MX Programmer's Guide

for MX 1.5.0

William M. Lavender

October 18, 2007

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

Contents

Part I

Introduction

The introduction goes here.

Part II

C Application Programming Summary

Chapter 1

API Introduction

The C language application programming interface (API) for MX is defined by the header files in the *mx/libMx* directory of the source distribution with names of the form **mx_*.h**. In these files, any function declared using the macro MX_API is intended to be available to normal user programs. Functions declared in these header files using the macro MX_API_PRIVATE are *not* intended for use by typical user programs. For a binary distribution of MX installed at **/opt/mx**, these files are found in the directory **/opt/mx/include**. There are other header files in the **mx/libMx** directory with names like **d_*.h**, **i_*.h**, and so forth, but they are mostly intended to be used by MX device drivers.

Chapter 2

Common Functions

2.1 MX Record Functions

2.1.1 Setting up an MX database

mx_setup_database() - A simplified MX startup routine that reads an MX database file and then automatically initializes all the hardware connections. Many application programs will only need to call this one function at startup time to be ready to use MX controlled devices.

2.1.2 Finding a record in the database

mx_get_record() - Used to find a record in the database by its name.

2.1.3 Displaying the status of a record

mx_print_structure() - Writes a formatted report of the status of the record to the supplied FILE pointer.

Most simple programs will only use the preceding three functions to manipulate a running MX database. However, if the programmer wants more control over the setting up and shutting down of the database, the following lower level functions are also available.

2.1.4 Setting up an MX database (the long way)

An MX database can also be set up by using the following lower level functions essentially in the order show. **mx_setup_database()** itself is implemented by calling these functions.

mx_initialize_drivers() - Initializes all of the MX device drivers.

mx_initialize_record_list() - Creates an empty MX record list.

mx_read_database_file() - Reads an MX database file and adds the listed MX records to the running database.

mx_finish_database_initialization() - Performs MX database initialization steps that can only be performed after all of the records in the database have been created.

mx_initialize_hardware() - Makes the original connections to the control system hardware and initializes the devices.

2.1.5 Shutting down an MX database

Often, an MX client that does not need to preserve a local copy of database parameters can simply call **exit()** when it finishes running. However, if preserving the local copies is required or if devices that need careful shutdown are in use, the following functions are available.

mx_write_database_file() - Writes some or all of the MX records out to an MX database file.

mx_shutdown_hardware() - Shuts down the connections to the control system hardware.

2.1.6 Creating a single MX record

mx_create_record_from_description() - Uses a character string description of an MX record to create the data structures for the record in the running database.

mx_finish_record_initialization() - Finishes setting up the data structures for the new record.

mx_open_hardware() - Makes the initial connection to the hardware controlled by this record.

mx_write_parms_to_hardware() - Writes initial device settings to the hardware.
*(This function is deprecated. Use **mx_open_hardware()** instead.)*

2.1.7 Creating a record description from a current MX record

mx_create_description_from_record() - Creates a character string description of the record in the format used by MX database files.

2.1.8 Destroying a single MX record

mx_read_parms_from_hardware() - Reads device settings from the hardware.
*(This function is deprecated. Use **mx_close_hardware()** instead.)*

mx_close_hardware() - Shuts down the connection to the hardware.

mx_delete_record() - Deletes the record. **mx_close_hardware()** should be performed first to shutdown the associated hardware.

2.1.9 Communication error recovery

mx_resynchronize_record() - This function attempts to resynchronize communication with a hardware controller. It is intended for use when errors from other functions indicate that the handshake between MX and the remote controller is out of sync. For example, this would be used to resynchronize an RS-232 connection if somehow a few characters were lost.

There are a number of other MX functions that operate on generic records, but they are less commonly used and are not mentioned in this summary. However, they will be described in the API reference document.

2.2 MX Network Functions

2.2.1 Socket functions

At present, TCP sockets and Unix domain sockets are supported.

mx_socket_open_as_client() - Makes a client connection to a TCP server.

mx_socket_open_as_server() - Creates a server TCP socket that is ready for connections from clients.

mx_socket_close() - Closes an already open TCP socket.

mx_socket_is_open() - Checks to see if the specified socket corresponds to an open connection.

mx_socket_mark_as_closed() - Replaces the socket value with a value that indicates that this is an invalid socket.

mx_socket_get_last_error() - Gets the error code corresponding to the last socket error. This returns either *errno* or *WSAGetLastError()* depending on the operating system platform.

mx_socket_strerror() - Returns the error text corresponding to a given socket error code.

mx_socket_check_error_status() - Socket related functions may return any of three different kinds of values for indicating an error, namely NULL, 0, or -1 (INVALID_SOCKET on Win32). This function figures out whether the value returned by a socket function corresponds to an error or not.

mx_socket_get_inet_address() - Converts a character string hostname to a numerical internet address represented as an unsigned long.

2.2.2 MX Network I/O

The MX network I/O functions are used by a number of MX device drivers to communicate with devices controlled by a remote MX server. Examples of such drivers include the *network_motor*, *network_scaler*, *network_scaler*, *network_mca*, *network_rs232*, and *network_gpib* drivers. However, the underlying functions can be used if you want to directly modify individual record fields in a remote MX server.

2.2.2.1 Network Server Functions

In order to use the API for MX network I/O, you must have a pointer to the MX server record that manages the connection to the remote server. If you have started an MX database using one of the variants of **mx_setup_database()**, then all you need to know is the name of the server record. For example, if you know that the server is named *mono_server* in the running database, then all you have to do is something like this:

```
...
server_record = mx_get_record( record_list, "mono_server" );
...
```

If you know the server's hostname and port number, but are not running an MX database, then you can use **mx_connect_to_mx_server()** to start a new MX database with a predefined server record. In this case, you have to do something like this:

```
...
mx_status = mx_connect_to_mx_server( &server_record,
"server.example.com", 9727, 8 );
...
```

where 9727 is the remote port number (this is the standard MX port number), and 8 is the default display precision used to format floating point numbers from MX.

2.2.2.2 Network Fields

The MX network API makes use of MX network field structures. The MX_NETWORK_FIELD structure is defined in a manner equivalent to this:

```
typedef struct {
    char nfname[MXU_RECORD_FIELD_NAME_LENGTH+1];
    MX_RECORD *server_record;
    long record_handle;
    long field_handle;
} MX_NETWORK_FIELD;
```

The members of this structure are

nfname - This is an ASCII representation of the record field name in the remote MX server. An example of this would be *theta.position*.

server_record - This is a pointer to an MX_RECORD in the currently running database that manages the connection to a remote MX server. Typically, this will be a record of type *tcpip_server* or *unix_server*.

record_handle - This is a 32-bit integer that uniquely identifies the record specified by *nfname* in the remote MX server.

field_handle - This is a 32-bit integer that uniquely identifies the field specified by *nfname* in the remote MX server.

The combination of a record handle with a field handle is typically referred to collectively as an MX network field handle.

2.2.2.3 Network Handle Functions

The following are the preferred network handle functions.

mx_network_field_init() - This function initializes the *server_record* and the *nfname* members of the MX_NETWORK_FIELD structure. The *record_handle* and *field_handle* members are not initialized here. Instead, they are initialized by a record handle and field handle lookup procedure in the server that is performed during the first time that the MX_NETWORK_FIELD structure is used by one of the family of **mx_get()** and **mx_put()** functions described below. On subsequent calls, the network API functions use the already cached record and field handles.

mx_network_field_connect() - Performs an immediate lookup of the record and field handles. Normally, calling this function directly is not necessary, since it will automatically be invoked when needed by the functions below.

mx_get() - Reads a single value from a remote MX server using the specified MX network field.

mx_put() - Writes a single value to a remote MX server using the specified MX network field

mx_get_array() - Reads a multidimensional array from a remote MX server using the specified MX network field.

mx_put_array() - Writes a multidimensional array to a remote MX server using the specified MX network field.

