx_image.h"
#include "mx_area_detector.h"

int
main( int argc, char *argv[] )
{
        MX_RECORD *mx_database, *ad_record;
        MX_IMAGE_FRAME *roi_frame;
        void *roi_data_pointer;
        uint16_t *uint16_array;
        char mx_database_name[] = "./mx_client.dat";
        char area_detector_name[] = "ad";
        unsigned long i, roi_number, roi[4];
        size_t roi_length;
        double exposure_time;
        char roi_filename[MXU_FILENAME_LENGTH+1];
        mx_bool_type busy;
        mx_status_type mx_status;

        /* Initialize the MX database. */

        mx_status = mx_setup_database( &mx_database, mx_database_name );

        if ( mx_status.code != MXE_SUCCESS ) {
                fprintf( stderr,
                        "Did not successfully initialize MX database '%s'.\n",
                        mx_database_name );
                exit(1);
        }

        /* Find the area detector record in the database. */

        ad_record = mx_get_record( mx_database, area_detector_name );

        if ( ad_record == NULL ) {
```

```
        fprintf( stderr,
        "Did not find area detector record '%s' in MX database '%s'.\n",
                area_detector_name, mx_database_name );
        exit(1);
}

/* Configure the detector to acquire a single frame. */

exposure_time = 0.5;            /* in seconds */

fprintf( stderr,
"Taking a single frame exposure of %g seconds.\n", exposure_time );

mx_status = mx_area_detector_set_one_shot_mode( ad_record,
                                                exposure_time );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* Arm the detector. */

fprintf( stderr, "Arming the detector.\n" );

mx_status = mx_area_detector_arm( ad_record );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* We assume for this example that the area detector is configured
 * for internal trigger mode.  If so, then we will need to explicitly
 * trigger the data acquisition sequence.  If we were using an
 * external trigger, then this step would not be necessary.
 */

fprintf( stderr, "Triggering the detector.\n" );

mx_status = mx_area_detector_trigger( ad_record );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* Monitor the progress of the data acquisition sequence. */

while (1) {
        /* Check to see if the detector is still acquiring a frame. */
```

```
            mx_status = mx_area_detector_is_busy( ad_record, &busy );

            if ( mx_status.code != MXE_SUCCESS )
                    exit(1);

            if ( busy == FALSE ) {

                    /* If the area detector has finished the measurement,
                     * then break out of the while() loop.
                     */

                    break;
            }

            /* Sleep for a millisecond so that we do not use up
             * all of the CPU time on the computer.
             */

            mx_msleep(1);
    }

    fprintf( stderr, "Exposure complete.\nReading out frame 0.\n" );

    /* Tell the detector computer to readout the requested frame from
     * the area detector hardware into the detector computer's primary
     * image frame buffer.
     */

    mx_status = mx_area_detector_readout_frame( ad_record, 0 );

    if ( mx_status.code != MXE_SUCCESS )
            exit(1);

    /* Tell the detector computer to correct the frame that was
     * just read out.
     */

    fprintf( stderr, "Correcting the frame.\n" );

    mx_status = mx_area_detector_correct_frame( ad_record );

    if ( mx_status.code != MXE_SUCCESS )
            exit(1);

    /* Define the boundaries of region of interest 5.  This can be done
     * either before or after acquiring the image frame.
```

```
 */

roi_number = 5;

/* The following limits are specified in binned coordinates. */

roi[0] = 1000;     /* X minimum */
roi[1] = 2000;     /* X maximum */
roi[2] = 250;      /* Y minimum */
roi[3] = 750;      /* Y maximum */

fprintf( stderr,
"Setting ROI %lu to Xmin = %lu, Xmax = %lu, Ymin = %lu, Ymax = %lu\n",
        roi_number, roi[0], roi[1], roi[2], roi[3] );

mx_status = mx_area_detector_set_roi( ad_record, roi_number, roi );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* Make sure that mx_area_detector_get_roi_frame() allocates
 * a new frame by setting roi_frame to NULL.
 */

roi_frame = NULL;

/* Transfer the contents of the detector ROI to the client ROI frame. */

fprintf( stderr, "Reading out ROI %lu\n", roi_number );

mx_status = mx_area_detector_get_roi_frame( ad_record, NULL,
                                            roi_number, &roi_frame );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

/* Display the first 10 pixels in the ROI frame. */

mx_status = mx_image_get_image_data_pointer( roi_frame,
                                             &roi_length,
                                             &roi_data_pointer );

if ( mx_status.code != MXE_SUCCESS )
        exit(1);

fprintf( stderr, "The transferred ROI frame is %lu bytes long.\n",
```

```
                         (unsigned long) roi_length );

        uint16_array = roi_data_pointer;

        for ( i = 0; i < 10; i++ ) {
                fprintf( stderr, "image[%lu] = %hu\n",
                        i, (unsigned short) uint16_array[i] );
        }

        /* We finish by writing the ROI frame to a disk file in PNM format.
         *
         * FIXME: For the real release, we need to convert this to SMV format.
         */

        strlcpy( roi_filename, "roifile.pgm", MXU_FILENAME_LENGTH );

        mx_status = mx_image_write_file( roi_frame,
                                         MXT_IMAGE_FILE_PNM,
                                         roi_filename );

        if ( mx_status.code != MXE_SUCCESS )
                exit(1);

        fprintf(stderr, "Successfully wrote ROI file '%s'.\n", roi_filename );

