MX Driver Reference Manual

William M. Lavender

October 20, 2007

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

Contents

Chapter 1

Introduction

MX is a portable beamline control and data acquisition toolkit currently in use at a number of synchrotron beamlines and laboratory X-ray generator systems. The purpose of this manual is to explain how to set up the configuration files that control how MX works.

The most important step in configuring a new MX system is the building of the MX configuration files which are generally found in the directory *$MXDIR/etc* where *MXDIR* is an environment variable that points to the top of the MX directory tree. If you use the standard definition that *MXDIR = /opt/mx*, then the configuration files will be found in */opt/mx/etc*. The most important configuration files are:

/opt/mx/etc/motor.dat	- The primary MX client-side database file.
/opt/mx/etc/mxserver.acl	- An access control list for the MX server.
/opt/mx/etc/mxserver.dat	- The MX server-side database file.
/opt/mx/etc/mxupdate.dat	- The *mxupdate* process's configuration file.

We will cover *mxserver.acl* and *mxupdate.dat* first since they are simple and relatively easy to explain.

1.1 *mxserver.acl*

mxserver.acl is an access control list file that describes what computers are allowed to connect to the local MX server. It is a simple text file with one entry per line. The entries are either IP addresses or Internet domain names. I recommend the use of IP addresses since then the MX server does not need to perform potentially time consuming DNS lookups to find the IP address from the domain name, but it is your choice. Here is an example *mxserver.acl* file:

```
127.0.0.1
192.168.17.5
192.168.17.27
192.168.22.*
myhost.example.com
*.example.net
```

This configuration files says that the individual IP addresses 127.0.0.1, 192.168.17.5 and 192.168.17.27 are allowed, that any host on the subnet 192.168.22.* is allowed, that *myhost.example.com* is allowed, and that any host in

the domain **.example.net* is allowed. You will note that * wildcards are allowed. In addition, the *?* wildcard, which stands for a single character, is also allowed although it is not as easy to use.

At present, MX only does access control on a host level basis and allows any username from a given machine to connect. Although MX clients do transmit their username to the remote MX server, this information is not used for access control since it is trivially spoofed. Support for user level access control will be added once something along the lines of an SSL/TLS certificate infrastructure has been added to MX.

1.2 *mxupdate.dat*

The *mxupdate* process exists to save and restore MX database variables so that their values can be preserved when the MX server is stopped and restarted. It is another simple text file that contains the name of one record field per line. An example *mxupdate.dat* file looks like:

```
edge_energy.value     1 0
d_spacing.value       1 0
beam_offset.value     1 0
shutter_policy.value  1 0
xafs_header1.value    1 0
xafs_header2.value    1 0
xafs_header3.value    1 0
sff_header1.value     1 0
sff_header2.value     1 0
sff_header_fmt.value  1 0
theta_enabled.value   1 0
momega_enabled.value  1 0
normpoly_enable.value 1 0
normal_enabled.value  1 0
id_ev_enabled.value   1 0
momega_params.value   1 0
normpoly_params.value 1 0
id_ev_harmonic.value  1 0
id_ev_offset.value    1 0
id_ev_params.value    1 0
```

The first field on each line is the name of an MX record field. For example, *edge_energy.value* refers to the record called *edge_energy* and the field within it called *value*. Any record field listed here will have its value saved twice a minute and will have its value restored the next time the MX server is started from the *mxupdate* state files called */opt/mx/state/mxsave.1* and */opt/mx/state/mxsave.2*. When *mxupdate* restores the values the next time the MX server is started, *mxupdate* will choose whichever of the two state files which is both complete and most recent.

For the present, you should set the second and third fields on each line above to 1 and 0 respectively. These fields are not currently documented, since they are scheduled to be changed in the future.

1.3 The MX Record Database Files *motor.dat* and *mxserver.dat*

The most important configuration files in MX are the client-side *motor.dat* file and the server-side *mxserver.dat* file. These files describe the set of objects called *MX records* which are used to represent the motors, counter, MCAs,

serial ports, and so forth that an MX client or server will manage.

The first thing you will note if you compare *motor.dat* and *mxserver.dat* is that they have exactly the same format. This is due to the fact that MX servers and clients actually use exactly the same set of device drivers. In fact, the only thing that marks a process as being an MX client is the use of one of the device drivers that send command requests across the network to a remote server rather than to a device directly attached to the client process.

For example, suppose you have a Compumotor 6K motor controller that is managed by an MX server and a remote MX client wants to move a motor belonging to the 6K. The MX server will have in its database a record for the motor of type *compumotor* which communicates via a directly attached RS-232 port, while the MX client will have a record of type *network_motor* which sends the request across the network via a socket. But other than the fact that the server or client are using different low level drivers, they treat the motor moves the same.

To make the example more concrete, let us display some example MX configuration files. First, here is an *mxserver.dat* file for an MX server managing a 4-axis Compumotor 6K motor controller:

```
6k_rs232  interface rs232 tty "" "" 9600 8 N 1 S 0x0d0a 0x0d0a /dev/ttyS1
6k interface generic compumotor_int "" "" 6k_rs232 0x0 1 1 4
m1 device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 0.04 0 urad 6k 1 1 1
m2 device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 0.04 0 urad 6k 1 2 1
m3 device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 .0188 0 um 6k 1 3 1
m4 device motor compumotor "" "" 0  0 -200000 200000 0 -1 -1 0.25 0 urad 6k 1 4 1
```

For the moment, we will not go too deeply into the details of the format of the lines. The first thing to note here is that each line of the database file corresponds to one MX record, whose name is the first item on the line. Thus, there are six MX records in this database named *6k_rs232*, *6k*, *m1*, *m2*, *m3*, *m4*. There is nothing special about these record names. All that MX expects is that they be unique, be ordinary printing ASCII characters, and be 15 characters long or less.