There are also older non-handle versions of these functions such as **mx_get_by_name()**, but those functions should be avoided since they perform a record handle lookup and field handle lookup for each and every call.

2.2.2.4 Examples of MX Network I/O

Here is an example of using these functions to read the current position of the motor θ.

```
...
MX_NETWORK_FIELD nf;
...
server_record = mx_get_record( record_list, "mono_server" );

mx_status = mx_network_field_init( &nf, server_record, "theta.position" );

mx_status = mx_get( nf, MXFT_DOUBLE, &theta_position );

fprintf(stderr, "The current position of theta = %g degrees\n",
theta_position );
...
```

For a real program, you would need to check the return values from each of the functions above.

Here is a similar example of writing to the 2-dimensional network field *controller.params*.

```
...
MX_NETWORK_FIELD nf;
long dimension[2];
long params_array[3][2];
...
server_record = mx_get_record( record_list, "mono_server" );

mx_status = mx_network_field_init( &nf, server_record, "controller.params" );

for ( i = 0; i < 3; i++ ) {
    for ( j = 0; j < 3; j++ ) {
        params_array[i][j] = 10 * i + j;
    }
}

dimension[0] = 2;
dimension[1] = 3;

mx_status = mx_put_array( nf, MXFT_LONG, 2, dimension, params_array );

...
```

2.2.3 Foreign Control Systems

2.2.3.1 EPICS

The following functions provide a client-side interface to EPICS Channel Access. They have been repackaged such that they return standard MX status structures.

mx_caget() - Gets a process variable from an EPICS IOC.

mx_caput() - Sends a process variable to an EPICS IOC and waits until the data transfer is complete.

mx_caput_nowait() - Sends a process variable to an EPICS IOC without waiting for the data transfer to be complete.

mx_epics_get_debug_flag() - Finds out whether or not MX debugging of EPICS channel access I/O is turned on.

mx_epics_set_debug_flag() - Turns on or off MX debugging of EPICS channel access I/O.

These functions convert the integer flags used to stand for individual data types between MX data types and either EPICS Channel Access data types or EZCA data types.

mx_epics_convert_mx_type_to_epics_type()

mx_epics_convert_epics_type_to_mx_type()

mx_epics_convert_mx_type_to_ezca_type()

mx_epics_convert_ezca_type_to_mx_type()

The following functions support EPICS groups.

mx_epics_start_group() - Starts an EPICS synchronous group.

mx_epics_end_group() - Ends an EPICS synchronous group.

mx_epics_delete_group() - Deletes an existing synchronous group.

mx_group_caget() - Queues up an EPICS caget to be performed when the group is ended.

mx_group_caput() - Queues up an EPICS caput to be performed when the group is ended.

2.3 MX Utility Functions

2.3.1 Array allocation

mx_allocate_array() - Allocate a multi-dimensional array composed either of intrinsic types like long, double, etc., or of C structures.

mx_free_array() - Frees a multi-dimensional array allocated by **mx_allocate_array()**].

mx_array_add_overlay() - This function takes a 1-dimensional data buffer and constructs a multidimensional array on top of it. This will only work correctly if the 1-dimensional buffer contains values in row rank order using standard C conventions.

mx_array_free_overlay() - This function frees the overlay array created by **mx_array_add_overlay()**, but does not free the 1-dimensional buffer.

2.3.2 Clock functions

The following clock tick handling functions are used by the MX server for the scheduling of events. They use operating system dependent functions like *times()* on several versions of Unix or *timeGetTime()* on Win32. Clock tick values are represented using the MX_CLOCK_TICK C structure.

mx_clock_ticks_per_second() - Reports the number of clock ticks per second.

mx_initialize_clock_ticks() - Performs any operating system dependent initialization required for handling operating system clock ticks.

mx_current_clock_tick() - Gets the current time in units of clock ticks.

mx_convert_seconds_to_clock_ticks() - Converts a time interval in seconds into the equivalent number of clock ticks.

mx_convert_clock_ticks_to_seconds() - Converts a time interval in clock ticks into the equivalent number of seconds.

mx_add_clock_ticks() - Since MX_CLOCK_TICK is a C structure, a function is required to add two clock tick values.

mx_subtract_clock_ticks() - Subtracts one clock tick value from another.

mx_compare_clock_ticks() - Reports whether a given clock tick value corresponds to an earlier or a later time than another clock tick value.

2.3.3 Coprocess handling functions

These functions create a coprocess connected to the current process by a pair of pipes. One pipe is used to send to the coprocess and the other pipe is used to receive from the coprocess.

mx_create_coprocess() - Creates a coprocess using the specified command *argv[]* array.

mx_kill_coprocess() - Shuts down the coprocess.

2.3.4 Delay functions

The following functions implement programmable delays. Applications must be prepared for the possibility that the delay may last longer than requested.

mx_sleep() - Tells the operating system to pause the process for the requested number of seconds.

mx_msleep() - Tells the operating system to pause the process for the requested number of milliseconds.

mx_usleep() - Tells the operating system to pause the process for the requested number of microseconds.

2.3.5 Handle table functions

MX handles are integer numbers that are used as indices into a table of C pointers. This feature exists so that programming languages that cannot directly manipulate MX objects using C pointers can refer to the objects via handle numbers. MX network communications normally uses handles to refer to record fields in remote MX servers.

mx_create_handle_table() - Creates a handle table of the specified size.

mx_resize_handle_table() - Increases the size of an already existing handle table.

mx_delete_handle_table() - Destroys a currently existing handle table.

mx_create_handle() - Adds a new handle to a handle table.

mx_delete_handle() - Removes an already existing handle from a handle table.

mx_get_handle_from_pointer() - Converts a handle into the corresponding C pointer.

mx_get_pointer_from_handle() - Converts a C pointer into the corresponding handle.

2.3.6 Log file support

mx_log_open() - Opens a log file.

mx_log_close() - Closes the log file.

mx_log_message() - Writes a message to the log file.

mx_log_timestamp() - Writes a timestamp to the log file.

2.3.7 Math functions

mx_divide_safely() - Divides one *double* value by another while testing for and avoiding division by zero.

mx_difference() - Computes a relative difference function.

mx_round() - Rounds a *double* value to the nearest *long* value.

mx_round_up() - This function rounds up to the next higher *long* value. A small threshold value is subtracted from the number before rounding to prevent getting the wrong number due to previous roundoff errors.

mx_round_down() - This function rounds down to the next lower *long* value. A small threshold value is added to the number before rounding to prevent getting the wrong number due to previous roundoff errors.

mx_byteswap() - Swaps the low order and high order bytes of a 16 bit integer.

2.3.8 String functions

strlcpy() - A replacement for *strncpy()* that ensures that the string fits into the buffer and is NULL terminated.

strlcat() - A replacement for *strncat()* that ensures that the string fits into the buffer and is NULL terminated.

mx_match() - Performs a simple wildcard match of a pattern to a string. It does not implement full regular expressions.

mx_skip_string_fields() - Skips over the requested number of fields in a character string that are separated by standard MX record field separators (space, tab, newline).

2.3.9 Syslog support

mx_install_syslog_handler() - Arranges for messages from MX output handlers to be sent to the Unix syslog daemon. The output may be configured to be instead of or in addition to messages to *stderr*.