        exit(0);
}
```

Chapter 3

Area Detector API Reference

3.1 Area Detector Definitions

3.1.1 Area Detector Status Word

The current status of the area detector can be determined by calling either **mx_area_detector_get_status()** or **mx_area_detector_get_exten** both of which return a 32-bit status word. The long term plan is that clients will be able to detect hardware faults in the detector system through this status word. However, at the moment the only status bit in the status word with a defined use is the *Busy* bit. It is defined as follows:

MXSF_AD_IS_BUSY (0x1)

This status bit is set if the detector is currently recording a sequence of image frames. For One-shot, Multiframe, Strobe, and Bulb sequences, the *Busy* bit will turn off after the last frame has been acquired. However, for Continuous and Circular Multiframe modes, the *Busy* bit will continue to be set until the imaging sequence is explicitly stopped or aborted.

3.1.2 Frame Buffer Types

The MX area detector record makes provision for several different frame buffers that are used both for image transfer and image correction operations. These buffers are normally identified using individual bits in a hexadecimal bitmask. The currently defined frame buffers are

MXFT_AD_IMAGE_FRAME (0x1)

This is the primary frame buffer that image frames are initially read into from the area detector hardware.

MXFT_AD_MASK_FRAME (0x2)

The mask frame is optionally used to ignore masked off pixels during the flood field average intensity calculation.

MXFT_AD_BIAS_FRAME (0x4)

CCD detectors typically add a bias offset to pixels returned by the area detector hardware to raise the values above the noise floor and to reduce the likelihood that dark current subtraction will produce a negative value. The bias frame is optionally used to subtract the bias from the frame during image correction. The bias is exposure time independent.

MXFT_AD_DARK_CURRENT_FRAME (0x8)
> The dark current frame is optionally used to subtract an exposure time dependent dark current from the image frame. Normally, the dark current frame should be taken with no photons hitting the imaging surface. Depending on the setting of **mx_area_detector_set_use_scaled_dark_current_flag()**, the dark current frame will either be scaled to match the image frame exposure time, or else the dark current frame will be subtracted as is without any scaling.

MXFT_AD_FLOOD_FIELD_FRAME (0x10)
> The flood field frame is optionally used to perform a flood field (also known as flat field) correction to the image frame. Generally, different pixels in an area detector will return slightly different signals for the same number of incident photons. This can be thought of as a variation in gain for different pixels. The flood field frame, if configured, is used to correct for this variation in gain. Ideally, the flood field frame should be taken with a uniform photon intensity across the entire imaging surface.

Internally, there is an additional image frame buffer, namely the ROI (Region of Interest) frame buffer. In general, this buffer is a different size than the other buffers above and it does not take part in image correction or normal image and file I/O. Instead, there are special ROI-specific functions that operate on the ROI frame buffer. In addition, although the MX area detector class supports multiple ROIs, the ROI frame buffer itself only contains the contents of one ROI at a time.

3.2   mx_area_detector_abort

NAME
> **mx_area_detector_abort** - immediately stop all area detector activity

SYNOPSIS
> mx_status_type **mx_area_detector_abort** (MX_RECORD *record);

DESCRIPTION
> This function tells the area detector to abort all current operations such as image acquisition as quickly as possible. If an imaging sequence is currently in progress, the current image frame may be lost.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_stop()**

3.3   mx_area_detector_arm

NAME
> **mx_area_detector_arm** - prepare the area detector for image acquisition

SYNOPSIS
> mx_status_type **mx_area_detector_arm** (MX_RECORD *record);

DESCRIPTION

This function tells the area detector to perform all operations needed to get ready to be triggered for an imaging sequence. If the area detector has been set by **mx_area_detector_set_trigger_mode()** to a triggering mode that requires an external trigger, the imaging sequence will start when the first external trigger pulse arrives.

If the area detector is in an internal triggering mode, then the application program must invoke **mx_area_detector_trigger()** to start the imaging sequence.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_set_trigger_mode(), **mx_area_detector_trigger()**

3.4 mx_area_detector_copy_frame

NAME

mx_area_detector_copy_frame - copy a frame on the area detector computer

SYNOPSIS

mx_status_type **mx_area_detector_copy_frame** (MX_RECORD *record,
                                                 long source_frame_type,
                                                 long destination_frame_type);

DESCRIPTION

This function copies a frame on the area detector control computer from one of the predefined frame buffers to another of the predefined frame buffers. The arguments *source_frame_type* and *destination_frame_type* are long integers that refer to the buffers. The list of available frame buffers can be found in Section 3.1.2

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

3.5 mx_area_detector_correct_frame

NAME

mx_area_detector_correct_frame - tell the detector computer to perform image correction

SYNOPSIS

mx_status_type **mx_area_detector_correct_frame** (MX_RECORD *record);

DESCRIPTION

This function tells the detector computer to perform all image corrections that are currently enabled. At present, the types of corrections available are *mask correction*, *bias correction*, *dark current correction*, and *flood field correction*.

There are two things that must be true for a given correction to be performed:

- The correction frame must have been loaded into the appropriate buffer on the detector computer. This can either be done automatically at program startup time or manually by a later call to **mx_load_frame()**.

- The bit in the correction flags bitmask for this correction must have been set either at program startup time or by a call to **mx_set_correction_flags()**.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_load_frame(), **mx_area_detector_set_correction_flags()**

3.6   mx_area_detector_get_binsize

NAME
mx_area_detector_get_binsize - reports the current x and y image binning

SYNOPSIS
mx_status_type **mx_area_detector_get_binsize** (MX_RECORD *record,
                                                        long *x_binsize,
                                                        long *y_binsize);

DESCRIPTION
This function reports the scale factors for image frame binning in the detector. For example, if the *x* binsize is 2, then the values of pairs of adjacent pixels in the X direction will added together and returned as one pixel value. If both the *x* and *y* binsizes are set to 2, then a two by two square of four pixels will be added together and returned as one pixel. If the detector is in unbinned mode, the binsizes reported will be 1.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_set_binsize()

3.7   mx_area_detector_get_bytes_per_frame

NAME
mx_area_detector_get_bytes_per_frame - reports the number of bytes in a single image frame

SYNOPSIS
mx_status_type **mx_area_detector_get_bytes_per_frame** (MX_RECORD *record,
                                                        long *bytes_per_frame);

DESCRIPTION
This function reports the number of bytes in a single image frame for the current image format, framesize, and binning. This size does not include the size of the image header (if present).

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

3.8   mx_area_detector_get_bytes_per_pixel

NAME

   mx_area_detector_get_bytes_per_pixel - reports the number of bytes per pixel in the current image format

SYNOPSIS

   mx_status_type **mx_area_detector_get_bytes_per_pixel** (MX_RECORD *record,
                                                                  double *bytes_per_pixel);

DESCRIPTION

   This function reports the number of bytes in a single image pixel for the current image format. *Bytes_per_pixel* is returned as a textitdouble due to the fact that there exist image formats for which the number of bytes per pixel is not an integer.

RETURN VALUE

   On success, the status code MXE_SUCCESS is returned.

3.9   mx_area_detector_get_correction_flags

NAME

   mx_area_detector_get_correction_flags - reports a bitmask listing the currently enabled image corrections

SYNOPSIS

   mx_status_type **mx_area_detector_get_correction_flags** (MX_RECORD *record,
                                                                  unsigned long *correction_flags);

DESCRIPTION

   This function returns a bitmask that describes the set of image corrections that are currently enabled. A given correction is enabled if the bit for that correction in the bitmask has a value of 1. A description of the bitmask for the available corrections can be found in the description for the **mx_area_detector_copy_frame()** function. Please note that the MXFT_AD_IMAGE_FRAME (0x1) bit is ignored in this context.

RETURN VALUE

   On success, the status code MXE_SUCCESS is returned.

SEE ALSO

   mx_area_detector_copy_frame(), **mx_area_detector_set_correction_flags()**

3.10   mx_area_detector_get_extended_status

NAME

   mx_area_detector_get_extended_status - reports both the last frame number and the status flags for an area detector.

SYNOPSIS

   mx_status_type **mx_area_detector_get_extended_status** (MX_RECORD *record,
                                                                  long *last_frame_number,
                                                                  unsigned long *status_flags);

DESCRIPTION

This function returns both the last area detector frame number just like **mx_area_detector_get_last_frame_number()** and the area detector status flags just like **mx_area_detector_get_status()**. See the descriptions of the other two functions for a more detailed description of what they return.

If the application program is polling for these values across an MX network connection, then this function will be more efficient than two separate calls to the other two functions, since only one network transaction will take place.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_last_frame_number(), **mx_area_detector_get_status()**

3.11   mx_area_detector_get_frame

NAME

mx_area_detector_get_frame - returns the requested image frame

SYNOPSIS

mx_status_type **mx_area_detector_get_frame** (MX_RECORD *record,
                                                                 long frame_number,
                                                                 MX_IMAGE_FRAME **frame);

DESCRIPTION

This is a utility function that returns an MX_IMAGE_FRAME structure that contains the image corresponding to the requested *frame_number*. If *frame_number* has the value -1, then the most recently acquired frame will be returned. This function consolidates a sequence of calls to **mx_area_detector_setup_frame()**, **mx_area_detector_readout_frame(** **mx_area_detector_correct_frame()**, and **mx_area_detector_transfer_frame()** into one call.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_setup_frame(), **mx_area_detector_readout_frame()**, **mx_area_detector_correct_frame()**, **mx_area_detector_transfer_frame()**

3.12   mx_area_detector_get_framesize

NAME

mx_area_detector_get_framesize - reports the current x and y image frame size

SYNOPSIS

mx_status_type **mx_area_detector_get_framesize** (MX_RECORD *record,
                                                                      long *x_framesize,
                                                                      long *y_framesize);

DESCRIPTION

This function reports the current resolution of image frames in the area detector taking binning into account. For example, if the maximum frame size is 4096 by 4096 with 2 by 2 binning, then **mx_area_detector_get_framesize()** will report a frame size of 2048 by 2048. If the detector is in unbinned mode, the frame size reported will be the same as that reported by **mx_area_detector_get_maximum_framesize()**.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_binsize(), **mx_area_detector_get_maximum_framesize()** **mx_area_detector_set_framesize()**

3.13 mx_area_detector_get_image_format

NAME

mx_area_detector_get_image_format - reports the current image format

SYNOPSIS

mx_status_type **mx_area_detector_get_image_format** (MX_RECORD *record,
long *image_format);

DESCRIPTION

This function returns the detector image format as a numerical value. The list of supported formats can be found in Section 4.1.1.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_set_image_format()

3.14 mx_area_detector_get_last_frame_number

NAME

mx_area_detector_get_last_frame_number - reports the most recently acquired frame number

SYNOPSIS

mx_status_type **mx_area_detector_get_last_frame_number** (MX_RECORD *record,
long *last_frame_number);

DESCRIPTION

This function reports the frame number for the image frame most recently acquired. Another way to look at it is that the frame number returned is the highest frame number for which a call to **mx_area_detector_readout_frame()** or **mx_area_detector_get_frame()** will currently succeed, if the current sequence is not a circular sequence. If the area detector has not yet finished taking the first frame in a new sequence, the value reported for *last_frame_number* will be -1.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_get_extended_status()**

3.15   mx_area_detector_get_maximum_framesize

NAME
> **mx_area_detector_get_maximum_framesize** - reports the maximum possible x and y image frame size

SYNOPSIS
> mx_status_type **mx_area_detector_get_maximum_framesize** (MX_RECORD *record,
>   long *maximum_x_framesize,
>   long *maximum_y_framesize);

DESCRIPTION
> This function reports the maximum possible image frame size for the area detector. If the area detector is currently in unbinned mode (binsize = 1), **mx_area_detector_get_framesize()** will report the same value.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_get_binsize()**, **mx_area_detector_get_framesize()**

3.16   mx_area_detector_get_property_double

NAME
> **mx_area_detector_get_property_double** - returns the value of the requested detector property as a double

SYNOPSIS
> mx_status_type **mx_area_detector_get_property_double** (MX_RECORD *record,
>   char *property_name,
>   double *property_double);

DESCRIPTION
> This function returns the property value for the requested detector property name as a double. If the property value is not representable as double, then the function returns an error.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_get_property_double()**, **mx_area_detector_set_property_double()**,
> **mx_area_detector_set_property_long()**, **mx_area_detector_get_property_string()**,
> **mx_area_detector_set_property_string()**

3.17 mx_area_detector_get_property_long

NAME

mx_area_detector_get_property_long - returns the value of the requested detector property as a long integer

SYNOPSIS

mx_status_type **mx_area_detector_get_property_long** (MX_RECORD *record,
                                                    char *property_name,
                                                    long *property_long);

DESCRIPTION

This function returns the property value for the requested detector property name as a long integer. If the property value is not representable as long integer, then the function returns an error.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_property_double(), **mx_area_detector_set_property_double**(),
mx_area_detector_set_property_long(), **mx_area_detector_get_property_string**(),
mx_area_detector_set_property_string()

3.18 mx_area_detector_get_property_string

NAME

mx_area_detector_get_property_string - returns the string value of the requested detector property

SYNOPSIS

mx_status_type **mx_area_detector_get_property_string** (MX_RECORD *record,
                                                      char *property_name,
                                                      char *property_string,
                                                      size_t max_string_length);

DESCRIPTION

This function returns the property string value for the requested detector property name.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_property_double(), **mx_area_detector_set_property_double**(),
mx_area_detector_get_property_long(), **mx_area_detector_set_property_long**(),
mx_area_detector_set_property_string()

3.19 mx_area_detector_get_roi

NAME

mx_area_detector_get_roi - returns the boundaries of the requested region of interest

SYNOPSIS

    mx_status_type **mx_area_detector_get_roi** (MX_RECORD *record,
                                   unsigned long roi_number,
                                   unsigned long *roi);

DESCRIPTION

    This function returns the boundaries of the region of interest (ROI) specified by *roi_number*. The boundaries of the ROI are expressed in binned coordinates. The data in the boundary rows and columns is considered to be part of the ROI.

    The *roi* array argument is an array of four unsigned longs with the boundaries stored in the order X_{min}, X_{max}, Y_{min} and Y_{max}. Here is an example of using this function:

```
...
MX_RECORD *ad_record;
unsigned long roi_number;
unsigned long roi[4];
...
roi_number = 5;

mx_status = mx_area_detector_get_roi( ad_record, roi_number, roi );

if ( mx_status.code != MXE_SUCCESS )
    return mx_status;

fprintf(stderr,
    "ROI(%lu) = Xmin = %lu, Xmax = %lu, Ymin = %lu, Ymax = %lu\n",
            roi_number, roi[0], roi[1], roi[2], roi[3] );
...
```

RETURN VALUE

    On success, the status code MXE_SUCCESS is returned.

SEE ALSO

    mx_area_detector_set_roi()

3.20   mx_area_detector_get_roi_frame

NAME

    mx_area_detector_get_roi_frame - returns the requested ROI as an image frame

SYNOPSIS

    mx_status_type **mx_area_detector_get_roi_frame** (MX_RECORD *record,
                                       MX_IMAGE_FRAME *frame);
                                       unsigned long roi_number,
                                       MX_IMAGE_FRAME **roi_frame);

DESCRIPTION

This returns an MX_IMAGE_FRAME structure that contains the contents of the region of interest (ROI) corresponding to the requested *roi_number*.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_roi(), **mx_area_detector_set_roi**()

3.21 mx_area_detector_get_sequence

NAME

mx_area_detector_get_sequence - reads all image frame in a sequence

SYNOPSIS

mx_status_type **mx_area_detector_get_sequence** (MX_RECORD *record,
                                                    long num_frames,
                                                    MX_IMAGE_SEQUENCE **sequence);

DESCRIPTION

Reads out all of the frames in an MX_IMAGE_SEQUENCE. The images are read out by a series of calls to **mx_area_detector_get_sequence**().

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_frame()

3.22 mx_area_detector_get_sequence_parameters

NAME

mx_area_detector_get_sequence_parameters - reports the current imaging sequence type

SYNOPSIS

mx_status_type **mx_area_detector_get_sequence_parameters** (MX_RECORD *record,
                                                    MX_SEQUENCE_PARAMETERS *sequence_parameters);

DESCRIPTION

This function reports the current imaging sequence parameters for the area detector in an MX_SEQUENCE_PARAMETERS structure. The MX_SEQUENCE_PARAMETERS structure is defined as follows:

```
typedef struct {
        long sequence_type;
        long num_parameters;
        double parameter_array[MXU_MAX_SEQUENCE_PARAMETERS];
} MX_SEQUENCE_PARAMETERS;
```

The *sequence_type* member specifies which type of sequence has been requested using the sequence type definitions near the top of the **$MXDIR/include/mx_image.h** header file. The *num_parameters* and the *parameter_array* members provide sequence type specific information for the sequence in question.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_set_sequence_parameters()

3.23   mx_area_detector_get_status

NAME
mx_area_detector_get_status - gets the area detector status flags

SYNOPSIS
mx_status_type **mx_area_detector_get_status** (MX_RECORD *record,
unsigned long status_flags);

DESCRIPTION
Returns a bitmap of status flags where the individual flag bits represent different aspects of the current status of the area detector. The currently defined status flags are described in Section 3.1.1.

At present, the only status flag defined is MXSF_AD_IS_BUSY. It is anticipated that this will expand to include status flags for internal software or hardware faults of the detector system.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_get_extended_status()

3.24   mx_area_detector_get_subframe_size

NAME
mx_area_detector_get_subframe_size - reports the number of columns in a subframe

SYNOPSIS
mx_status_type **mx_area_detector_get_subframe_size** (MX_RECORD *record,
unsigned long *num_columns);

DESCRIPTION
For the AVIEX PCCD-170170, it is possible to configure the detector hardware to only readout a subset of the columns in the CCD centered at the center of the <**FIXME**> element of the detector. This function tells how many columns are in the subframe. If we are not in subframe mode, the reported subframe size will be -1. While this mode is in effect, the software ROIs will be relative to the boundaries of this subframe rather than the dimensions of the full frame.

<**FIXME**> - How does binning affect this?

WARNING

This function is only available for the AVIEX PCCD-170170 CCD detector.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_set_subframe_size()

3.25 mx_area_detector_get_trigger_mode

NAME

mx_area_detector_set_trigger_mode - reports the internal/external trigger mode for the detector

SYNOPSIS

mx_status_type **mx_area_detector_get_trigger_mode** (MX_RECORD *record,
long *trigger_mode);

DESCRIPTION

This function is used to report whether or not the area detector is in internal or external trigger mode.

WARNING

This function has not yet been really implemented, so for now you should assume that you are in internal trigger mode.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_set_trigger_mode()

3.26 mx_area_detector_get_use_scaled_dark_current_flag

NAME

mx_area_detector_get_use_scaled_dark_current_flag - returns the use_scaled_dark_current flag

SYNOPSIS

mx_status_type **mx_area_detector_get_use_scaled_dark_current_flag** (MX_RECORD *record,
mx_bool_type *use_scaled_dark_current);

DESCRIPTION

This function returns the state of the *use_scaled_dark_current* flag in the detector computer. If the *use_scaled_dark_current* flag is set in the detector computer, dark current subtraction will be done using dark current values that have been rescaled to the actual exposure time of the image. If the *use_scaled_dark_current* flag is *not* set, then raw unscaled dark current values will be subtracted instead.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_set_use_scaled_dark_current_flag**

3.27 mx_area_detector_is_busy

NAME
> **mx_area_detector_is_busy** - reports whether or not the area detector is acquiring an image sequence

SYNOPSIS
> mx_status_type **mx_area_detector_is_busy** (MX_RECORD *record,
>
> mx_bool_type *busy);

DESCRIPTION
> This is a utility function that reports whether or not the area detector is currently acquiring images for an image sequence. **mx_area_detector_is_busy()** is implemented by invoking **mx_area_detector_get_status()** and returning only the value of the MXSF_AD_IS_BUSY as an MX boolean value.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_get_status()**, **mx_area_detector_get_extended_status()**

3.28 mx_area_detector_load_frame

NAME
> **mx_area_detector_load_frame** - load an image frame into the requested image buffer

SYNOPSIS
> mx_status_type **mx_area_detector_load_frame** (MX_RECORD *record,
>
> long frame_type,
>
> char *frame_filename);

DESCRIPTION
> This function loads an image frame into a frame buffer on the detector computer specified by *frame_type* from the file on the detector computer specified by *frame_filename*. This function is intended to be used for loading mask, bias, dark current, and flood field image frames on the detector computer. The image format for the file must match the image format of the detector computer's area detector record. If it does not, the load will fail. The allowed values of the *frame_type* argument can be found in the description of the **mx_area_detector_copy_frame()** function.

> If you want to load an image frame into a user application program, then you should be using **mx_image_read_file()** instead.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_copy_frame()**, **mx_area_detector_save_frame()**, **mx_image_read_file()**,
> **mx_image_write_file()**

3.29 mx_area_detector_measure_correction_frame

NAME

mx_area_detector_measure_correction_frame - perform a series of measurements to construct a correction frame

SYNOPSIS

mx_status_type **mx_area_detector_measure_correction_frame** (MX_RECORD *record,
                                        long correction_measurement_type,
                                        double correction_measurement_time,
                                        long num_correction_measurements);

DESCRIPTION

mx_area_detector_measure_correction_frame() is used to generate the data with which to construct either a dark current correction frame or a flood field correction frame. The *correction_measurement_type* is one of the two values MXFT_AD_DARK_CURRENT_FRAME or MXFT_AD_FLOOD_FIELD_FRAME as described in Section 3.1.2. The correction frame is measured by taking the average of *num_correction_measurements* worth of detector images for an exposure time per frame of *correction_measurement_time*.

In general, the dark current frame should be measured without any radiation hitting the detector, while the flood field frame should be measured with a uniform source of radiation hitting the detector.

Since the absolute intensity of the dark current frame is normally a function of the exposure time, you can choose to either rescale the dark current intensity to the actual exposure time or else merely subtract the unscaled dark current value. The choice of mode can be made using the function **mx_area_detector_set_use_scaled_dark_current_flag()**.

There are also a pair of convenience macros called **mx_area_detector_measure_dark_current_frame()** and **mx_area_detector_measure_flood_field_frame()** which are merely front ends to **mx_area_detector_measure_correction_frame()**.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_measure_dark_current_frame(), **mx_area_detector_measure_flood_field_frame()**, **mx_area_detector_set_use_scaled_dark_current_flag()**

3.30 mx_area_detector_measure_dark_current_frame

NAME

mx_area_detector_measure_dark_current_frame - perform a series of measurements to construct a dark current frame

SYNOPSIS

mx_status_type **mx_area_detector_measure_dark_current_frame** (MX_RECORD *record,
                                        double correction_measurement_time,
                                        long num_correction_measurements);

DESCRIPTION

mx_area_detector_measure_dark_current_frame() is a convenience macro that invokes the function **mx_area_detector_measure** with the *correction_measurement_type* set to MXFT_AD_DARK_CURRENT_FRAME. See the description of **mx_area_detector_measure_correction_frame()** for more information.

RETURN VALUE
     On success, the status code MXE_SUCCESS is returned.

SEE ALSO
     mx_area_detector_measure_correction_frame(), **mx_area_detector_measure_flood_field_frame()**

3.31   mx_area_detector_measure_flood_field_frame

NAME
     mx_area_detector_measure_flood_field_frame - perform a series of measurements to construct a dark current
     frame

SYNOPSIS
     mx_status_type **mx_area_detector_measure_dark_current_frame** (MX_RECORD *record,
                                                                      double correction_measurement_time,
                                                                      long num_correction_measurements);

DESCRIPTION
     mx_area_detector_measure_flood_field_frame() is a convenience macro that invokes the function **mx_area_detector_measure_c**
     with the *correction_measurement_type* set to MXFT_AD_FLOOD_FIELD_FRAME. See the description of **mx_area_detector_mea**
     for more information.

RETURN VALUE
     On success, the status code MXE_SUCCESS is returned.

SEE ALSO
     mx_area_detector_measure_correction_frame(), **mx_area_detector_measure_dark_current_frame()**

3.32   mx_area_detector_readout_frame

NAME
     mx_area_detector_readout_frame - read a frame into the primary image buffer

SYNOPSIS
     mx_status_type **mx_area_detector_readout_frame** (MX_RECORD *record,
                                                         long frame_number);

DESCRIPTION
     This function reads the requested image frame from the area detector hardware into the primary image buffer
     of the detector computer. If the requested frame number is -1, the most recently acquired image will be read
     out.

RETURN VALUE
     On success, the status code MXE_SUCCESS is returned.

3.33 mx_area_detector_save_frame

NAME

mx_area_detector_save_frame - save a frame from the requested image buffer to a file

SYNOPSIS

mx_status_type **mx_area_detector_save_frame** (MX_RECORD *record,
                                                long frame_type,
                                                char *frame_filename);

DESCRIPTION

This function saves an image frame from the frame buffer on the detector computer specified by *frame_type* to the file on the detector computer specified by *frame_filename*. This function is primarily intended to be used for saving the most recently acquired image frame to a disk file on the detector computer. However, it save frames from any of the image buffers listed in the description of the **mx_area_detector_copy_frame()** function. The image file will be written using the current image format of the area detector.

If you want a user application program to save an image frame to disk, then you should be using **mx_image_write_file()** instead.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_copy_frame(), **mx_area_detector_load_frame()**, **mx_image_read_file()**, **mx_image_write_file()**

3.34 mx_area_detector_set_binsize

NAME

mx_area_detector_set_binsize - set the current x and y image binning factors

SYNOPSIS

mx_status_type **mx_area_detector_set_binsize** (MX_RECORD *record,
                                                 long x_binsize,
                                                 long y_binsize);

DESCRIPTION

This function sets the scale factor for image frame binning in the detector. For example, if the *x* binsize is 2, then the values of pairs of adjacent pixels in the X direction will added together and returned as one pixel value. If both the *x* and *y* binsizes are set to 2, then a two by two square of four pixels will be added together and returned as one pixel. If you want to put the detector into unbinned mode, then set both binsizes to 1.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_binsize()

3.35   mx_area_detector_set_bulb_mode

NAME

mx_area_detector_set_bulb_mode - change the area detector to use Bulb mode image sequences

SYNOPSIS

mx_status_type **mx_area_detector_set_bulb_mode** (MX_RECORD *record,
long num_frames);

DESCRIPTION

This function configures the area detector to use a *Bulb mode* image sequence. In Bulb mode, the area detector uses the current state of an external trigger signal to determine when the exposure for the current frame should start and end. In other words, in Bulb mode the area detector will expose the current frame for as long as the external trigger input is set to high. When the external trigger input goes to the low state, the exposure for the current frame ends. The next frame does not start until the external trigger input goes high again. The area detector stops taking frames once the number of frames specified by the *num_frames* argument have been taken.

If the area detector has not been configured by **mx_area_detector_set_trigger_mode()** to use an external trigger at the time that the next image sequence is started, the attempt to start the sequence will fail with an error.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_set_circular_multiframe_mode(), mx_area_detector_set_continuous_mode(),
mx_area_detector_set_geometrical_mode(), mx_area_detector_set_multiframe_mode(),
mx_area_detector_set_one_shot_mode(), mx_area_detector_set_sequence_parameters(),
mx_area_detector_set_strobe_mode(), mx_area_detector_set_trigger_mode()

3.36   mx_area_detector_set_circular_multiframe_mode

NAME

mx_area_detector_set_circular_multiframe_mode - change the area detector to use Circular Multiframe mode image sequences

SYNOPSIS

mx_status_type **mx_area_detector_set_circular_multiframe_mode** (MX_RECORD *record,
long num_frames,
double exposure_time,
double gap_time);

DESCRIPTION

This function configures the area detector to use a *Circular Multiframe mode* image sequence. Circular Multiframe mode sequences are almost identical to Multiframe mode sequences as described by the documentation for **mx_area_detector_set_multiframe_mode()**. The only difference is that once the last frame has been acquired, the sequence goes back to the beginning to overwrite the first frame. The sequence does not stop until explicitly stopped or aborted.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO
     mx_area_detector_abort(), **mx_area_detector_set_multiframe_mode()**, **mx_area_detector_stop()**

3.37   mx_area_detector_set_continuous_mode

NAME
     mx_area_detector_set_continuous_mode - change the area detector to use Continuous mode image sequences

SYNOPSIS
     mx_status_type **mx_area_detector_set_continuous_mode** (MX_RECORD *record,
                                                                                double exposure_time)

DESCRIPTION
     This function configures the area detector to use a *Continuous mode* image sequence. In Continuous mode, the
     area detector repeatedly takes image frames that all have the same duration as requested by the *exposure_time*
     argument in seconds. Each new frame overwrites the previous one. The detector will continue taking frames
     until explicitly stopped or aborted. It is anticipated that this mode will be mostly useful for diagnostic applica-
     tions that want a continuously updated GUI display of images acquired by the detector.

     Continuous mode can be used with either an internal trigger or an external trigger. If a strobed external trig-
     ger mode been requested by **mx_area_detector_set_trigger_mode()**, the detector will take a new frame for
     each external trigger pulse received by the detector. If a non-strobed external trigger mode has been selected,
     the detector will merely use the first external trigger received to start the sequence and ignore all subsequent
     triggers.

RETURN VALUE
     On success, the status code MXE_SUCCESS is returned.

SEE ALSO
     mx_area_detector_abort(), **mx_area_detector_set_bulb_mode()**,
     mx_area_detector_set_circular_multiframe_mode(), **mx_area_detector_set_geometrical_mode()**,
     mx_area_detector_set_multiframe_mode(), **mx_area_detector_set_one_shot_mode()**,
     mx_area_detector_set_sequence_parameters(), **mx_area_detector_set_strobe_mode()**,
     mx_area_detector_set_trigger_mode(), **mx_area_detector_stop()**

3.38   mx_area_detector_set_correction_flags

NAME
     mx_area_detector_set_correction_flags - specifies which image corrections are to be performed.

SYNOPSIS
     mx_status_type **mx_area_detector_set_correction_flags** (MX_RECORD *record,
                                                                                unsigned long correction_flags);

DESCRIPTION
     The *correction_flags* argument for this function is a bitmask that describes the set of image corrections to be en-
     abled. A given correction is enabled if the bit for that correction in the bitmask has a value of 1. A description of
     the bitmask for the available corrections can be found in the description for the **mx_area_detector_copy_frame()**
     function. Please note that the MXFT_AD_IMAGE_FRAME (0x1) bit has no meaning in this context.

RETURN VALUE
    On success, the status code MXE_SUCCESS is returned.

SEE ALSO
    mx_area_detector_copy_frame(), **mx_area_detector_get_correction_flags()**

3.39   mx_area_detector_set_framesize

NAME
    mx_area_detector_get_framesize - sets the current x and y image frame size

SYNOPSIS
    mx_status_type **mx_area_detector_get_framesize** (MX_RECORD *record,
                                                    long x_framesize,
                                                    long y_framesize);

DESCRIPTION
    For some detectors, this function sets the current resolution of image frames in the area detector taking binning
    into account. Not all detectors support this function. Some will round the requested dimensions to the reported
    size, while others will return an error regardless of the framesize values supplied.

RETURN VALUE
    On success, the status code MXE_SUCCESS is returned.

SEE ALSO
    mx_area_detector_get_binsize(), **mx_area_detector_get_framesize()**,
    mx_area_detector_get_maximum_framesize()

3.40   mx_area_detector_set_geometrical_mode

NAME
    mx_area_detector_set_geometrical_mode - change the area detector to use Geometrical mode image sequences

SYNOPSIS
    mx_status_type **mx_area_detector_set_geometrical_mode** (MX_RECORD *record,
                                                    long num_frames,
                                                    double exposure_time,
                                                    double gap_time, double exposure_multiplier,
                                                    double gap_multiplier);

DESCRIPTION
    Geometrical mode sequences are currently only supported by the AVIEX PCCD-170170 CCD detector. Geo-
    metrical mode sequences are multiframe sequences that take the number of frames requested by the *num_frames*
    argument. For the first frame, the duration of the exposure time in seconds will be the value requested by the
    exposure_time argument. The gap between the first and the second frames in seconds will be the time requested
    by the *gap_time*. For each subsequent frame, the exposure time will be the exposure time for the previous
    frame multiplied by the value of the *exposure_multipler* argument. Similarly, the duration of the gap between
    frames will be the previous gap time multiplied by the value of the *gap_multiplier* argument. This means that
    the exposure time and the gap time will continue to get longer as the sequence progresses.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_set_bulb_mode()**, **mx_area_detector_set_circular_multiframe_mode()**,
> **mx_area_detector_set_continuous_mode()**, **mx_area_detector_set_multiframe_mode()**,
> **mx_area_detector_set_one_shot_mode()**, **mx_area_detector_set_sequence_parameters()**,
> **mx_area_detector_set_strobe_mode()**, **mx_area_detector_set_trigger_mode()**

3.41 mx_area_detector_set_image_format

NAME
> **mx_area_detector_set_image_format** - changes the current image format

SYNOPSIS
> mx_status_type **mx_area_detector_set_image_format** (MX_RECORD *record,
> > long image_format);

DESCRIPTION
> This function sets the detector image format as a numerical value. A list of the supported image formats can be found in Section 4.1.1.

WARNING
> Most detectors do not support this function.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_get_image_format()**

3.42 mx_area_detector_set_multiframe_mode

NAME
> **mx_area_detector_set_multiframe_mode** - change the area detector to use Multiframe mode image sequences

SYNOPSIS
> mx_status_type **mx_area_detector_set_multiframe_mode** (MX_RECORD *record,
> > long num_frames,
> > double exposure_time,
> > double gap_time);

DESCRIPTION
> This function configures the area detector to use a *Multiframe mode* image sequence. In Multiframe mode, all image frames are of the same duration as specified by the *exposure_time* argument in seconds. In addition, the spacing between all pairs of image frames will be the same as specified by the *gap_time* argument in seconds.
>
> Multiframe mode can be used with either an internal trigger or an external trigger. If an external trigger mode has been configured by **mx_area_detector_set_trigger_mode()**, the sequence uses the next pulse from

the external trigger to start the imaging sequences. All subsequent trigger pulses are ignored. The imaging sequence will stop once the number of image frames requested by the *num_frames* argument have been taken.

If you want the sequence to loop by going back to overwrite the first frame once the last frame has been acquired, you should use **mx_area_detector_set_circular_multiframe_mode()** instead.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_set_bulb_mode(), **mx_area_detector_set_circular_multiframe_mode()**,
mx_area_detector_set_continuous_mode(), **mx_area_detector_set_geometrical_mode()**,
mx_area_detector_set_one_shot_mode(), **mx_area_detector_set_sequence_parameters()**,
mx_area_detector_set_strobe_mode(), **mx_area_detector_set_trigger_mode()**

3.43   mx_area_detector_set_one_shot_mode

NAME
mx_area_detector_set_one_shot_mode - change the area detector to use One-shot mode image sequences

SYNOPSIS
mx_status_type **mx_area_detector_set_one_shot_mode** (MX_RECORD *record,
                                                                                                double exposure_time)

DESCRIPTION
This function configures the area detector to use a *One-shot mode* image sequence. In One-shot mode, the area detector takes a single frame that is exposed for the duration requested by the *exposure_time* argument. One-shot mode can be used with either an internal trigger or an external trigger.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_abort(), **mx_area_detector_set_bulb_mode()**,
mx_area_detector_set_circular_multiframe_mode(), **mx_area_detector_set_continuous_mode()**,
mx_area_detector_set_geometrical_mode(), **mx_area_detector_set_multiframe_mode()**,
mx_area_detector_set_sequence_parameters(), **mx_area_detector_set_strobe_mode()**,
mx_area_detector_set_trigger_mode(), **mx_area_detector_stop()**

3.44   mx_area_detector_set_property_double

NAME
mx_area_detector_set_property_double - sets the value of the requested detector property

SYNOPSIS
mx_status_type **mx_area_detector_set_property_double** (MX_RECORD *record,
                                                                                                char *property_name,
                                                                                                double property_double);

DESCRIPTION

This function sets the property value for the requested detector property name as a double. If the property value is not representable as double, then the function returns an error.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_property_double(), **mx_area_detector_get_property_long**(), **mx_area_detector_set_property_long**(), **mx_area_detector_get_property_string**(), **mx_area_detector_set_property_string**()

3.45   mx_area_detector_set_property_long

NAME

mx_area_detector_set_property_long - sets the value of the requested detector property

SYNOPSIS

mx_status_type **mx_area_detector_set_property_long** (MX_RECORD *record,
                                                char *property_name,
                                                long property_long);

DESCRIPTION

This function sets the property value for the requested detector property name as a long integer. If the property value is not representable as long integer, then the function returns an error.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_property_double(), **mx_area_detector_set_property_double**(), **mx_area_detector_get_property_long**(), **mx_area_detector_get_property_string**(), **mx_area_detector_set_property_string**()

3.46   mx_area_detector_set_property_string

NAME

mx_area_detector_set_property_string - sets the string value of the requested detector property

SYNOPSIS

mx_status_type **mx_area_detector_set_property_string** (MX_RECORD *record,
                                                char *property_name,
                                                char *property_string);

DESCRIPTION

This function sets the property string value for the requested detector property name.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO
 mx_area_detector_get_property_string(), **mx_area_detector_get_property_value()**,
 mx_area_detector_set_property_value()

3.47   mx_area_detector_set_roi

NAME
 mx_area_detector_set_roi - sets the boundaries of the requested region of interest

SYNOPSIS
 mx_status_type **mx_area_detector_set_roi** (MX_RECORD *record,
               unsigned long roi_number,
               unsigned long *roi);

DESCRIPTION
 This function returns the boundaries of the region of interest (ROI) specified by *roi_number*. The boundaries of
 the ROI are expressed in binned coordinates. The data in the boundary rows and columns is considered to be
 part of the ROI.

 The *roi* array argument is an array of four unsigned longs with the boundaries stored in the order X_{min}, X_{max},
 Y_{min} and Y_{max}. Here is an example of using this function:

```
...
MX_RECORD *ad_record;
unsigned long roi_number;
unsigned long roi[4];
...
roi_number = 5;

roi[0] = 1000;     /* X minimum */
roi[1] = 2000;     /* X maximum */
roi[2] = 1250;     /* Y minimum */
roi[3] = 1750;     /* Y maximum */

mx_status = mx_area_detector_set_roi( ad_record, roi_number, roi );

if ( mx_status.code != MXE_SUCCESS )
    return mx_status;
...
```

RETURN VALUE
 On success, the status code MXE_SUCCESS is returned.

SEE ALSO
 mx_area_detector_get_roi()

3.48 mx_area_detector_set_sequence_parameters

NAME

   mx_area_detector_set_sequence_parameters - sets up any of the available types of image sequences.

SYNOPSIS

   mx_status_type **mx_area_detector_set_sequence_parameters** (MX_RECORD *record,
                                                         MX_SEQUENCE_PARAMETERS *sequence_parameters);

DESCRIPTION

   mx_area_detector_set_sequence_parameters() is the common function that underlies all of the other com-
   mands for selecting specific sequence modes. The command takes a single argument which is an MX_SEQUENCE_PARAMETERS
   structure. The MX_SEQUENCE_PARAMETERS structure is defined as follows:

```
typedef struct {
        long sequence_type;
        long num_parameters;
        double parameter_array[MXU_MAX_SEQUENCE_PARAMETERS];
} MX_SEQUENCE_PARAMETERS;
```

   The *sequence_type* member specifies which type of sequence has been requested. The *num_parameters* and the
   parameter_array members provide sequence type specific information for the sequence in question. In general,
   it is simpler to use the higher level sequence specific mode setting functions.

RETURN VALUE

   On success, the status code MXE_SUCCESS is returned.

SEE ALSO

   mx_area_detector_set_bulb_mode(), **mx_area_detector_set_circular_multiframe_mode()**,
   mx_area_detector_set_continuous_mode(), **mx_area_detector_set_geometrical_mode()**,
   mx_area_detector_set_multiframe_mode(), **mx_area_detector_set_one_shot_mode()**,
   mx_area_detector_set_strobe_mode(), **mx_area_detector_set_trigger_mode()**

3.49 mx_area_detector_set_strobe_mode

NAME

   mx_area_detector_set_strobe_mode - change the area detector to use Strobe mode image sequences

SYNOPSIS

   mx_status_type **mx_area_detector_set_strobe_mode** (MX_RECORD *record,
                                                 long num_frames,
                                                 double exposure_time);

DESCRIPTION

   This function configures the area detector to use a *Strobe mode* image sequence. In Strobe mode, the start of
   each frame in the sequence is triggered by an external trigger signal. The exposure time for each frame lasts for
   the amount of time in seconds requested by the *exposure_time* argument. The area detector stops taking frames
   once the number of frames specified by the *num_frames* argument have been taken. If an external trigger signal

arrives before the preceding frame has finished, the results are undefined and depend on the particular area detector hardware in use.

If the area detector has not been configured by **mx_area_detector_set_trigger_mode()** to use an external trigger at the time that the next image sequence is started, the attempt to start the sequence will fail with an error.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_set_bulb_mode(), **mx_area_detector_set_circular_multiframe_mode()**,
mx_area_detector_set_continuous_mode(), **mx_area_detector_set_geometrical_mode()**,
mx_area_detector_set_multiframe_mode(), **mx_area_detector_set_one_shot_mode()**,
mx_area_detector_set_sequence_parameters(), **mx_area_detector_set_trigger_mode()**

3.50   mx_area_detector_set_subframe_size

NAME
mx_area_detector_set_subframe_size - tells the area detector to readout only a subframe

SYNOPSIS
mx_status_type **mx_area_detector_set_subframe_size** (MX_RECORD *record,
                                                                                  long num_columns);

DESCRIPTION
This function, which is available for the AVIEX PCCD-170170, tells the detector hardware to only readout a subset of the columns in the CCD centered at the center of the <**FIXME**> element of the detector. While this mode is in effect, the software ROIs will be relative to the boundaries of this subframe rather than the dimensions of the full frame. To revert back to full frame operation, call this function again with the number of columns set to -1.

<**FIXME**> - How does binning affect this?

WARNING
This function is only available for the AVIEX PCCD-170170 CCD detector.

RETURN VALUE
On success, the status code MXE_SUCCESS is returned.

SEE ALSO
mx_area_detector_get_subframe_size()

3.51   mx_area_detector_set_trigger_mode

NAME
mx_area_detector_set_trigger_mode - sets the internal/external trigger mode for the detector

SYNOPSIS
mx_status_type **mx_area_detector_set_trigger_mode** (MX_RECORD *record,
                                                                                  long trigger_mode);

DESCRIPTION

This function will be used to switch between internal and external trigger mode.

WARNING

This function has not yet been really implemented, so for now you should assume that you are in internal trigger mode.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_trigger_mode()

3.52   mx_area_detector_set_use_scaled_dark_current_flag

NAME

mx_area_detector_set_use_scaled_dark_current_flag - changes the state of the use_scaled_dark_current flag

SYNOPSIS

mx_status_type **mx_area_detector_set_use_scaled_dark_current_flag** (MX_RECORD *record,
                                                    mx_bool_type use_scaled_dark_current);

DESCRIPTION

This function changes the state of the *use_scaled_dark_current* flag in the detector computer. If the *use_scaled_dark_current* flag is set in the detector computer, dark current subtraction will be done using dark current values that have been rescaled to the actual exposure time of the image. If the *use_scaled_dark_current* flag is *not* set, then raw unscaled dark current values will be subtracted instead.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_get_use_scaled_dark_current_flag

3.53   mx_area_detector_setup_frame

NAME

mx_area_detector_setup_frame - create a new frame or modify the size of an existing one.

SYNOPSIS

mx_status_type **mx_area_detector_setup_frame** (MX_RECORD *record,
                                    MX_IMAGE_FRAME *frame);

DESCRIPTION

This function takes a pointer to an MX_IMAGE_FRAME pointer as its second argument as in this example:

```
            ...
            MX_IMAGE_FRAME *frame;
            ...
            mx_status = mx_area_detector_setup_frame( record, &frame );
            ...
```

If you assign NULL to the *frame* pointer before invoking **mx_area_detector_setup_frame**(), it assumes that you want to create a new MX_IMAGE_FRAME structure with dimensions that match the current configuration of the specified area detector.

If you assign NULL to the *frame* pointer before invoking **mx_area_detector_setup_frame**(), it assumes that you just want to verify that the MX_IMAGE_FRAME object you are passing contains an *image_data* array that is big enough to hold a new image frame read from the detector. If the array is already big enough, then nothing is done to the MX_IMAGE_FRAME object. If the array is not big enough, the old array is freed and a new array is allocated to take its place.

The purpose of **mx_area_detector_setup_frame()** is to make it easy to always ensure that the image frame object you are using is big enough to hold a new frame from the area detector, while minimizing the number of memory allocations that have to be performed.

Please note that framesize, format, and other parameters of the MX_IMAGE_FRAME are determined by looking at the current configuration of the specified area detector record. If you want to directly specify all of these parameters yourself, then the function you want to use is **mx_image_alloc()**.

RETURN VALUE
    On success, the status code MXE_SUCCESS is returned.

SEE ALSO
    mx_image_alloc(), **mx_image_free()**

3.54 mx_area_detector_start

NAME
    mx_area_detector_start - starts a imaging sequence

SYNOPSIS
    mx_status_type **mx_area_detector_start** (MX_RECORD *record);

DESCRIPTION
    This is a utility function that starts an imaging sequence by invoking **mx_area_detector_arm()** followed by **mx_area_detector_trigger()**.

RETURN VALUE
    On success, the status code MXE_SUCCESS is returned.

SEE ALSO
    mx_area_detector_arm(), **mx_area_detector_trigger()**

3.55 mx_area_detector_stop

NAME

mx_area_detector_stop - stops all area detector activity after the current frame

SYNOPSIS

mx_status_type **mx_area_detector_stop** (MX_RECORD *record);

DESCRIPTION

This function tells the area detector to stop any imaging sequence in process after the current frame completes.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_abort()

3.56 mx_area_detector_transfer_frame

NAME

mx_area_detector_transfer_frame - sends an image frame to a user application

SYNOPSIS

mx_status_type **mx_area_detector_transfer_frame** (MX_RECORD *record,
long frame_type);

DESCRIPTION

This function tells the detector computer to send an image from the requested detector frame buffer to the user application. The transferred frame will be saved in the *image_frame* buffer attached to the user application's area detector record. You must make sure that the *image_frame* buffer has been set up in advance by a call to either **mx_area_detector_setup_frame**() or **mx_image_alloc**() before invoking this function or else it will return an error.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_setup_frame(), **mx_area_detector_readout_frame**(), **mx_image_alloc**()

3.57 mx_area_detector_trigger

NAME

mx_area_detector_trigger - sends a internal trigger to the area detector

SYNOPSIS

mx_status_type **mx_area_detector_trigger** (MX_RECORD *record);

DESCRIPTION

This function tells the detector computer to send an internal trigger to the area detector hardware to start an imaging sequence.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_arm(), **mx_area_detector_start()**

Chapter 4

Image API Reference

4.1 Image Definitions

4.1.1 Image Formats

Each image handled by MX has a defined image format. The following three formats are the most important ones:

MXT_IMAGE_FORMAT_RGB (1)
> This is a 24-bit color format with 8 bits each for Red, Green, and Blue.

MXT_IMAGE_FORMAT_GREY8 (2)
> This format is 8-bit greyscale.

MXT_IMAGE_FORMAT_GREY16 (3)
> This format is 8-bit greyscale.

These formats are important since all other in memory image formats are eventually converted to one of these three format by MX. Keep in mind that in memory image formats are not the same as datafile image formats.

The following formats are raw formats as generated by image capture boards:

MXT_IMAGE_FORMAT_RGB565 (1001)
> This is a 16-bit color format with 5 bits for red, 6 bits for green, and 5 bits for blue.

MXT_IMAGE_FORMAT_YUYV (1002)
> This is a packed YCbCr color format which is known in the Windows world as YUY2.

4.1.2 Datafile Formats

At the moment, only one image file format has been implemented for temporary testing purposes. This format is:

MXT_IMAGE_FILE_PNM (1)
> The PNM file format is described at *http://netpbm.sourceforge.net/doc/index.html*. It is supported by a wide variety of image viewing programs in the Linux/Unix world and is supported at least by Irfanview in the Windows world. This format is not intended for routine use with scientific data, since it does not really have a place to store header information.

4.2   mx image alloc

NAME

   mx image alloc - allocate an MX IMAGE FRAME object

SYNOPSIS

   mx status type **mx image alloc** (MX IMAGE FRAME **frame,
                                long image type,
                                long *framesize,
                                long image format,
                                long pixel order,
                                double bytes per pixel,
                                size t header length,
                                size t image length);

WARNING

   The argument list for this function is likely to change in the near future. For now, if you can, you are probably
   better off relying on **mx area detector setup frame()** to initialize the MX IMAGE FRAME structure for you.

DESCRIPTION

   This function either creates a new MX IMAGE FRAME object or else changes the size of an existing MX IMAGE FRAME
   object that matches the supplied function arguments. If you want to automatically fetch the appropriate config-
   uration from your area detector record, you should use **mx area detector setup frame()** instead.

   This function takes as its first argument a pointer to an MX IMAGE FRAME pointer. If the frame pointer
   passed is NULL, as in this example