Going to a little lower level of detail, the second, third, and fourth fields on each line describe the MX device driver that will be used to control the underlying hardware. The second field is called the *mx_superclass* field, the third field is the *mx_class* field, and the fourth field is the *mx_type* field. Typically the MX driver as a whole will be referred to by the name of the *mx_type* field. For the example database above, this breaks down as follows:

Record Name	Superclass	Class	Type	Explanation
6k_rs232	interface	rs232	tty	This record uses the *tty* driver for Posix serial ports.
6k	interface	generic	compumotor_int	This record manages the 6K controller as a whole.
m1	device	motor	compumotor	This record manages axis 1 of the 6K controller.
m2	device	motor	compumotor	This record manages axis 2 of the 6K controller.
m3	device	motor	compumotor	This record manages axis 3 of the 6K controller.
m4	device	motor	compumotor	This record manages axis 4 of the 6K controller.

To finish the example, we now show the client side *motor.dat* file that matches the MX server database shown above:

```
serv server network tcpip_server "" "" 0x0 192.168.17.10 9727
m1 device motor network_motor "" "" 0 0 -1e+06 1e+06 0 -1 -1 1 0 urad serv m1
m2 device motor network_motor "" "" 0 0 -1e+06 1e+06 0 -1 -1 1 0 urad serv m2
omega device motor network_motor "" "" 0 0 -1e+06 1e+06 0 -1 -1 1 0 urad serv m3
chi   device motor network_motor "" "" 0 0 -1e+06 1e+06 0 -1 -1 1 0 urad serv m4
```

In more verbose language, this breaks down as:

Record Name	Superclass	Class	Type	Explanation
serv	server	network	tcpip_server	This record manages the connection to an MX server running on port 9727 of host 192.168.17.10.
m1	device	motor	network_motor	This record requests server *serv* to perform actions on its behalf on the server's record *m1*.
m2	device	motor	network_motor	This record requests server *serv* to perform actions on its behalf on the server's record *m2*.
omega	device	motor	network_motor	This record requests server *serv* to perform actions on its behalf on the server's record *m3*.
chi	device	motor	network_motor	This record requests server *serv* to perform actions on its behalf on the server's record *m4*.

An important detail to notice here is that the name of the client's record does not have to be the same as the name the server knows it by. For example, client record *omega* sends requests to the server for server record *m3*. However, it is common and convenient to give the records the same name in the clients and in the servers.

One last detail to note is that an MX client is not restricted to only one server connection. If you had an MX client database that started with the following

```
serv1 server network tcpip_server "" "" 0x0 192.168.17.11 9727
serv2 server network tcpip_server "" "" 0x0 192.168.17.12 9727
serv3 server network tcpip_server "" "" 0x0 192.168.17.13 9727
```

then the client would have three simultaneous connections to three different MX servers.

1.4 Records and Record Fields

So what is a record, actually? On a technical level, it is a C data structure of type *MX_RECORD* that is declared in the MX source code in the header file *mx/libMx/mx_record.h*. But most of you probably did not want to know that.

On a more practical level, it is a repository for most of the information that MX program needs to know about a given object. The reason I say "most" is that MX records often have pointers to other MX records in the database. Thus, the *6k* record from the example in the previous section does not itself contain the information about the RS-232 settings of the port used to communicate with the Compumotor controller. Instead it uses the pointer *6k_rs232* in its own record description so that it can find that information in the record *6k_rs232*.

MX records are the primary "objects" of MX. They encapsulate both data values such as motor position, scaler counts, etc. and tables of functions ("methods") that operate on that data. Many of the data fields will be the same for all records of a given class. For example, all motor drivers need to have a place to store the current position of the motor. However, each record type will have type specific information in it. For example, a Pontech *stp100* record contains a field for the digital I/O pin number used to implement a home switch, a concept which many motor drivers would have no need for.

Internally, the data belonging to a record is contained in a variety of C data structures with names like *MX_MOTOR*, *MX_COMPUMOTOR*, *MX_PMAC_MOTOR*, and so forth. However, when it comes time to read data from a disk file or send it across the network, we can't really use the binary C structures or pointers to them for this. Theoretically you could, but it would be a really bad idea to do so. Instead, we use the concept of a *record field*.

An MX record field contains a pointer to a piece of data and also a description of its datatype and size. The record field also has a name that we can use to refer to it by. For example, if we look at an MX motor record called *theta*,

its position will be found in the record field *theta.position*. Thus, the record field name has two parts, namely, the record name *theta* and the field name *position*. Information read from MX database files and sent across MX network connections is identified by it record field name.

We said earlier in this chapter that each horizontal line in an MX database corresponds to one MX record. Within a given database line, the data is organized by field name. As we saw earlier, the first four fields are called *name*, *mx_superclass*, *mx_class*, and *mx_type*. These four fields are always found at the start of an MX database line. They are followed by two more fields called *label* and *acl_description* which are also common to all record types. These record fields can be summarized by the following table:

Field Name	**Field Type**	**Number of Dimensions**	**Sizes**	**Description**
name	string	1	16	The name of the record
mx_superclass	recordtype	0	0	The string “device”
mx_class	recordtype	0	0	The string “motor”
mx_type	recordtype	0	0	The name of the motor driver for this motor.
label	string	1	40	A verbose description of the record.
acl_description	string	1	40	Placeholder for an access control list *(not yet implemented)*.