2.3.10 Version number reporting functions

These functions return information about the version of MX that is running.

mx_get_major_version() - For MX version 1.4.1, this would return 1.

mx_get_minor_version() - For MX version 1.4.1, this would return 4.

mx_get_update_version() - For MX version 1.4.1, this would return 1.

mx_get_version_date() - This returns the date of the last modification to the MX library such as "May 15, 2007".

mx_get_version_string() - This returns a complete version string for MX such as "MX version 1.4.1 (May 15, 2007)".

2.3.11 Miscellaneous utility functions

mx_free() - Calls *free()* and then makes sure that the pointer we just freed has been set to NULL.

mx_username() - Returns the alphanumeric name of the current user.

mx_get_max_file_descriptors() - This finds out the maximum number of file descriptors that the current process is allowed to use.

mx_current_time_string() - Returns an alphanumeric representation of the current time. This is essentially the output of *ctime()* with the newline at the end removed.

mx_standard_signal_error_handler() - This function is used by many MX programs to provide a standard response to a program crash.

mx_stack_traceback() - This function does its best to provide a traceback of the function call stack at the time of invocation. The stack traceback may be incorrect if the crash was messy enough to corrupt the stack frames. This function is not supported on all MX platforms.

2.4 User I/O Functions

2.4.1 Keyboard handling functions

These functions are often used to allow user's to interrupt long running operations.

mx_getch() - Gets a single character from the user's keyboard.

mx_kbhit() - Checks to see if the user has typed a character on the keyboard.

2.4.2 User interrupt functions

mx_user_requested_interrupt() - Checks to see if the user has requested to interrupt the operation that is currently in progress.

mx_set_user_interrupt_function() - Can be used to check for user interrupts from an alternate source. Generally used by GUIs to allow an "Abort" button to be set up.

mx_default_user_interrupt_function() - This is the default function used by **mx_user_requested_interrupt()**. It uses **mx_kbhit()** and **mx_getch()**].

2.4.3 Debugging functions

MX_DEBUG() - A C macro that is used extensively to generate debugging messages. It uses a trick to get something equivalent to a varargs macro since ANSI C does not explicitly support varargs macros. See the description in **libMx/mx_util.h** for more information.

mx_set_debug_level() - Sets the current debugging level.

mx_get_debug_level() - Gets the current debugging level.

mx_set_debug_output_function() - Can be used to redirect debugging output to an alternate destination. Generally used by GUIs to capture the text.

mx_debug_default_output_function() - This is the default function used by **MX_DEBUG()** output. It merely sends the output to *stderr*.

2.4.4 Information display functions

mx_info() - Displays an informational message to the user.

mx_info_dialog() - Displays an informational message to the user and waits for the user to acknowledge the message. For text mode clients, the user hits a key on the keyboard, while for GUI clients, the user typically clicks on an "OK" button.

mx_set_info_output_function() - Can be used to redirect informational output to an alternate destination. Generally used by GUIs to capture the text.

mx_info_default_output_function() - This is the default function used by **mx_info()** output. It merely sends the output to *stderr*.

mx_set_info_dialog_function() - Can be used to redirect a user dialog to an alternate destination. Generally used by GUIs to display a dialog box with an "OK" button.

mx_info_default_dialog_function() - This is the default function used by **mx_info_dialog()**]. It merely sends the output to *stderr* and waits for the user to hit a key using **mx_kbhit()**.

2.4.5 Warning message functions

mx_warning() - Displays an warning message to the user.

mx_set_warning_output_function() - Can be used to redirect warning messages to an alternate destination. Generally used by GUIs to capture the text.

mx_warning_default_output_function() - This is the default function used by **mx_warning()** output. It merely sends the output to *stderr*.

2.4.6 Error reporting functions

These functions use an *mx_status_type* structure which contains an error code, the name of the function that the error occurred in, and a programmer specified error message.

mx_error() - Used by most MX functions to report errors to the calling routine. It also displays an error message to the user.

mx_error_quiet() - Used occasionally to report an error back to the calling routine *without* displaying an error message to the user.

mx_strerror() - Converts an MX error code to a verbose error description.

MX_SUCCESSFUL_RESULT - This is a macro that is used by MX functions to return a success status to the calling routine. This almost always takes the form of "`return MX_SUCCESSFUL_RESULT;`" in the source code.

mx_set_error_output_function() - Can be used to redirect error messages to an alternate destination. Generally used by GUIs to capture the text.

mx_error_default_output_function() - This is the default function used by **mx_error()** output. It merely sends the output to *stderr*.

Chapter 3

Device API

3.1 MX Amplifier Functions

mx_amplifier_get_gain() - Get the amplifier gain.

mx_amplifier_set_gain() - Set the amplifier gain.

mx_amplifier_get_offset() - Get the amplifier offset in units of the output voltage.

mx_amplifier_set_offset() - Set the amplifier offset in units of the output voltage.

mx_amplifier_get_time_constant() - Get the amplifier time constant in units of seconds.

mx_amplifier_set_time_constant() - Set the amplifier time constant in units of seconds.

mx_amplifier_get_gain_range() - Reports the minimum and maximum allowed gains for this amplifier.

3.2 MX Analog Input Functions

mx_analog_input_read() - Reports the ADC value in user units.

mx_analog_input_read_raw_long() - Returns the raw value from an ADC as a *long* integer.

mx_analog_input_read_raw_double() - Returns the raw value from an ADC as a *double* value.

mx_analog_input_get_dark_current() - Returns the MX ADC dark current as a *double* value. If the 0x1000 bit is set in the *analog_input_flags* field, then this function returns the dark current per second.

mx_analog_input_set_dark_current() - Sets the MX ADC dark current to the supplied *double* value. If the 0x1000 bit is set in the *analog_input_flags* field, then the supplied value is expected to be the dark current per second.

3.3 MX Analog Output Functions

mx_analog_output_read() - Reads the setpoint for a DAC in user units.

mx_analog_output_write() - Write to a DAC in user units.

mx_analog_output_read_raw_long() - Reads the raw setpoint for a DAC as a *long* integer.

mx_analog_output_write_raw_long() - Sets the raw setpoint for a DAC to the supplied *long* integer.

mx_analog_output_read_raw_double() - Reads the raw setpoint for a DAC as a *double* value.

mx_analog_output_write_raw_double() - Sets the raw setpoint for a DAC to the supplied *double* value.

3.4 MX Area Detector Functions

3.4.1 Image size and format functions

mx_area_detector_get_image_format() - Returns the currently configured image format of the area detector using the MXT_IMAGE_FORMAT definitions from **mx/libMx/mx_image.h**.

mx_area_detector_get_image_format() - Changes the image format for the area detector using the MXT_IMAGE_FORMAT definitions from **mx/libMx/mx_image.h**. This function is not supported for many types of area detectors.

mx_area_detector_get_maximum_framesize() - Returns the maximum allowed framesize for the area detector.

mx_area_detector_get_framesize() - Returns the current framesize for the area detector.

mx_area_detector_set_framesize() - Sets the current framesize for the area detector. If the requested framesize is not available, the new framesize will be the nearest available framesize.

mx_area_detector_get_binsize() - Returns the current binsize for the area detector.

mx_area_detector_set_binsize() - Sets the current binsize for the area detector. If the requested binsize is not available, the new binsize will be the nearest available binsize.

mx_area_detector_get_bytes_per_frame() - Reports the number of bytes per frame for the current image size and format.

mx_area_detector_get_bytes_per_pixel() - Reports the number of bytes per pixel for the current image format. This number is not necessarily an integer.

mx_area_detector_get_bits_per_pixel() - Reports the number of bits per pixel for the current image format. This number is not necessarily a multiple of 8.