```
...
MX_IMAGE_FRAME *frame;
...
frame = NULL;
...
mx_status = mx_image_frame( &frame, ... );
...
```

   then **mx image alloc()** will create a new MX IMAGE FRAME structure using the requested configuration.

   If the frame pointer passed to **mx image alloc()** is *not* NULL, **mx image alloc()** will examine the current
   configuration of the supplied MX IMAGE FRAME structure to see if it is already capable of holding an image
   frame with the requested configuration. If the object can already hold the frame, **mx image alloc()** returns
   without doing anything else. If the object *cannot* already hold the image frame, **mx image alloc()** will resize
   the *image data* and *header data* arrays in the object so that they are big enough to hold an image frame with
   the new dimensions. If the image frame object is resized, the old contents of the *image data* and *header data*
   arrays are not preserved.

   The arguments for **mx image alloc()** as follows:

   *MX IMAGE FRAME **frame*
       A pointer to the MX IMAGE FRAME object as described above.

long image_type
> The image type is intended as part of a way to refer to images that are not on the local machine. This feature is not really implemented yet, so for now you should always use the value MXT_IMAGE_LOCAL_1D_ARRAY (1) for this argument.

*long *framesize*
> This is a 1-dimensional two element array containing the width and height of the image in pixels. The first element of the array is the width in pixels, while the second element is the height in pixels.

long image_format
> This argument specifies the greyscale or color format of the image data. The currently supported values for the image format are described in Section 4.1.1.

long pixel_order
> This feature is not yet implemented. Set it to 0.

double bytes_per_pixel
> The number of bytes that corresponds to one pixel. There exist some image formats for which this quantity is not an integer, so we specify it here as a *double*.

size_t header_length
> The length of the image header in bytes. If this length is specified as 0, then no memory will be allocated for an image header.

size_t image_length
> The length of the image data array in bytes. This length must be greater than 0.

RETURN VALUE
> On success, the status code MXE_SUCCESS is returned.

SEE ALSO
> **mx_area_detector_setup_frame**(), **mx_image_free**()

4.3   mx_image_copy_1d_pixel_array

NAME
> **mx_image_copy_1d_pixel_array** - copies the image data array to an application program buffer

SYNOPSIS
> mx_status_type **mx_image_copy_1d_pixel_array** (MX_IMAGE_FRAME *frame,
> > > void *destination_pixel_array,
> > > size_t max_array_bytes,
> > > size_t *num_bytes_copied);

DESCRIPTION
> This function copies the contents of the *image_data* array to an application program supplied buffer. No more than *max_array_bytes* will be copied to the buffer. On return, the *num_bytes_copied* pointer will point to the the number of bytes actually copied to the destination array. This number can be smaller than the requested number of bytes if the actual length of the *image_data* array is shorter than the value of *max_array_bytes*. If you do not need to know the number of bytes copied, you can set the last argument of the function to a NULL pointer.

RETURN VALUE
     On success, the status code MXE_SUCCESS is returned.

SEE ALSO
     mx_image_get_image_data_pointer()

4.4   mx_image_copy_frame

NAME
     mx_image_copy_frame - copies the contents of one MX_IMAGE_FRAME object to another

SYNOPSIS
     mx_status_type **mx_image_copy_frame** (MX_IMAGE_FRAME **new_frame,
                                      MX_IMAGE_FRAME *old_frame);

DESCRIPTION
     This function copies the contents of the existing *old_frame* object to the *new_frame* object, which may or may
     not already exist. This function uses **mx_image_alloc**() internally, so if *new_frame* does not already exist, it will
     be created, while if *new_frame* does already exist, it will be enlarged to contain the copy, if necessary.

RETURN VALUE
     On success, the status code MXE_SUCCESS is returned.

SEE ALSO
     mx_image_alloc()

4.5   mx_image_free

NAME
     mx_image_free - frees an MX_IMAGE_FRAME structure.

SYNOPSIS
     void **mx_image_free** (MX_IMAGE_FRAME *frame);

DESCRIPTION
     This function frees all of the data structures allocated for the specified MX_IMAGE_FRAME object. The object
     must not be used after this function has been invoked.

RETURN VALUE
     mx_image_free does not return a value, since the **free**() function invoked by it does not return a value either.

SEE ALSO
     mx_image_alloc()

4.6   mx_image_get_exposure_time

NAME

    mx_image_get_exposure_time - reports the exposure time of an MX_IMAGE_FRAME

SYNOPSIS

    mx_status_type **mx_image_get_exposure_time** (MX_IMAGE_FRAME *frame,

                             double *exposure_time);

DESCRIPTION

    This function returns the recorded exposure time in seconds for the specified MX_IMAGE_FRAME object. If the header of the frame does not contain the exposure time or the frame was read from a file format that does not store the exposure time, the exposure time will be reported as 1 second.

RETURN VALUE

    On success, the status code MXE_SUCCESS is returned.

4.7   mx_image_get_format_name_from_type

NAME

    mx_image_get_format_name_from_type - converts a numerical image format type to a text representation

SYNOPSIS

    mx_status_type **mx_image_get_format_name_from_type** (long type,

                             char *name,

                             size_t max_name_length);

DESCRIPTION

    This function takes a numerical image format type as defined near the top of the **$MXDIR/include/mx_image.h** header file and converts it into a matching text representation. The function will only copy up to *max_name_length* bytes to the *name* buffer. The text representation returned will be in upper case.

RETURN VALUE

    On success, the status code MXE_SUCCESS is returned.

SEE ALSO

    mx_image_get_format_type_from_name()

4.8   mx_image_get_format_type_from_name

NAME

    mx_image_get_format_name_from_type - converts the name of an image format type to a numerical value

SYNOPSIS

    mx_status_type **mx_image_get_format_type_from_name** (char *name,

                             long type);

DESCRIPTION

This function takes the text representation of an image format and converts it to a numerical image format type as defined near the top of the **$MXDIR/include/mx_image.h** header file. The text representation can be in either upper or lower case.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_image_get_format_name_from_type()

4.9   mx_image_get_frame_from_sequence

NAME

mx_image_get_frame_from_sequence - returns the requested frame from an MX_IMAGE_SEQUENCE object

SYNOPSIS

mx_status_type **mx_image_get_frame_from_sequence** (MX_IMAGE_SEQUENCE *sequence,
                                                                                    long frame_number,
                                                                                    MX_IMAGE_FRAME **image_frame);

DESCRIPTION

This function returns the MX_IMAGE_FRAME object corresponding to the requested *frame_number* from the specified MX_IMAGE_SEQUENCE object.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

4.10   mx_image_get_image_data_pointer

NAME

mx_image_get_image_data_pointer - returns the length and a pointer to the image data for an MX_IMAGE_FRAME object

SYNOPSIS

mx_status_type **mx_image_get_image_data_pointer** (MX_IMAGE_FRAME *frame,
                                                                                    size_t *image_length,
                                                                                    void **image_data_pointer);

DESCRIPTION

This function returns the length of the image and a pointer to the image data for the specified MX_IMAGE_FRAME object. You must know the image format, which can be found at *frame-¿image_format*, in order to successfully manipulate the image data. However, for most area detectors this value will always be the same.

As an example, the AVIEX PCCD-170170 CCD detector always uses a *GREY16* image format, which means that the image data made up of 16-bit greyscale pixels. MX already has a C99-compatible *uint16_t* typedef in the **$MXDIR/include/mx_stdint.h** header file which can be used to manipulate this data format.

WARNING

If an MX imaging function increases the size of the *image_data* array in an MX_IMAGE_FRAME structure, the old *image_data* array will be freed and a new one allocated in its place. In general, the new *image_data* array will be at a different address, which means that an *image_data_pointer* returned by a previous call to **mx_image_get_image_data_pointer()** will no longer be valid. In general, the safest thing to do is to reinvoke **mx_image_get_image_data_pointer()** each time that you need this pointer.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_image_copy_1d_pixel_array()

4.11 mx_image_read_file

NAME

mx_image_read_file - reads an image frame from a disk file into an MX_IMAGE_FRAME object

SYNOPSIS

mx_status_type **mx_image_read_file** (MX_IMAGE_FRAME **frame,
                        unsigned long datafile_type,
                        char *datafile_name);

DESCRIPTION

This function reads an image frame from the disk file specified by the *datafile_name* argument on the user application computer into the specified MX_IMAGE_FRAME. If needed, the MX_IMAGE_FRAME object will be created or resized internally by **mx_image_alloc()**.

If you want to read an image on the detector computer's disk into one of the frame buffers of the detector computer's server, you should be using **mx_area_detector_load_frame()** instead.

WARNING

The *datafile_type* supplied must match the actual data format of the disk file or else **mx_image_read_file()** will fail with an error. **mx_image_read_file()** does not attempt to detect the file format on its own. See Section 4.1.2 for the list of supported datafile formats.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_load_frame(), **mx_image_write_file()**

4.12 mx_image_write_file

NAME

mx_image_write_file - writes the contents of an MX_IMAGE_FRAME object to a disk file

SYNOPSIS

mx_status_type **mx_image_write_file** (MX_IMAGE_FRAME **frame,
                                                    unsigned long datafile_type,
                                                    char *datafile_name);

DESCRIPTION

This function writes the contents of an MX_IMAGE_FRAME to the disk file specified by the *datafile_name* argument on the user application computer. You must specify the file format you want in the *datafile_type* argument. See Section 4.1.2 for the list of supported datafile formats.

If you want to write an image buffer in the detector computer's server to a file on the detector computer's disk, you should be using **mx_area_detector_save_frame()** instead.

RETURN VALUE

On success, the status code MXE_SUCCESS is returned.

SEE ALSO

mx_area_detector_save_frame(), **mx_image_read_file()**

Chapter 5

Utility API Reference

5.1   mx_get_record

NAME

> **mx_get_record** - Get an MX_RECORD object from the MX database.

SYNOPSIS

> MX_RECORD ***mx_get_record** (MX_RECORD *mx_database_record,
>   char *record_name);

DESCRIPTION

> **mx_get_record()** searches the MX database specified by the *mx_database_record* argument for the MX_RECORD
> object that has a name that matches the *record_name* argument. Although it is conventional to supply a pointer
> to the MX list head record called *mx_database*, you can actually supply a pointer to any of the records in the
> running database and it will find the record with the matching name if it exists.

RETURN VALUE

> If successful, **mx_get_record()** returns a pointer to the MX_RECORD object with the specified name. If no
> record with that name exists in the MX runtime database, **mx_get_record()** returns a NULL pointer.

SEE ALSO

> **mx_setup_database()**, **mx_setup_database_from_array()**

5.2   mx_setup_database

NAME

> **mx_setup_database** - configures and initializes the MX runtime database

SYNOPSIS

> mx_status_type **mx_setup_database** (MX_RECORD **mx_database_record,
>   char *database_filename);

DESCRIPTION

    mx_setup_database() is a utility function that does all of the work necessary to create an MX runtime database that is ready to be used by application programs. In outline, **mx_setup_database()** does the following:

- Initialize the MX runtime environment using **mx_initialize_runtime()**.

- Initialize the MX device drivers using **mx_initialize_drivers()**.

- Create an empty MX runtime database using **mx_initialize_record_list()**.

- Setup all of the records in the MX runtime database using **mx_read_database_file()** followed by **mx_finish_database_initiali**

- Initialize connections to the data acquisition hardware and remote servers using **mx_initialize_hardware()**.

After the MX runtime database has been initialized, you may get pointers to individual records in it using the function **mx_get_record()**.

RETURN VALUE

    On success, the status code MXE_SUCCESS is returned.

SEE ALSO

    mx_get_record(), **mx_setup_database_from_array()**

5.3   mx_setup_database_from_array

NAME

    mx_setup_database_from_array - configures and initializes the MX runtime database from an in memory database

SYNOPSIS

    mx_status_type **mx_setup_database_from_array** (MX_RECORD **mx_database_record,
                                              long num_records,
                                              char **record_description_array);

DESCRIPTION

    mx_setup_database_from_array() is a variant of **mx_setup_database()** that reads the database records from an in memory array rather than a disk file.

    See Section 2.2.6 for an example of how to use **mx_setup_database_from_array()**.

RETURN VALUE

    On success, the status code MXE_SUCCESS is returned.

SEE ALSO

    mx_get_record(), **mx_setup_database()**

Appendix A

Using *motor* to Test an MX Area Detector

Motor is a command-line based MX application program used by some beamlines to control many of their experiments. It is not anticipated that users of MX-controlled area detectors will use the **motor** program. However, for initial testing by beamline staff, it may be useful since it implements commands for controlling area detectors. In addition, it is the only significant user application bundled with the core MX distribution.

If you have chosen the name **$MXDIR/etc/motor.dat** for your MX client database and setup the MXDIR environment variable, you can start up the **motor** program by just typing the command `motor` at a Linux/Unix shell prompt or Win32 DOS prompt. The procedure should look something like this:

```
id1:~$ motor

MX version 1.4.0 (October 20, 2006)
WARNING: Save file '/home/lavender/scan.dat' is empty.

motor>
```

You will not be using MX scans, so do not worry about the error message concerning the *scan.dat* file.

If you are using an MX server and the server is not running, you will get a series of errors like the following.

```
id1:~$ motor

MX version 1.4.0 (October 20, 2006)
WARNING: Save file '/home/lavender/scan.dat' is empty.