You will see tables like the above throughout the rest of this manual, so we will try to explain it further here.

First of all, the **Field Name** column is just what it says, the name of the field. The **Field Type** column tells you what datatype the field contains. At present, MX supports the following datatypes, which are mostly modeled on the C datatypes:

Internal Name	Common Name	Description
MXFT_STRING	string	Null terminated C string
MXFT_CHAR	char	C char
MXFT_UCHAR	uchar	C unsigned char
MXFT_SHORT	short	C short
MXFT_USHORT	ushort	C unsigned short
MXFT_INT	int	C int
MXFT_UINT	uint	C unsigned int
MXFT_LONG	long	C long
MXFT_ULONG	ulong	C unsigned long
MXFT_FLOAT	float	C float
MXFT_DOUBLE	double	C double
MXFT_HEX	hex	A C unsigned long, usually represented in hexadecimal notation, such as 0x27a5.
MXFT_RECORD	record	A pointer to another MX record, represented by the name of the record in the database file.
MXFT_RECORDTYPE	recordtype	Used to point to device driver structures. Represented by the name of the driver type.
MXFT_INTERFACE	interface	A generalization of the MX_RECORD type which includes an optional address field. Typically used for devices that can be controlled by both RS-232 and GPIB. An example would be *gpib0:4* which refers to GPIB address 4 for GPIB interface record *gpib0*.

The **Number of Dimensions** column refers, of course, to the dimensions of the array containing the data. The case "0" stands for a single scalar value. The **Sizes** column contains a list of the sizes of each dimension.

Chapter 2

Motors

All motor records in MX support a common set of operations that are described in this chapter. We describe first the set of record fields found in the record description string in an MX database file for a motor.

Motor records are divided into two subclasses, namely, *stepper* and *analog* motors. The two classes are distinguished by the format of the numbers used to communicate with the underlying controller. Motor controllers for which positions, speeds, etc. are specified in integer units (*steps or encoder ticks*) are called *stepper* motors by MX motor support. Motor controllers for which positions, speeds, etc. are specified in floating point units are called *analog* motors by MX motor support.

2.1 Record Fields in the Record Description

The following fields must be included in the record description for a record in an MX database file. They must appear in the order presented below.

Field Name	Field Type	Number of Dimensions	Sizes	Description
name	string	1	16	The name of the record
mx_superclass	recordtype	0	0	The string "device"
mx_class	recordtype	0	0	The string "motor"
mx_type	recordtype	0	0	The name of the motor driver for this motor.
label	string	1	40	A verbose description of the record.
acl_description	string	1	40	Placeholder for an access control list *(not yet implemented)*.
raw_position	long *for stepper*, double *for analog*	0	0	The motor position in raw units. Generally this value will be overwritten by the position read from the motor controller.
raw_backlash_correction	long *for stepper*, double *for analog*	0	0	The MX backlash correction in raw units.
raw_negative_limit	long *for stepper*, double *for analog*	0	0	The software negative limit in raw units.
raw_positive_limit	long *for stepper*, double *for analog*	0	0	The software positive limit in raw units.
raw_deadband	long *for stepper*, double *for analog*	0	0	The motion deadband in raw units. A requested move is not performed unless the difference between the requested and the current positions is bigger than the deadband distance.
raw_minimum_speed_limit	long *for stepper*, double *for analog*	0	0	The slowest raw speed that can be requested for this motor. Negative values have special meanings. -1 means there are no restrictions on the requested raw speed. -2 means that the speed cannot be changed.
raw_maximum_speed_limit	long *for stepper*, double *for analog*	0	0	The fastest raw speed that can be requested for this motor. Negative values have the same meaning as for "raw_minimum_speed_limit".
scale	double	0	0	The "scale" field is used together with the "offset" field to compute positions in user units using the formula: *user_units = scale * raw_units + offset*.
offset	double	0	0	See the description of the "scale" field.
units	string	1	16	User units for the motor, such as *um*, or *deg*.

An example motor record description for a “disabled motor” is shown below.

```
theta device motor disabled_motor "" "" 0 0 -20000000 20000000 0 -1 -1 5e-05 0 deg
```

The disabled motor record was chosen for this example since it has no type-specific fields.

2.2 Motor Controllers

MX currently supports a wide variety of motor controllers.

2.2.1 Advanced Control Systems MCU-2

2.2.2 Aerotech Unidex 500

This set of drivers is for the Unidex 500 family of motor controllers from Aerotech. The available drivers include:

u500 Interface driver for controlling one or more Unidex 500 motor controllers.

u500_motor Motor driver for an individual axis of a Unidex 500 motor controller.

The MX drivers are only supported on Microsoft Windows since they depend on the binary WAPI Windows DLLs distributed by Aerotech.

u500 Record Fields

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common record field definitions*				
num_boards	long	0	0	The number of Unidex 500 boards attached to the computer.
firmware_filename	string	2	(*num_boards*, *mxu_filename_length*)	An array of firmware file names.
parameter_filename	string	2	(*num_boards*, *mxu_filename_length*)	An array of parameter file names.
calibration_filename	string	2	(*num_boards*, *mxu_filename_length*)	An array of calibration file names. This field is optional. Set it to an empty string "" if not needed.
pso_firmware_filename	string	2	(*num_boards*, *mxu_filename_length*)	An array of PSO firmware file names. This field is optional. Set it to an empty string "" if not needed.