3.4.2 Image correction functions

mx_area_detector_get_correction_flags() - Returns a bitmap specifying which individual types of image corrections are currently enabled.

mx_area_detector_set_correction_flags() - Uses the supplied bitmap to determine which individual types of image corrections are currently enabled.

mx_area_detector_measure_correction_frame() - Measures the requested type of correction frame and leaves the frame in the matching image buffer, ready to be used for corrections.

mx_area_detector_measure_dark_current_frame() - Acquires a dark current frame and places the frame in the dark current correction image buffer.

mx_area_detector_measure_flood_field_frame() - Acquires a flood field frame and places the frame in the flood field correction image buffer.

mx_area_detector_get_use_scaled_dark_current_flag() - Gets the current value of the *use_scaled_dark_current_flag* field. If this flag is set, a dark current image acquired for a different measurement time can be used to correct the image by rescaling the pixel values to match the current measurement time of the area detector.

mx_area_detector_set_use_scaled_dark_current_flag() - Assigns a new value to the *use_scaled_dark_current_flag* field.

3.4.3 Imaging sequence functions

mx_area_detector_get_total_acquisition_time() - Returns the amount of time in seconds devoted to image acquisition in the currently configured image sequence.

mx_area_detector_get_total_sequence_time() - Returns the amount of time in seconds required to execute the currently configured image sequence.

mx_area_detector_get_detector_readout_time() - Returns the amount of time in seconds required to read out a single image frame from the currently configured sequence.

mx_area_detector_get_sequence_start_delay() - For some area detectors, it is possible to configure a delay of the start of a sequence. This function returns the value of that delay in seconds.

mx_area_detector_get_sequence_parameters() - Returns a structure containing the currently configured image sequence for the area detector.

mx_area_detector_set_sequence_parameters() - Reconfigures the area detector for the requested image sequence parameters.

mx_area_detector_set_one_shot_mode() - Configures the area detector to acquire a single frame using either an internal or an external trigger.

mx_area_detector_set_continuous_mode() - Configures the area detector to acquire an unending sequence of image frames into the same buffer until told to stop. The sequence can be started using either an internal or an external trigger.

mx_area_detector_set_multiframe_mode() - Configures the area detector to acquire the requested number of image frames into a matching number of image buffers. The sequence can be started using either an internal or an external trigger.

mx_area_detector_set_circular_multiframe_mode() - Configures the area detector to acquire an unending sequence of image frames into the specified number of image buffers. When the last frame buffer is reached, the sequence wraps around to overwrite the first frame buffer. The sequence can be started using either an internal or an external trigger.

mx_area_detector_set_strobe_mode() - Configures the area detector to acquire the requested number of image frames into a matching number of image buffers. The sequence is driven by external trigger pulses. When the next external trigger is received, the next image frame will be acquired.

mx_area_detector_set_bulb_mode() - Configures the area detector to acquire the requested number of image frames into a matching number of image buffers. The external trigger input is used as a gate signal. When the gate is turned on, the area detector starts to acquire the next image in the sequence. The image acquisition for that frame lasts until the gate is turned off. Then, the next frame will begin once the gate is turned back on.
Warning: For the AVIEX PCCD-170170 detector, this sequence mode can only run for 1 frame.

3.4.4 Aviex-specific sequence types

The Aviex PCCD-170170 area detector has three sequence types that are specific to it.

mx_area_detector_set_geometrical_mode() - The first frame in the sequence is acquired using the specified exposure time and the detector waits for the specified gap time before starting the next frame. On subsequent frames, the exposure time and gap time for a given frame are those for the previous frame multiplied by an exposure time multiplier and a gap time multiplier. The sequence can be started using either an internal trigger or an external trigger.

mx_area_detector_set_streak_camera_mode() - In this mode, the area detector acquires a new single line of the image using the requested exposure time for the requested number of lines. Once all the lines have been acquired, the resulting data is returned as one long image. The sequence can be started using either an internal trigger or an external trigger.

mx_area_detector_set_subimage_mode() - In this mode, the full frame of the detector is broken up into several strips called *subimages* which are each acquired at a different time. When all of the requested subimages have been acquired, the resulting data is returned as one long image. The sequence can be started using either an internal trigger or an external trigger.

3.4.5 Trigger configuration functions

mx_area_detector_get_trigger_mode() - Reports whether the detector is configured for internal trigger mode or external trigger mode.

mx_area_detector_set_trigger_mode() - Configures the detector for the requested choice of either internal trigger mode or external trigger mode.

3.4.6 Image acquisition functions

This set of functions is used to control image acquisition and monitor the status of an imaging sequence.

mx_area_detector_arm() - Makes sure that the area detector is ready to execute the previously requested sequence. If the area detector is in *external trigger* mode, the sequence will start when the next external trigger arrives.

mx_area_detector_trigger() - If the detector is in *internal trigger* mode, this function starts the imaging sequence. If the detector is in *external trigger* mode, then this function does nothing.

mx_area_detector_start() - This function is merely a convenience function that calls **mx_area_detector_arm()** followed by **mx_area_detector_trigger()**.

mx_area_detector_stop() - If an imaging sequence is in progress, this function tells the area detector to end the sequence when it has finished acquiring the current frame.

mx_area_detector_abort() - If an imaging sequence is in progress, this function tells the area detector to end the sequence as quickly as possible, even if this means losing the image frame currently being acquired.

mx_area_detector_is_busy() - The function reports whether or not an imaging sequence is in progress.

mx_area_detector_get_last_frame_number() - The function reports the number of the frame buffer containing the most recently acquired frame. If the area detector is currently in continuous or circular multiframe mode, the frame buffer number will wrap back to 0 when the sequence wraps back to the first frame buffer.

mx_area_detector_get_total_num_frames() - Reports the total number of frames acquired since the counter was reset. Usually, this will either be the total number of frames acquired since the MX server was started or it will be the total number of frames acquired since the computer booted.

mx_area_detector_get_status() - Reports the current detector status flags. In general, this will at least include the *busy* flag.

mx_area_detector_get_extended_status() - Reports the current detector status flags as well as the last frame number acquired and the total number of frames acquired. For some area detectors, calling this function will be more efficient than separately calling the individual functions listed above.

3.4.7 Image setup and transfer functions

mx_area_detector_setup_frame() - This function makes sure that an image buffer has been allocated that is large enough to contain the next frame to be acquired. If the image buffer is already large enough, then this function does nothing.

mx_area_detector_readout_frame() - On some systems, the image data is initially saved into an inaccessible buffer either on the imaging board or in kernel memory belonging to the operating system. This function copies the image frame data from the inaccessible system buffer into an MX controlled image buffer in the address space of the MX process. If the initial frame buffer is in memory that is already controlled by MX, then this function does nothing.

mx_area_detector_correct_frame() - This function performs the set of image corrections specified by the correction flags bitmap.

mx_area_detector_transfer_frame() - Transfers the requested type of remote image frame to the local image frame buffer. For network-based systems, this is the step that sends the image data across the network.

mx_area_detector_load_frame() - Tells the MX server to load the requested frame buffer from a file which is located on the server.

mx_area_detector_save_frame() - Tells the MX server to save the requested frame buffer to a file which is located on the server.

mx_area_detector_copy_frame() - Tells the MX server to copy image data from one internal frame buffer to another. This is used for things like moving a recently acquired or loaded image frame to another buffer such as the dark current frame or the flood field frame.

mx_area_detector_get_frame() - Transfers the contents of the requested frame number from the server to the supplied MX_IMAGE_FRAME object.

mx_area_detector_get_sequence() - Transfers the contents of the requested number of image frames on the server to the supplied MX_IMAGE_SEQUENCE object.