MXE_NETWORK_IO_ERROR in mx_tcp_socket_open_as_client():
-> connect() to host '192.168.137.3', port 9727 failed.  Errno = 111. >
  Error string = 'Connection refused'.
MXE_NETWORK_IO_ERROR in mxn_tcpip_server_open():
-> The MX server at port 9727 on the computer '192.168.137.3' is either >
  not running or is not working correctly.  You can try to fix this by >
  restarting the MX server.
*** Reconnecting to server 'adserver' at '192.168.137.3', port 9727.
MXE_NETWORK_IO_ERROR in mx_tcp_socket_open_as_client():
```

```
-> connect() to host '192.168.137.3', port 9727 failed.  Errno = 111. >
  Error string = 'Connection refused'.
MXE_NETWORK_IO_ERROR in mxn_tcpip_server_open():
-> The MX server at port 9727 on the computer '192.168.137.3' is either >
  not running or is not working correctly.  You can try to fix this by >
  restarting the MX server.
motor>
```

Some of the lines of output are too wide to fit the page in this manual, so we have artificially broken the lines of output so that all of the output can be seen. The $>$ symbols above mark the places where we have broken the lines. Note that **motor** attempted to reconnect just in case the failure was a momentary failure. This is a general feature of MX clients such that if the connection to the server goes down, the client attempts to reconnect to the server the next time that it wants to send a command to the server.

A.1   Motor Commands

Motor has a large number of commands for controlling a variety of different classes of devices. However, for area detector use, there are really only three commands you should need to know about.

A.1.1   exit

The first command you should know about is the `exit` command which allows you to leave the **motor** program. The procedure should look like this:

```
motor> exit
id1:~$
```

A.1.2   show record

The `show record` command is useful since it allows you to verify that the MX runtime database is working. Using this command should look like this:

```
motor> show record
mx_database      list_head
adserver         tcp_server "192.168.137.3" 9727
ad               network_area_detector 8 0 (0,0,0,0) (0,0,0,0) adserver "ad"
motor>
```

If you see any error messages starting with prefixes like MXE_..., then something is wrong with your configuration and you need to fix it.

A.1.3   area_detector

The `area_detector` command is the most important command that you will use, since it allows you access to much of the functionality of the area detector. You can get help for the `area_detector` command by typing it at a **motor** prompt with no arguments. Generally you can abbreviate command names and command arguments to the shortest unique string that matches. In addition, you can use the alias `ad` for `area_detector`.

Thus, if your area detector is named, for example, `aviex`, you can abbreviate a command like

```
    area_detector aviex set one_shot_mode 2.5
```

down to something like this

```
    area aviex set one 2.5
```

or something even shorter like this

```
    ad aviex se o 2.5
```

NOTE: If the first thing you want to do with a new detector is take an image with it, the quickest place to start is with the snap command described below.

The following sections describe the commands listed by the area detector command's help message with explanatory comments inserted. Most of these commands are one-to-one matches to the MX Area Detector API described in Chapter 3.

Configuration Commands

```
motor> ad
Usage:
  area_detector 'name' get bytes_per_frame
  area_detector 'name' get bytes_per_pixel
  area_detector 'name' get format
  area_detector 'name' get framesize
  area_detector 'name' set framesize 'x_framesize' 'y_framesize'
  area_detector 'name' get maximum_framesize
```

The commands above return information about the current format of image frames in the detector computer.

```
  area_detector 'name' get trigger_mode
  area_detector 'name' set trigger_mode 'trigger mode'
```

The trigger commands are used to switch between internal trigger and external trigger modes.

```
  area_detector 'name' get correction_flags
  area_detector 'name' set correction_flags 'correction flags'
```

The correction flags commands are used to change the list of corrections applied to image frames, using a hexadecimal format for displaying the individual bits in the flags value. For each bit that is set to 1, the corresponding correction will be applied. The bit values are defined in Section 3.1.2.

```
  area_detector 'name' get property_double 'property_name'
  area_detector 'name' set property_double 'property_name' 'property_double'
  area_detector 'name' get property_long 'property_name'
  area_detector 'name' set property_long 'property_name' 'property_long'
  area_detector 'name' get property_string 'property_name'
  area_detector 'name' set property_string 'property_name' 'property_string'
```

Each area detector has internal parameters that are specific to that model of area detector. The area detector *property* commands are used to read and write these values. Each internal property is first given an ASCII name. Then, if the property has a value that can be represented as a long integer, then the *property_long* commands can be used to change it. Similarly, if the property has a value that can be represented as a double, then the *property_double* commands can be used. Otherwise, you must use the *property_string* commands.

   Note: Documentation for these properties has not yet been written.

Sequence Commands

```
area_detector 'name' get sequence_parameters
area_detector 'name' set one_shot_mode 'exposure time in seconds'
area_detector 'name' set continuous_mode 'exposure time in seconds'
area_detector 'name' set multiframe_mode '# frames 'exposure time' 'gap_time'
area_detector 'name' set circular_multiframe_mode '# frames'
                                           'exposure time' 'gap_time'
area_detector 'name' set strobe_mode '# frames' 'exposure time'
area_detector 'name' set bulb_mode '# frames'
area_detector 'name' set geometrical_mode '# frames'
        'exposure time' 'gap_time' 'exposure multiplier' 'gap multiplier'
```

The sequence commands above can be used to control the type of imaging sequence that is to be performed by the detector. A description of the available sequence types can be found in Section 2.5.

Binsize and ROI Configuration Commands

```
area_detector 'name' get binsize
area_detector 'name' set binsize 'x_binsize' 'y_binsize'
```

The *binsize* commands are used to control the binning of pixels in the area detector. In general, the allowed bin sizes are powers of two. Typically, the X and Y bin sizes have the same values, but not all area detectors require this.

```
area_detector 'name' get roi 'roi_number'
area_detector 'name' set roi 'roi_number' 'xmin' 'xmax' 'ymin' 'ymax'
```

The ROI commands are used to set the boundaries of regions of interest. The implementation of the ROIs is all managed in software on the detector computer, so the maximum number of ROIs is limited only by the configuration of the area detector record in the detector computer's database.

   Please note that the rows and columns specified for the X and Y minima and maxima are included within the returned ROI data. In other words, a command like this

```
                ad aviex set roi 5 1000 2000 200 500
```

will return a 1001 column by 301 row array including columns 1000 and 2000 and rows 200 and 500 in the data returned for ROI 5.

Action Commands

```
area_detector 'name' snap 'exposure_time' 'file_format' 'filename'
```

The `snap` command is the simplest way to get the detector to acquire and correct a single frame and then transfer the frame to a disk file on the client computer. You can choose to save the image in any of the file formats supported by MX. However, at the moment, the only file format that has been implemented already is PNM format.

   Here is an example `snap` command line that commands a 2.5 second exposure and then writes the image to *"myimage.pgm"*.

```
ad aviex snap 2.5 pnm myimage.pgm
```

Internally, the `snap` command selects One-shot mode, starts the detector, waits for the detector to finish, gets the image data from the detector computer, and then writes it to a file on the local disk.

```
  area_detector 'name' take frame
```

The `take frame` command is somewhat different in that it starts the detector in whatever sequencer mode it happens to be in, waits for the sequence to finish, and then transfers the file to the memory of the client. It does not write the file to disk.

```
  area_detector 'name' write frame 'file_format' 'filename'
  area_detector 'name' write roiframe 'file_format' 'filename'
```

The above commands do exactly what they say, namely, write out the contents of either the primary image buffer or the ROI image buffer on the client side.

```
  area_detector 'name' arm
  area_detector 'name' trigger
  area_detector 'name' start
  area_detector 'name' stop
  area_detector 'name' abort
```

The above commands are the low level primitives for starting and stopping the detector.

```
  area_detector 'name' get last_frame_number
  area_detector 'name' get status
  area_detector 'name' get extended_status
  area_detector 'name' get busy
```

The commands above report on the current status of the area detector.

```
  area_detector 'name' get frame 'frame_number'
```

The `get frame` command does an image readout, image correction, and image transfer to the client. The available values for *frame_number* are described in Section 3.1.2.

```
  area_detector 'name' get roiframe 'roi_number'
```

The `get roiframe` command transfers to the client the contents of the specified ROI number. You must have defined the boundaries for this particular ROI before invoking `get roiframe`. If you do not, you may end up with the default ROI boundaries which only contain the single pixel (0,0) in binned coordinates.

```
  area_detector 'name' readout 'frame_number'
```

The `readout` command reads out the requested frame number from the camera hardware into the primary image buffer on the detector computer.

```
area_detector 'name' correct
```

The `correct` command performs mask, bias, dark current, and flood field corrections to the contents of the primary image buffer on the detector computer. A given correction will only be performed if the corresponding bit is set in the area detector correction flags and if a correction frame has been loaded into the corresponding image buffer on the detector computer. You may find more information about this operation in the function descriptions of **mx_area_detector_set_correction_flags()** and **mx_area_detector_correct_frame()** as well as the definition of the frame buffer types used in the correction flag bits in Section 3.1.2.

```
area_detector 'name' transfer 'frame_type'
```

The `transfer` command transfers the contents of the requested frame buffer on the detector computer to the primary image buffer of the client. You may find the definitions of the frame buffer types in Section 3.1.2.

```
area_detector 'name' load frame 'frame_type' 'filename'
area_detector 'name' save frame 'frame_type' 'filename'
area_detector 'name' copy frame 'src_frame_type' 'dest_frame_type'
```

The above commands load frames, save frames, and copy frames between the various image buffers on the detector computer. The frame buffer types are described in Section 3.1.2. The files that frame buffers are loaded from or saved to are found on the detector computer.

```
area_detector 'name' measure dark_current 'measurement_time' '# measurements'
area_detector 'name' measure flood_field 'measurement_time' '# measurements'
```

The above commands measure dark current and flood field images on the detector computer. At the end of the measurement, the resulting image frames are left in the matching dark current or flood field image buffers on the detector computer and are ready to be used immediately for correction of new images as they are acquired. However, the contents of the correction frames will be lost when the detector computer's server shuts down. If you want to preserve the contents of the new dark current and flood field correction frames, you must write them to disk using the `save frame` command described above.

Appendix B

Python

The existing MX bindings for Python, called MP, will be extended soon to include support for the new MX area detector class. The plan is that this support will be used to implement a new area detector GUI for MX.