In the table above, *mxu_filename_length* is a platform specific maximum filename length.

u500_motor Record Fields

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common motor field definitions*				
u500_record	record	0	0	The name of the U500 controller record for this motor.
board_number	int	0	0	The number of the U500 board used to control this motor.
axis_name	char	0	0	The single character U500 axis name for this motor.
default_speed	double	0	0	The default speed for the motor.

2.2.3 Am9513-based Motor

It is possible to configure an Am9513 chip to act as a very basic motor controller which only knows how to run at the base speed. See the **Am9513** subsection in the **Counter/Timer** section.

2.2.4 Animatics SmartMotor

This set of drivers is for the SmartMotor integrated motor/controller units from Animatics *(http://www.animatics.com/)*. The available drivers for this type of controller include:

smartmotor - A motor driver for the Animatics SmartMotor.
smartmotor_ain - An analog input driver for the Animatics SmartMotor.
smartmotor_aout - An analog output driver for the Animatics SmartMotor.
smartmotor_din - An digital input driver for the Animatics SmartMotor.
smartmotor_dout - An digital output driver for the Animatics SmartMotor.

2.2.5 APS Insertion Device

This driver is used to control either the gap or the harmonic energy of an undulator/wiggler insertion device at the Advanced Photon Source *(http://www.aps.anl.gov/)*. These driver make use of the information found at the APS ID Controls Information *(http://www.aps.anl.gov/aod/blops/IDINFO/ID_Controls.html)* web page.

The only driver supported is:

aps_gap - Controls either the gap or energy of the insertion device.

The record fields for this driver are:

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common motor field definitions*				
sector_number	int	0	0	The number of the APS sector that the insertion device belongs to. For example, at APS Sector 10-ID, the sector number would be 10.
motor_subtype	int	0	0	This field has two possible values: 1 - for APS gap control in millimeters. 2 - for APS ID energy control in keV.

2.2.6 Blu-Ice Motor

2.2.7 Bruker D8

This driver is for the D8 motor controller made by Bruker/Nonius.

Warning: This driver was only tested with a prerelease version of the D8.

2.2.8 Compumotor 6K and 6000 Series Motor Controllers

This set of drivers supports both the Compumotor 6000 and 6K series of controllers from the Compumotor division of Parker Hannifin *(http://www.compumotor.com/)*. Several different MX drivers are associated with this type of controller. They are:

- *compumotor_int* - An interface driver that manages all of the Compumotor controllers attached to a particular serial port or Ethernet connection. There will be one of these records for each controller.
- *compumotor* - The basic motor driver for Compumotor controllers. Each axis will use its own separate instance of this driver.
- *compumotor_lin* - Designed for performing linear interpolation moves with multiple axes. **Warning:** currently somewhat broken.
- *compumotor_din* - Used to read the digital input pins on a Compumotor 6K controller.
- *compumotor_dout* - Used to control the digital output pins on a Compumotor 6K controller.

So far these drivers have been tested with both the 6K and Zeta 6104 controllers.

An example database for a 4-axis Compumotor 6K controller would look like this

```
6k_rs232 interface rs232 tty "" "" 9600 8 N 1 S 0x0d0a 0x0d0a 10 0x0 /dev/ttyS5
6k interface generic compumotor_int "" "" 6k_rs232 0x0 1 1 4
m1 device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 1 0 um 6k_test 1 1 1
m2 device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 1 0 um 6k_test 1 2 1
m3 device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 1 0 um 6k_test 1 3 1
m4 device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 1 0 um 6k_test 1 4 1
```

compumotor_int

This interface driver manages information about the Compumotor controller as a whole that is not specific to a particular axis.

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common record field definitions*				
rs232_record	record	0	0	The name of the *rs232_record* record that the controller is plugged into. In general, the *rs232_record* can either correspond to a real RS-232 device such as the *tty* or *win32_com* driver or to a device connected via Ethernet, which will use the *tcp232* driver.
interface_flags	hex	0	0	Flags to select optional features. They are described in more detail below.
num_controllers	long	0	0	For a multidrop system such as the Zeta 6104, this record will control all of the real controllers attached to the multidrop cable. You must specify the number of controllers here. For a 6K controller, this value will always be 1.
controller_number	int	1	*num_controllers*	This varying length array lists the controller address for each controller as set by the Compumotor `ADDR` command. For a 6K controller, there will only be one value here.
num_axes	int	1	*num_controllers*	This varying length array lists the maximum number of motor axes for each controller on the multidrop cable. For a 6K controller, there will only be one value here. The allowed range of values for this field are from 1 to 8.

The *interface_flags* field will be the logical OR of the option bits selected from the following list:

Flag Name	Flag Value	Flag Description
MXF_COMPUMOTOR_AUTO_ADDRESS_CONFIG	0x1	If this flag is selected, the MX driver will send a Compumotor `ADDR` command that will cause the controllers on a multidrop to automatically configure their addresses from 1 to N. In general, you are recommened not use this option, but instead to configure the address in the startup program of the controller.
MXF_COMPUMOTOR_ECHO_ON	0x2	If this flag is selected, the MX driver expects the Compumotor controller to echo all commands sent to it. This option is not normally recommended since it adds extra RS-232 I/O for reading and discarding the echoed command strings. It is just here in case you are alternating operation of the controller between MX and some other package that expects the commands to be echoed.

compumotor

This motor driver handles one particular axis in a Compumotor controller. Since Compumotor interface records support multiple controllers and axes, both the controller number and the axis number must be specified.