3.4.8 ROI functions

mx_area_detector_get_roi() - Returns the boundaries of the specified region of interest.

mx_area_detector_set_roi() - Sets the boundaries of the specified region of interest.

mx_area_detector_get_roi_frame() - Transfers the contents of the requested ROI from the current image frame buffer on the server to the supplied local MX_IMAGE_FRAME object.

3.5 MX Autoscale Functions

In MX, *autoscale* devices are pseudodevices used by the *autoscale_scaler* device driver. Autoscaling scalers set upper and lower limits on their count rate so that they can stay within the most linear part of their dynamic range.

If a string record with the specific name**mx_scan_fault** is defined, MX step scans will check after each step to see if any scan faults have occurred. The contents of the string in the define what scan faults will be checked for. For example, if **mx_scan_fault** contains the string "autoscale:I0", this tells the scan that I0 is an autoscaling scaler and that the scan should check for autoscale faults using that record.

An autoscale fault is declared if the measured intensity of the scaler is not within the upper and lower limits of the scaler record. If the intensity is not between those limits, the autoscaling scaler tells its associated *autoscale* record to make whatever changes to amplifier gains or inserted filters are necessary to bring the raw autoscaled value back within the upper and lower limits. After this is done, the scan step is retried. If the measured intensity is still not between the limits, then the autoscaling scaler will try making further changes to the gain and/or filter settings until the fault is fixed or the maximum or minimum gain or filter settings are reached.

The autoscale record itself contains the logic required to implement the feedback control. You can find more information about autoscale records and other associated records in the "Autoscale Related Pseudoscalers" section and the "Autoscale Devices" chapter of the MX Driver Reference Manual. Those sections include examples of how to set up the appropriate records in your MX database.

Here are the functions used by autoscale devices. The only reason for which you may need to know what these functions do is if you plan to implement a new type of autoscale fault.

mx_autoscale_read_monitor() -

mx_autoscale_get_change_request() -

mx_autoscale_change_control() -

mx_autoscale_get_offset_index() -

mx_autoscale_set_offset_index() -

mx_autoscale_get_offset_array() -

mx_autoscale_set_offset_array() -

mx_autoscale_get_limits() -

mx_autoscale_set_limits() -

mx_autoscale_compute_dynamic_limits() -

FIXME - I need to refresh my memory of what all of these functions do.

3.6 MX CCD Functions

WARNING: The CCD class has been replaced by the new Area Detector class.
The existing *remote_marccd* CCD driver will be replaced by a new *marccd* Area Detector driver at some point in the near future.

3.7 MX Digital Input Functions

mx_digital_input_read() - Read a digital input register.

3.8 MX Digital Output Functions

mx_digital_output_read() - Read the output value for a digital output register.

mx_digital_output_write() - Write to a digital output register.

3.9 MX Encoder Functions

mx_encoder_read() - Read the encoder position.

mx_encoder_write() - Write the encoder position.

mx_encoder_get_overflow_status() - Checks for overflow or underflow.

mx_encoder_reset_overflow_status() - Clears any overflow or underflow flags.

3.10 MX Multichannel Analyzer Functions

3.10.1 Counting Functions

mx_mca_start() - Starts the MCA counting.

mx_mca_stop() - Stops the MCA from counting.

mx_mca_start_without_preset() - Starts the MCA counting without a preset. The MCA will continue counting until it is asked to stop.

mx_mca_start_with_preset() - Starts the MCA counting with a preset. You must specify the type of preset and a value for the preset when you invoke this function.

mx_mca_start_for_preset_live_time() - Starts the MCA counting for a preset live time interval.

mx_mca_start_for_preset_real_time() - Starts the MCA counting for a preset real time interval.

mx_mca_start_for_preset_count() - Starts the MCA counting until the preset count is reached in a user specified window.

mx_mca_is_busy() - Checks to see if the MCA is currently acquiring data.

mx_mca_is_new_data_available() - Checks to see if the MCA spectrum has changed since the last time it was read.

3.10.2 Data Handling Functions

mx_mca_clear() - Clears the spectrum in the MCA.

mx_mca_read() - Reads the spectrum from the MCA.

mx_mca_get_roi_integral() - Gets the integrated counts from the MCA region of interest.

mx_mca_get_roi_integral_array() - Gets an array containing the integrated counts of all regions of interest for the MCA.

mx_mca_get_channel() - Gets the value from the specified MCA channel.

mx_mca_get_real_time() - Get the MCA real time.

mx_mca_get_live_time() - Get the MCA live time.

mx_mca_get_counts() - Gets the integrated number of counts within the MCA preset count window.

mx_mca_get_input_count_rate() - Gets the input count rate in counts per second for the MCA.

mx_mca_get_output_count_rate() - Gets the output count rate in counts per second for the MCA.

3.10.3 Parameter Setting Functions

mx_mca_get_parameter() - Reads the value of one or more MCA parameters from the controller.

mx_mca_set_parameter() - Sets the value of one or more MCA parameters in the controller.

mx_mca_get_preset_type() - Gets the type of preset used by the MCA.

mx_mca_set_preset_type() - Sets the type of preset used by the MCA.

mx_mca_get_preset_count_region() - Gets the boundaries of the current preset count region of the MCA.

mx_mca_set_preset_count_region() - Sets the boundaries of the preset count region of the MCA.

mx_mca_get_roi() - Gets the boundaries of the specified MCA region of interest.

mx_mca_set_roi() - Sets the boundaries of the specified MCA region of interest.

mx_mca_get_roi_array() - Gets the boundaries of all regions of interest for the MCA.

mx_mca_set_roi_array() - Sets the boundaries of all regions of interest for the MCA.

mx_mca_get_num_channels() - Gets the number of channels to be used by the MCA.

mx_mca_set_num_channels() - Sets the number of channels to be used by the MCA.

3.10.4 Soft ROI Functions

The soft ROI functions are used to provide an alternate way of implementing region of interest functionality for multichannel analyzers. Soft ROIs do not depend on the presence of hardware ROIs in the MCA. Instead, the ROI integrals are computed within the associated MX driver itself. Typically, this will be used for MCAs that do not provide hardware ROIs. However, there is nothing stopping you from using soft ROIs with MCAs that also have hardware ROIs. For example, you may want to do this if the hardware only has one hardware ROI, but you want more than one ROI for your measurement. Typically, the processing of soft ROIs will be slower than that for hardware ROIs since the entire MCA spectrum must be read into the MX driver, so that the MX driver can compute the integral itself.

Keep in mind that the hardware ROIs and the software ROIs have no connection to each other. For example, in general, hardware ROI 4 will have different limits and a different integral from software ROI 4.

mx_mca_get_soft_roi() - Gets the boundaries of the specified software region of interest.

mx_mca_set_soft_roi() - Sets the boundaries of the specified software region of interest.

mx_mca_get_soft_roi_integral() - Gets the integrated counts from the software region of interest.

mx_mca_get_soft_roi_integral_array() - Gets an array containing the integrated counts of all software regions of interest for the MCA.

3.10.5 Energy Scale Functions

These functions can be used to handle the MCA X axis in units of energy rather than channel number.

mx_mca_get_energy_scale() - Gets the scale factor for converting from MCA channel number to energy.

mx_mca_set_energy_scale() - Sets the scale factor for converting from MCA channel number to energy.

mx_mca_get_energy_offset() - Gets the offset for converting from MCA channel number to energy.

mx_mca_set_energy_offset() - Sets the offset for converting from MCA channel number to energy.

mx_mca_get_energy_axis_array() - Returns an array that contains the energy corresponding to each MCA channel number.

3.11 MX Multichannel Encoder Functions

mx_mce_read() - Reads the position values from a multichannel encoder.

mx_mce_get_overflow_status() - Checks to see if the MCE has overflowed.

mx_mce_reset_overflow_status() - Clears the overflow status of an MCE.

3.12 MX Multichannel Scaler Functions

3.12.1 Counting Functions

mx_mcs_start() - Starts the MCS counting.

mx_mcs_stop() - Stops the MCS from counting.

mx_mcs_is_busy() - Checks to see if the MCS is counting.

3.12.2 Data Handling Functions

mx_mcs_clear() - Clears the data in the MCS.

mx_mcs_read_all() - Reads all values from the MCS.

mx_mcs_read_scaler() - Reads the values from the specified scaler channel in the MCS.

mx_mcs_read_measurement() - Reads all the scaler values from the specified measurement in the MCS.

mx_mcs_read_timer() - Reads the timer channel from the MCS.

3.12.3 Parameter Setting Functions

mx_mcs_get_mode() - Gets the counting mode for the MCS.

mx_mcs_set_mode() - Sets the counting mode for the MCS.

mx_mcs_get_measurement_time() - Gets the preset measurement time for the MCS.

mx_mcs_set_measurement_time() - Sets the preset measurement time for the MCS.

mx_mcs_get_measurement_counts() - Gets the preset measurement count for the MCS.

mx_mcs_set_measurement_counts() - Sets the preset measurement count for the MCS.

mx_mcs_get_num_measurements() - Gets the number of measurements for the MCS.

mx_mcs_set_num_measurements() - Sets the number of measurements for the MCS.

3.13 MX Motor Functions

3.13.1 Motor status functions

mx_motor_get_position() - Gets the motor position in user units.

mx_motor_get_status() - Gets a 32-bit set of status flags that report the current state of the motor axis. These bits include:

busy - is a move in progress?
positive limit hit
negative limit hit
home search succeeded
following error
drive fault
axis disabled
open loop
controller disabled
soft positive limit hit
soft negative limit hit
software error

mx_motor_get_extended_status() - This function reports both the position of the motor and the motor status flags in one call. In many situations, calling this function is more efficient performing than two separate calls to **mx_motor_get_position()** and **mx_motor_get_status()**.

3.13.2 Alternate motor status functions

These functions report the state of only a single bit in the status flag described above.

mx_motor_is_busy() - Check to see if a commanded move is still in progress.

mx_motor_positive_limit_hit() - Checks to see if the positive limit switch has been hit.

mx_motor_negative_limit_hit() - Checks to see if the negative limit switch has been hit.

3.13.3 Motor motion functions

mx_motor_move_absolute() - Moves the motor to an absolute position in user units.

mx_motor_move_relative() - Moves the motor by a relative distance in user units.

mx_motor_array_move_absolute() - Moves an array of motors to absolute positions in motor units.

mx_wait_for_motor_stop() - Waits for a motor move to complete. Use one of the motor status functions if you want to check the motion status without blocking.

mx_wait_for_motor_array_stop() - Waits for an array of motors to stop moving.

3.13.4 Motor abort functions

mx_motor_soft_abort() - Tells the motor to stop moving in a "nice" fashion so that it does not lose track of its position.

mx_motor_immediate_abort() - Tells the motor to stop moving as soon as it can.

3.13.5 Setting the motor position

mx_motor_set_position() - Sets the motor position in the controller to the requested value, if this is possible.

Many controllers have restrictions on what values the position can be set to or do not allow the position to be set at all. A common restriction is to only allow the current position to be set to zero.

3.13.6 Home search

mx_motor_find_home_position() - Performs a home search in the requested direction.

mx_motor_is_at_home_switch() - Finds out whether the motor is currently at the home switch.

3.13.7 Continuous moves

mx_motor_constant_velocity_move() - Moves the motor in the requested direction until the motor move is aborted or until the motor reaches a hardware or software limit.

3.13.8 Motor speeds

mx_motor_get_speed() - Get the programmed motor speed from the controller.

mx_motor_set_speed() - Set the programmed motor speed in the controller.

mx_motor_get_base_speed() - Get the programmed motor base speed from the controller.

mx_motor_set_base_speed() - Set the programmed motor base speed in the controller.

mx_motor_get_maximum_speed() - Get the programmed maximum speed from the controller.

mx_motor_set_maximum_speed() - Set the programmed maximum speed in the controller.

mx_motor_save_speed() - Saves the current programmed motor speed so that it can be restored at a later date.

mx_motor_restore_speed() - Restores the motor speed most recently set by **mx_motor_save_speed()**.

mx_motor_set_speed_between_positions() - Sets the motor controller's speed such that it will travel between the two requested positions in the requested amount of time. This function is more general than **mx_motor_set_speed()**] since it will also work for pseudomotors whose positions are non-linear functions of the underlying real motor. However, the function works for real motors as well.

3.13.9 Motor acceleration

mx_motor_get_acceleration_type() - Used to find out what type of acceleration parameters are used by this controller.

mx_motor_get_acceleration_parameters() - Get acceleration parameters used by the controller.

mx_motor_set_acceleration_parameters() - Set acceleration parameters used by the controller.

mx_motor_get_acceleration_time() - Calculates the total time required for the motor to accelerate from zero to the programmed speed for the motor.

mx_motor_get_acceleration_distance() - Calculates the total distance required for the motor to accelerate from zero to the programmed speed for the motor.

Motor acceleration is specified in a variety of different ways by different controllers such as steps/sec**2, acceleration time, acceleration slope, and so forth. Currently, MX uses the form of the acceleration parameters that is native to each controller rather than trying to force them all to conform to one particular scheme such as steps/sec**2.

3.13.10 Pseudomotor position calculation functions

mx_motor_compute_pseudomotor_position_from_real_position() - This function takes the supplied real motor position and converts it to the corresponding pseudomotor position.

mx_motor_compute_real_position_from_pseudomotor_position() - This function takes the supplied pseudomotor position and converts it to the corresponding real motor position.

3.13.11 Scan related functions

mx_motor_compute_extended_scan_range() - Uses the provided motor scan range to calculate an extended scan range that adds enough acceleration and deceleration distance such that the motor will already be moving at the slew speed when it reaches the start of the originally specified scan range.

3.13.12 Getting and setting other motor parameters

mx_motor_get_parameter() - Reads the value of one or more motor parameters from the controller.

mx_motor_set_parameter() - Sets the value of one or more motor parameters in the controller.

This pair of functions is used to get and set other parameters that a given record type may define for itself. The speed and acceleration support above is built on top of these two functions. A variety of other parameters such as PID servo loop gains can be controlled via these functions.

3.14 MX Relay Functions

mx_relay_command() - Opens or closes the relay.

mx_get_relay_status() - Checks to see if the relay is open or closed.

3.15 MX Scaler Functions

3.15.1 Preset time functions

mx_scaler_read() - Reads the current value of the scaler without stopping it.

mx_scaler_clear() - Stops the scaler, clears its value and rearms the scaler.

mx_scaler_read_and_clear() - Combines the two operations above.

mx_scaler_overflow_set() - Checks to see if the scaler has overflowed its maximum value.

3.15.2 Preset count functions

mx_scaler_start() - Start the scaler counting to a preset value. The scaler is assumed to generate a gate signal while it is counting.

mx_scaler_stop() - Stop the scaler from counting and removes the gate signal.

mx_scaler_is_busy() - Is the scaler still counting?

3.15.3 Counter mode functions

mx_scaler_get_mode() - Finds out whether the scaler is configured to be in preset time or preset count compatible mode.

mx_scaler_set_mode() - Sets the counter mode.

mx_scaler_set_modes_of_associated_counters() - Ensures that the other counter channels in a module with multiple counters are set to counter modes that are compatible with the given scaler.