Go to the MX Motor Driver Support page for the common motor record description fields. For the *compumotor* driver, the following driver specific fields are present:

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common motor field definitions*				
compumotor_interface_record	record	0	0	The name of the Compumotor generic interface for this motor.
controller_number	int	0	0	The controller number for this particular motor.
axis_number	int	0	0	The axis number for this particular motor in the specified controller.
flags	hex	0	0	Setting particular bits in the flags variable can modify the behavior of the driver. The individual bit values are specified below.

Notes

- The MX drivers assume that daisy-chained controllers are numbered from 1 to N where N is the number of controllers. This may be done via the **ADDR** command as described in the 6000 and 6K Command Reference manuals. For more information look at the sections named *RS-232C Daisy-Chaining* and *RS-485 Multi-Drop* in the Compumotor 6000 or 6K Programmer's Guide.

- If you use the ethernet port on a 6K controller, you will need to use the 'tcp232' RS-232 interface type and specify the port number to connect to as 502. There is not a lot of experience with controlling the 6K this way, so there may be lurking bugs related to the socket I/O.

  Warning: The port number seems to have changed for more recent 6K controllers. Using a packet sniffer while the vendor supplied code for Windows is running should be able to determine the correct port number.

- The MX Compumotor drivers make certain assumptions about the internal configuration of Compumotor controllers. The following figure shows an example startup script for Compmotor controllers that is compatible with MX. The MX drivers will not operate correctly if the variables **ERRLVL**, **EOT**, and **MA** are not set as shown. Also note that the setting shown for **EOT** means that the 'rs232' driver must be configured with the read and write terminators set to 0x0d0a. In addition, the setting **ECHO1** is required in order for multi-drop installations to function correctly. For a single controller, you may use **ECHO0** which will the amount of serial I/O required.

Bugs

At present, Zeta 6104s occasionally stop communicating with the MX driver. The exact circumstances under which this occurs is not entirely clear. However, since most of our Compumotor usage is migrating towards the 6K series, the need to fix this issue may become less important.

2.2.9 DAC Motor

This driver is used to control an MX analog output device as if it were a motor. The type of control supported here is fairly basic. When a move is commanded, all that happens is that the DAC output voltage is changed to the value corresponding to the new position.

There is only one driver specific specific field for the *dac_motor* driver. Here is the description of it:

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common motor field definitions*				
dac_record	record	0	0	The name of the MX analog output record used by this motor record.

2.2.10 Delta Tau PMAC

The PMAC series of motor controllers is manufactured by Delta Tau Data Systems of Chatsworth, CA. PMAC motor controllers are definitely the most powerful motor controllers supported by MX. However, they are also the most complicated to setup and program of all the controllers supported by MX, so they may not be the best choice for simple applications.

The MX PMAC drivers are designed to be easily adaptable to any model of PMAC motor controller. However, so far the drivers have been mostly used with the Turbo PMAC series of controllers. The drivers listed below currently all operate via PMAC ASCII communication interfaces of various types.

MX has a large number of drivers for interacting with PMAC motor controllers:

```
; This command script is to be downloaded into a Compumotor 6000 or 6K
; controller in order to set up the controller to be compatible with MX.
; The script sets up a startup command program to be executed by the
; Compumotor controller at power-on.  The commands for ERRLVL, EOT, and MA
; _must_ be set as shown below or else the MX driver will not work.
; If you need other commands to set motor parameters, network addresses,
; and so forth, add them to the commands listed below.
;
; Please note that if you use ECHO1, you must add the 0x2 echo on flag to
; the compumotor_int record that defines the connection to the interface.
; This lets the driver know that it needs to discard echoed characters.
;
; For a single controller, it is better to set ECHO0 since that will
; eliminate the overhead of discarding the characters.  However, in a
; multidrop daisy chain configuration, ECHO1 _must_ _be_ _set_ since the
; controllers rely on the echoing to send the command on to the next
; controller in the daisy chain.  If ECHO0 is set in a daisy chain
; configuration, the configuration will mostly work but will randomly
; lock up from time to time, so don't do it.
;
; William Lavender -- Last modified April 27, 2002
;
DEF mstart
ERRLVL1          ; Have the controller generate a minimum amount of output.
EOT13,10,0       ; Want all output lines to have the same line terminators.
MA1              ; Use absolute mode for positioning.
LH0              ; Disable limits (for testing only!)
ENC0             ; Use motor step mode ( or set ENC1 for encoder step mode ).
ECHO1            ; Enable command echoing.
END
```

Figure 2.1: Recommended STARTP program for MX controlled Compumotor motors.

pmac	Interface driver for controlling one or more PMAC motor controllers connected to an ASCII serial interface.
pmac_motor	Motor driver for controlling a single motor of a PMAC controller.
pmac_cs_axis	Motor driver for controlling a coordinate system axis belonging to a PMAC motor controller.
pmac_mce	Multichannel encoder (MCE) driver for reading out any motor belonging to a given PMAC motor controller.
pmac_ainput	Analog input driver for reading a floating point value from a PMAC variable.
pmac_aoutput	Analog output driver for writing a floating point value to a PMAC variable.
pmac_dinput	Digital input driver for reading an integer value from a PMAC variable.
pmac_doutput	Digital output driver for writing an integer value to a PMAC variable.
pmac_long	Variable driver for reading and writing signed integer values to and from a PMAC variable.
pmac_ulong	Variable driver for reading and writing unsigned integer values to and from a PMAC variable.
pmac_double	Variable driver for reading and writing floating point values to and from a PMAC variable.

MX PMAC drivers

pmac - *Pmac* interface records are used to control one or more PMAC motor controllers attached to a given external interface. An example *pmac* record looks like

```
pmac1 interface generic pmac "" "" rs232 pmac1_rs232 1
```

which describes a single PMAC motor controller attached to MX RS-232 record **pmac1_rs232**.

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common record field definitions*				
port_type_name	string	1	80	A string such as *rs232* that describes the type of PMAC interface this is. See below for more information.
port_args	string	1	80	This contains port type specific information such as the name of an RS-232 port record. See below for more information.
num_cards	int	0	0	The number of individual PMAC controllers attached to this interface. In most cases this will be 1. However, for a controller attached to a multidrop connection such as RS-485, this will be the number of controllers attached to the multidrop.

PMAC ASCII command interfaces are accessible via a variety of different mechanisms such as RS-422, Ethernet, USB, VME, etc. Since the information needed to describe the interface can vary widely from interface type

to interface type, the information needed to the interface is specified in a string called *port_args*. The currently defined port types are:

port_type_name	*port_args*
rs232	The name of the MX RS-232/422/485 record that this PMAC interface is attached to, such as *pmac1_rs232* in the example above.
epics_ect	This port type uses the string command and response interfaces provided with the EPICS PMAC software written by Tom Coleman of the Argonne National Laboratory ECT group. This port type uses EPICS process variable names that are constructed by appending "StrCmd" or "StrRsp" to the names. Thus, if the port args were "S18ID", then the EPICS process variables used by this interface would be *S18IDStrCmd.VAL* and *S18IDStrRsp.VAL*. Note: There is an alternate set of MX EPICS drivers called *pmac_tc_motor* and *pmac_bio_motor* that is described elsewhere in this manual.

Note: It is anticipated that **ethernet** and maybe **usb** port types will be added at some point in the future.

pmac_motor - A *pmac_motor* record refers to one particular motor in a PMAC motor controller. The motor is controlled mostly via PMAC "jog" mode commands except for certain features not available via jog commands.

An example *pmac_motor* record looks like

```
theta device motor pmac_motor "" "" 0 0 -10000000 10000000 0 -1 -1   0.05 0 um pmac1 0
```

which describes a motor called **theta** which belongs to controller 0, axis 4 of PMAC interface **pmac1**. The example motor uses a scale factor of 0.05 μ-meters per step and raw motion limits of $\pm$ 10000000 steps.

Field Name	**Field Type**	**Number of Dimensions**	**Sizes**	**Description**
See *Common motor field definitions*				
pmac_record	record	0	0	Name of the PMAC interface record that controls this motor.
card_number	int	0	0	Card number of the PMAC card that controls this motor. For PMACs that are not in a multi-drop configuration, the card number will normally be 0.
motor_number	int	0	0	The PMAC axis number for this specific motor.

pmac_cs_axis - A *pmac_cs_axis* record makes use of a specified coordinate system axis for a coordinate system defined in a PMAC controller. PMAC coordinate systems can be thought of as a way of defining "pseudomotors" inside a PMAC controller in a manner that is analogous to the way MX defines pseudomotors. However, PMAC coordinate system axes are more powerful than MX pseudomotors, since for a coordinate system, the PMAC controller is able to ensure that all of the raw motors are able to maintain their correct relative relationship even while the motors are moving. Ordinary MX pseudomotors make sure that the real motors are at the correct positions at the beginning and end of motor moves, but they cannot do this while a move is in progress.

Pmac_cs_axis motor records require that some preliminary setup be done in the PMAC before they may be used. There are three primary steps in this process:

- The coordinate system that this axis is to be part of must be set up before this record may be used.

- You must write a motion program that will be run every time a move of this axis is commanded. The motion program **must** define the move destination, the feedrate (*reciprocal of the speed*), the acceleration time, and the S curve acceleration times in terms of PMAC motion variables so that the *pmac_cs_axis* driver can set them. I recommend that you use Q-variables so that variables used by this coordinate system will not interfer with other coordinate systems used by your PMAC.

- You must arrange for the current position of the coordinate system axis to be continuously updated to a PMAC variable that you specify. The most obvious way to do this is with a constantly running PMAC PLC program which is set up to calculate the coordinate system axis position from the real motor positions at all times. I would recommend that you use a Q-variable for this too. Of course, the kinematic calculation logic of the PLC program must match the logic of the PMAC motion program mentioned above.

An example *pmac_cs_axis* record looks like

```
det_distance motor pmac_cs_axis "" "" 0 0 200 1000 0 -1 -1 1 0 mm pmac1 0 2 Z 3 Q50 Q5
```

This describes a motor called **det_distance** which corresponds to axis Z of coordinate system 2 running in card 0 of PMAC interface **pmac1**. The axis performs moves using motion program 3 with position, destination, feedrate, acceleration time, and S-curve acceleration time managed by PMAC coordinate system variables Q50 through Q54.

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common motor field definitions*				
pmac_record	record	0	0	Name of the PMAC interface record that controls this axis.
card_number	int	0	0	Card number of the PMAC card that controls this axis. For PMACs that are not in a multi-drop configuration, the card number will normally be 0.
axis_name	char	0	0	The name of the coordinate system axis used by this motor. The possible names are X, Y, Z, ???, A, B, and C.
move_program_number	int	0	0	The number of the motion program that is used to move this axis as part of the coordinate system.
position_variable	string	1	8	Name of the PMAC variable that the MX driver uses to read the current position of the axis from.
destination_variable	string	1	8	The MX driver writes the new axis destination to this PMAC variable before starting the motion program to perform the move.
feedrate_variable	string	1	8	The MX driver sets the axis speed by writing to the specified PMAC feedrate variable.
acceleration_time_variable	string	1	8	The MX driver sets the axis acceleration time by writing to this variable.
s_curve_acceleration_time_variable	string	1	8	The MX driver sets the axis S-curve acceleration time by writing to this variable.