3.15.4 Dark current functions

mx_scaler_get_dark_current() - Gets the dark current for the given scaler in counts per second. This does not actually perform a dark current measurement. It merely returns the previous value that was set by the **mx_scaler_set_dark_current()** function.

mx_scaler_set_dark_current() - Sets the dark current for the given scaler in counts per second.

3.16 MX Table Functions

This set of functions controls a diffractometer/goniostat table so that the table can make pure rotations and tranlations relative to an arbitrary point on the table.

mx_table_get_position() - Gets the current position of the specified table pseudomotor.

mx_table_set_position() - Sets the current position of the specified table pseudomotor.

mx_table_move_absolute() - Moves the specified table pseudomotor to the requested position.

mx_table_is_busy() - Checks to see if the specified table pseudomotor is currently moving.

mx_table_soft_abort() - Tells the table pseudomotor to stop moving in a "nice" fashion so that it does not lose track of its position.

mx_table_immediate_abort() - Tells the table pseudomotor to stop moving as fast as it can.

mx_table_positive_limit_hit() - Checks whether the limit for the specified table pseudomotor has been hit.

mx_table_negative_limit_hit() - Checks whether the limit for the specified table pseudomotor has been hit.

3.17 MX Timer Functions

3.17.1 Preset time functions

mx_timer_start() - Starts the timer counting for the specified number of seconds.

mx_timer_stop() - Stops the timer from counting and removes the gate signal.

mx_timer_is_busy() - Is the timer still counting?

3.17.2 Preset count functions

mx_timer_read() - Reads the elapsed time in seconds without stopping the timer.

mx_timer_clear() - Stops the timer, clears its value and rearms the timer.

3.17.3 Timer mode functions

mx_timer_get_mode() - Finds out whether the timer is configured to be in preset time or preset count compatible mode.

mx_timer_set_mode() - Sets the timer mode.

mx_timer_set_modes_of_associated_counters() - Ensures that the other counter channels in a module with multiple counters are set to counter modes that are compatible with the given timer.

Chapter 4

Interface API

4.1 MX CAMAC Functions

mx_camac() - Sends a CAMAC FNA command to the crate controller.

mx_camac_qwait() - Repeatedly sends a CAMAC FNA command to the crate controller until Q=1 is received. Needed by E500 motor controllers.

mx_camac_send_control_word() - Sends a Z, C, or I command to the crate controller.

mx_camac_get_lam_status() - Checks the Look At Me (LAM) status for the crate controller.

4.2 MX GPIB Functions

mx_gpib_open_device() - Makes a connection to the GPIB device at the specified GPIB address.

mx_gpib_close_device() - Shuts down the connection to the GPIB device at the specified GPIB address.

mx_gpib_read_parms_from_device() - Reads several GPIB parameters from the device at the specified GPIB address, such as the secondary address, I/O timeout, EOI mode, and EOS characters.

mx_gpib_write_parms_to_device() - Writes the GPIB parameters to the device at the specified GPIB address.

mx_gpib_read() - Reads up to a specified number of characters from the GPIB interface.

mx_gpib_write() - Writes up to a specified number of characters to the GPIB interface.

mx_gpib_getline() - Gets a line of text from the GPIB interface.

mx_gpib_putline() - Sends a line of text to the GPIB interface.

mx_gpib_interface_clear() - Sends an interface clear signal to the GPIB bus.

mx_gpib_device_clear() - Send a device clear (DCL) signal to all the devices on the GPIB bus.

mx_gpib_selective_device_clear() - Sends a selective device clear (SDC) signal to a particular device on the GPIB bus.

mx_gpib_local_lockout() - Perform a local lockout (LLO) on the GPIB bus.

mx_gpib_remote_enable() - Enable remote control for the specified device.

mx_gpib_go_to_local() - Changes the specified device to local mode (GTL).

mx_gpib_trigger() - Triggers the specified device (GET).

mx_gpib_wait_for_service_request() - Waits until an SRQ signal is received.

mx_gpib_serial_poll() - Performs a serial poll on the specified device (SPE and serial poll).

mx_gpib_serial_poll_disable() - Disables serial polling (SPD).

4.3 MX Portio Functions

These functions are used to access I/O ports on x86 platforms.

mx_portio_inp8() - Reads an 8 bit value from an I/O port.

mx_portio_inp16() - Reads a 16 bit value from an I/O port.

mx_portio_outp8() - Writes an 8 bit value to an I/O port.

mx_portio_outp16() - Writes a 16 bit value to an I/O port.

mx_portio_request_region() - Asks the operating system for access to the specified range of I/O ports.

mx_portio_release_region() - Releases control of the specified range of I/O ports back to the operating system.

4.4 MX RS-232 Functions

mx_rs232_getchar() - Gets a single character from the specified serial interface.

mx_rs232_putchar() - Sends a single character to the specified serial interface.

mx_rs232_read() - Reads up to a specified number of characters from the serial interface.

mx_rs232_write() - Writes up to a specified number of characters to the serial interface.

mx_rs232_getline() - Gets a line of text from the serial interface.

mx_rs232_putline() - Sends a line of text to the serial interface.

mx_rs232_input_is_available() - Checks to see if there are any characters in the input buffer for the serial interface.

mx_rs232_discard_unread_input() - Discards the contents of the input buffers for the serial interface.

mx_rs232_discard_unwritten_output() - Discards any characters that are still in the output buffers for the serial interface.

4.5 MX Generic Interface Functions

Warning: The Generic Interface Functions are deprecated, at least in their current form.

The *generic* interface class is essentially a catchall for various types of interfaces that do not fit somewhere else. It is quite common for motor drivers (which control a single motor) to have an associated generic interface driver which manages all of the axes belonging to the controller as a whole. Typically, if a driver needs to have a place to put information about all the resources controlled by a controller as a whole, the best place to put these is in a generic interface driver. Thus, for a Compumotor 6K motor controller, each axis has its own separate motor record, but the controller as a whole has a generic interface record which is used to manage sending and receiving ASCII commands to and from the controller.

Many, but not all, generic interface drivers have ways of sending and receiving ASCII strings. For those that do, the following functions are available. Please note that **mx_generic_getline()** and **mx_generic_putline()**] do not exist yet, but are scheduled to be added to the library.

mx_generic_getchar() - Gets a single character from the specified generic interface.

mx_generic_putchar() - Sends a single character to the specified generic interface.

mx_generic_read() - Reads up to a specified number of characters from the generic interface.

mx_generic_write() - Writes up to a specified number of characters to the generic interface.

mx_generic_input_is_available() - Checks to see if there are any characters in the input buffer for the generic interface.

mx_generic_discard_unread_input() - Discards the contents of the input buffers for the generic interface.

mx_generic_discard_unwritten_output() - Discards any characters that are still in the output buffers for the generic interface.

Chapter 5

MX Variable Functions

5.1 Generic variable functions

mx_send_variable() - Sends the value field of a local variable record to the remote server.

mx_receive_variable() - Receives the value field of a local variable record from the remote server.

mx_get_variable_parameters() - Gets the dimensionality and variable type of a local variable record.

mx_get_variable_pointer() - Gets a pointer to the data values for a local variable record.

5.2 One-dimensional variable functions

Since one-dimensional arrays are more common than higher-dimensional arrays, there are a number of convenience functions specifically for working with them.

mx_get_1d_array() - For a given one-dimensional variable record pointer, this function verifies that the variable is the correct data type. It then gets both the number of elements and a pointer to the data values.

mx_get_1d_array_by_name() - For a given MX record list pointer, this function tries to find a matching one-dimensional variable record with the listed name and data type. It then returns the number of elements and a pointer to the data values.

mx_test_1d_array() - Verifies that the specified record is a one-dimensional variable record with the requested field type and number of elements.