Note: If all you want is basic control of the individual motors belonging to a PMAC controller, then it is not necessary to create MX *pmac_cs_axis* motor records or coordinate systems in the PMAC. You can get basic control of the motors with just the *pmac_motor* records, with much less setup required. You only need *pmac_cs_axis* records if you want to make use of the special abilities of PMAC coordinate systems.

pmac_mce - A *pmac_mce* record is used to read out the position of a motor at the end of an MX quick scan.

To be continued...

2.2.11 Disabled Motor

This driver type was added to make it easier to quickly disable a particular motor in the MX database. The *disabled_motor* driver has no extra fields beyond the default motor fields. This makes it easy to just change the *mx_type* field in the record description to *disabled_motor* and have the record description still be a valid record description, albeit with ignored, trailing text.

The *disabled_motor* driver makes no attempt to accurately simulate the behavior of a real motor. If you want a more accurate simulation, select the *soft_motor* driver described further on in this chapter.

2.2.12 DSP E500

The DSP E500 was a CAMAC-based motor controller that is no longer manufactured. Ten or more years ago, this controller was one of the most popular motor controllers in use in the synchrotron radiation field, and there are still many in use at installations around the world.

2.2.13 EPICS Motor

The MX *epics_motor* driver uses EPICS Channel Access to communicate as an EPICS client with the Advanced Photon Source's EPICS motor driver *(http://www.aps.anl.gov/upd/people/sluiter/epics/motor/)* in an EPICS IOC.

There is only one driver specific specific field for the *epics_motor* driver. Here it is:

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common motor field definitions*				
epics_record_name	string	1	40	The name of the EPICS motor record used by this MX record.

2.2.14 IMS MDrive

The IMS MDrive is an integrated motor/controller combination that is housed as one unit. The MDrive is produced by Intelligent Motion Systems *(http://www.imshome.com/)*.

2.2.15 IMS Panther and IM483

This driver is for the Panther HI/HE microstepping driver/controllers and is also known to work for the IM483 as well. These controllers are manufactured by Intelligent Motion Systems *(http://www.imshome.com/)*.

Note: It is likely that this driver will work with other controllers from IMS, although the ones listed above are the only ones tested.

2.2.16 Joerger SMC24

This is an MX motor driver for the Joerger SMC24 CAMAC stepping motor controller, which is still available as of August 2003, from Joerger Enterprises, Inc.

Warning: As far as I know, this driver has not been tested in a long time. However, if broken, I expect that it would take less than a day to get the MX driver working again.

The Joerger SMC24 controller does not have an internal register to record its current position, so it needs the assistance of an external device to keep track of the motor's absolute position. Traditionally, a Kinetic Systems 3640 CAMAC up/down counter is used as the external device, but any device capable of acting as an encoder-like device may be used as long as there is an MX encoder driver for it.

Also, traditionally the Kinetic Systems 3640 up/down counter was modified in the field to connect pairs of 16-bit up/down counters to form 32-bit up/down counters. However, if this has not been done, the driver can also emulate in software a 32-bit step counter using a 16-bit hardware encoder by setting the bit in the "flags" variable called MXF_SMC24_USE_32BIT_SOFTWARE_COUNTER (0x1) .

2.2.17 Kohzu SC-200, SC-400, and SC-800

2.2.18 Lakeshore 330 Temperature Controller

2.2.19 Mar Desktop Beamline

2.2.20 McLennan

2.2.21 McLennan PM-304

An example database for the PM-304 looks like this:

```
theta_rs232 interface rs232 tty "" "" 9600 7 E 1 N 0x0d0a 0x0d0a 10 0x0 /dev/ttyS0
theta_real  device motor pm304 "" "" 0 0 -1800000 200000 0 -1 -1 -5e-05 0 deg theta_rs232 1
```

The MX driver for the PM304 requires that responses from the controller include an address prefix. By default, the PM304 has this feature turned off. You may turn it on by sending the string

```
1AD
```

to the PM304, assuming that it is configured for address 1. Please note that the AD command is a toggle, so if address prefixes are already turned on, the AD command will turn them off.

2.2.22 National Instruments PC-STEP

2.2.23 National Instruments ValueMotion

2.2.24 Network Motor

2.2.25 New Focus Picomotor

2.2.26 Newport

Note: For the MM4000, this driver *assumes* that the value of the field "Terminator" under "General Setup" for the controller is set to CR/LF.