5.3 Scalar variable functions

Several convenience functions for scalar values are available. The set of available convenience functions is incomplete at the moment. I will fill it out in the near future.

The following *get* functions require a record list pointer and a record name.

mx_get_long_variable_by_name()

mx_get_unsigned_long_variable_by_name()

mx_get_double_variable_by_name()

mx_get_string_variable_by_name()

The following *set* function requires a pointer to the MX record.

mx_set_long_variable()

Chapter 6

Server API

6.1 MX Server Functions

mx_get() - This is a macro that gets the value of a zero dimensional value (a scalar) from a remote MX server.

mx_put() - This is a macro that sends the value of a zero dimensional value (a scalar) to a remote MX server.

mx_get_array() - Receives the value of an MX record field from a remote MX server.

mx_put_array() - Sends a local value to an MX record field on a remote MX server.

mx_network_send_message() - Sends an MX protocol message to an MX server.

mx_network_receive_message() - Receives an MX protocol message from an MX server.

mx_socket_send_message() - This is a low level function that sends an MX protocol message on the specified socket.

mx_socket_receive_message() - This is a low level function that receives an MX protocol message on the specified socket.

mx_socket_send_error_message() - This is a low level function that sends an MX protocol error message on the specified socket.

Chapter 7

MX Scan Functions

There is only one scan record specific function available, namely,

mx_perform_scan() - Executes the specified scan from the MX database.

It is the only function that an application program using standard scan drivers should need to invoke.

7.1 Scan driver functions

There are a variety of other scan related functions that are mostly intended for use by *scan device drivers*, rather than application programs. These functions are normally invoked behind the scenes by *mx_perform_scan()*, but a custom scan driver is *not required* to use any of them if it has special needs. The functions are passed pointers to MX_SCAN structures rather than pointers to MX_RECORD functions.

mx_standard_prepare_for_scan_start() - This is the default routine that handles standard types of preparation before the beginning of a scan such as opening datafiles and starting plotting programs.

mx_standard_cleanup_after_scan_end() - This is the default routine that closes down all of the resources acquired by **mx_standard_prepare_for_scan_start()**.

mx_setup_scan_shutter() - This routine checks to see if the scan is using an X-ray shutter and initializes the associated data structures.

mx_setup_scan_enable() - This routine checks to see if the scan will be using a scan enable/disable interlock and initializes the associated data structures.

mx_scan_handle_data_measurement() - When it is time to perform a measurement, this routine invokes the function used by this scan's measurement type to perform measurements.

mx_scan_handle_pause_request() - When the user requests that a scan be paused, this function passes control to the installed pause request handler.

mx_set_scan_pause_request_handler() - Allows an application program to redefine the default pause request handler.

mx_default_scan_pause_request_handler() - This function merely prints out a message that a pause has been requested.

mx_scan_display_scan_progress() - This function displays the current status of the scan to the user. Typically, it is used to present the results of the most recent measurement.

mx_clear_scan_input_devices() - This function makes the current scan's input devices ready to make the next measurement.

mx_handle_abnormal_scan_termination() - This function is invoked any time a scan is abnormally terminated, both for user requested aborts and for failures in the beamline hardware such as hitting a limit switch.

mx_scan_restore_speeds() - Restores all the motor speeds to the values they had before the start of the scan.

mx_log_scan_start() - If scan logging is enabled, this function writes a "start of scan" message to the scan log.

mx_log_scan_stop() - If scan logging is enabled, this function writes an "end of scan" message to the scan log.

mx_setup_topup_interlock() - This is an APS specific scan function that sets up an scan interlock with the Advanced Photon Source topup injection. This function tries to ensure that measurements are not made during a topup injection and retakes a measurement if necessary.

mx_scan_finish_record_initialization() - Performs initialization steps that must be performed for all scan records. It is intended to be used in the **xxx_finish_record_initialization** driver function for a particular scan type.

7.2 Measurement driver functions

Each type of measurement that an MX scan record can perform such as preset time, preset count, etc. has an associated measurement type driver. These functions are passed pointers to MX_MEASUREMENT structures.

mx_measure_data() - This function performs the detector measurements required for the current step in the scan.

mx_configure_measurement_type() - Before a scan starts, this function reconfigures the devices used by this scan to be ready for the requested measurement type. This may involve things like reconfiguring which devices have presets and so forth.

mx_deconfigure_measurement_type() - After a scan ends, the devices involved in the scan are reprogrammed to revert back to the default measurement type, namely, preset time.

mx_prescan_measurement_processing() - Performs any measurement related functions required before the first data point is taken.

mx_postscan_measurement_processing() - Performs any measurement related functions required after the last data point is taken.

mx_preslice_measurement_processing() - For a multidimensional scan, this function performs any measurement related functions required before the first data point of the current slice is taken.

mx_postslice_measurement_processing() - For a multidimensional scan, this function performs any measurement related functions required after the last data point of the current slice is taken.

7.3 Data file driver functions

Each type of data file format that can be generated by an MX scan record has an associated data file driver. These functions are passed pointers to MX_DATAFILE structures.

7.3.1 Data file handling functions

mx_datafile_open() - Opens the file that the data will be written to.

mx_datafile_close() - Closes the file that the data have been written to.

mx_datafile_write_main_header() - This function writes out the header information required by this data file type.

mx_datafile_write_section_header() - For data files that are broken up into sections, this function writes out the header for the current section.

mx_datafile_write_trailer() - This function writes out any trailer information required by this data file type.

mx_add_measurement_to_datafile() - This function goes to each device in turn and copies the data for this datapoint to the data file.

mx_add_array_to_datafile() - For some scan types, the results of the measurement are already located in arrays in the memory of the MX application program. For this case, this function copies the contents of those arrays to the data file.

7.3.2 Data file version number functions

An MX scan record will autoincrement the version number for the next scan data file, if and only if, everything after the last period '.' in the filename consists only of the digits '0-9'. The following functions handle the job of autoincrementing the number.

mx_parse_datafile_name() - This function checks to see if the current data file name has a version number and splits the file name into the version number and the version independent part.

mx_construct_datafile_version_number() - Constructs a new data file version number from the old version number. This function wraps back to zero if the next version number would increase the number of digits in the version number. For example, if the most recently created data file had version number '.999', the next version number will be '.000'.

mx_update_datafile_name() - This function replaces the old version number with the new version number.

7.4 Plot driver functions

Each type of scan progress plotting program that can be used by an MX scan record has an associated plot driver. These functions are passed pointers to MX_PLOT structures.

mx_plot_open() - Initializes the connection to the plotting program.

mx_plot_close() - Shuts down the connection to the plotting program.

mx_add_measurement_to_plot_buffer() - This function goes to each device in turn and copies the data for this datapoint to the plot buffer.

mx_add_array_to_plot_buffer() - For some scan types, the results of the measurement are already located in arrays in the memory of the MX application program. For this case, this function copies the contents of those arrays to the plot buffer.

mx_display_plot() - This function tells the plotting program to update its scan progress windows to include all the measurements that have been sent to it.

mx_plot_set_x_range() - Sets the X axis limits of the scan progress window.

mx_plot_set_y_range() - Sets the Y axis limits of the scan progress window.

mx_plot_start_plot_section() - For multidimensional scans, MX at present only supports displaying the current one-dimensional slice of the scan in the scan progress window. **mx_plot_start_plot_section()** tells the plotting program that a new one-dimensional slice has just started.

mx_plotting_is_enabled() - Checks to see if the user has requested that a scan progress graph be displayed.

mx_set_plot_enable() - This function sets the flag that is used by **mx_plotting_is_enabled()**].