2.2.27 NSLS MMC32

2.2.28 OMS VME58

2.2.29 OSS μ-GLIDE

2.2.30 Oxford Cryosystems Cryostream 600 Temperature Controller

2.2.31 Oxford Instruments ITC503 Temperature Controller

itc503_control

The value of 'parameter_type' is the letter that starts the ITC503 command that will be sent. The currently supported values are:

- A - Set auto/manual for heater and gas
- C - Set local/remote/lock status
- G - Set gas flow (in manual only)
- O - Set heater output volts (in manual only)

There are several other ITC503 control commands, but only the ones likely to be used in routine operation are supported.

itc503_motor

The two lowest order bits in 'itc503_motor_flags' are used to construct a 'Cn' control command. The 'Cn' determines whether or not the controller is in LOCAL or REMOTE mode and also whether or not the LOC/REM button is locked or active. The possible values for the 'Cn' command are:

- C0 - Local and locked (default state)
- C1 - Remote and locked (front panel disabled)
- C2 - Local and unlocked
- C3 - Remote and unlocked (front panel disabled)

itc503_status

The value of 'parameter_type' is used to construct an ITC503 'R' command. Thus, the values of the parameters are as listed in the Oxford manual:

- 0 - Set temperature
- 1 - Sensor 1 temperature
- 2 - Sensor 2 temperature
- 3 - Sensor 3 temperature
- 4 - Temperature error

- 5 - Heater O/P (as
- 6 - Heater O/P (as Volts, approx.)
- 7 - Gas flow O/P (arbitrary units)
- 8 - Proportional band
- 9 - Integral action time
- 10 - Derivative action time
- 11 - Channel 1 freq/4
- 12 - Channel 2 freq/4
- 13 - Channel 3 freq/4

2.2.32 Pan-Tilt-Zoom Motor

2.2.33 Phidget Stepper (old version)

2.2.34 Physik Instrumente E662 Piezo Controller

2.2.35 Pontech STP100

The permitted board numbers are from 1 to 255.

The permitted values for digital I/O pins are:

0	-	Disable the pin.
3, 5, 6, 8	-	The pin is active closed.
-3, -5, -6, -8	-	The pin is active open.

Pins 5 and 6 are normally used for limit switches while either pin 3 or pin 8 is used for the home switch. This is because pins 5 and 6 already have pullup resistors.

The output of the RP command is 0 = closed and 1 = open.

2.2.36 Prairie Digital Model 40

2.2.37 Precision MicroControl MCAPI-based Motor Controllers

2.2.38 Radix Databox

2.2.39 Scientific Instruments 9650 Temperature Controller

2.2.40 SCIPE Motor

2.2.41 Soft Motor

2.2.42 Spec Motor

2.2.43 Velmex VP9000

2.2.44 XIA HSC-1 Huber Slit Controller

By default, the HSC-1 Huber Slit Controllers are delivered with default values that do not allow the slit blades to be moved to anywhere the blades can physically reach. The default values for parameters 1 and 2 are:

Parameter	Name	Default Value	Default in um
1	Outer Motion Limit	4400	11000 um
2	Origin Position	400	1000 um

The MX drivers for the HSC-1 assume that these two parameters have been redefined to have the following values:

Parameter	Name	Default Value	Default in um
1	Outer Motion Limit	10400	26000 um
2	Origin Position	5200	13000 um

The reprogramming must be done using a terminal program like Kermit or Minicom. Suppose you have an HSC-1 controller with a serial number of XIAHSC-B-0001. Then the appropriate commands to send to the HSC-1 would be:

```
!XIAHSC-B-0001 W 1 10400
!XIAHSC-B-0001 W 2 5200
```

Next, in the MX config file, specify the limits, scales and offsets of the various axes as follows:

XIA motor name	negative limit (raw units)	positive limit (raw units)	scale	offset
A	-65535	65535	2.5	-13000
B	-65535	65535	2.5	-13000
C	-65535	65535	2.5	0
S	0	131071	2.5	-26000

Then, you will be able to move the A, B, and C motors from -13000 um to +13000 um and the S motor from 0 um to 26000 um.

Please note that the HSC-1 motor positions can only be set to the value 0. A “set motor ... position” command to any other value than zero will fail. The “set motor ... position 0” command itself will cause the HSC-1 to execute an “Immediate Calibration” or “0 I” command. Also note that the slit size motor S cannot be moved to a negative value, so if S is at zero and there is a visible gap between the blades, then you will have to manually close the slit by hand.

Here is an example database for two HSC-1 controllers attached to the same serial port:

```
hsc1_rs232 interface rs232 tty "" "" 9600 8 N 1 N 0xd0a 0xd /dev/ttyS0
hsc1_1     interface generic hsc1 "" "" hsc1_rs232 2 XIAHSC-B-0067 XIAHSC-B-0069
hsc67a     device motor hsc1_motor "" "" 0 0 -65535  65535 0 -1 -1 2.5 -13000 um hsc1_1 0 A
hsc67b     device motor hsc1_motor "" "" 0 0 -65535  65535 0 -1 -1 2.5 -13000 um hsc1_1 0 B
hsc67c     device motor hsc1_motor "" "" 0 0 -65535  65535 0 -1 -1 2.5      0 um hsc1_1 0 C
hsc67s     device motor hsc1_motor "" "" 0 0      0 131071 0 -1 -1 2.5 -26000 um hsc1_1 0 S
hsc69a     device motor hsc1_motor "" "" 0 0 -65535  65535 0 -1 -1 2.5 -13000 um hsc1_1 1 A
hsc69b     device motor hsc1_motor "" "" 0 0 -65535  65535 0 -1 -1 2.5 -13000 um hsc1_1 1 B
hsc69c     device motor hsc1_motor "" "" 0 0 -65535  65535 0 -1 -1 2.5      0 um hsc1_1 1 C
hsc69s     device motor hsc1_motor "" "" 0 0      0 131071 0 -1 -1 2.5 -26000 um hsc1_1 1 S
```

2.3 Pseudomotors

Pseudomotor support goes here.

2.3.1 ADSC Two Theta

2.3.2 A-Frame Detector Motor

This is an MX motor driver for the pseudomotors used by Gerd Rosenbaum's A-frame CCD detector mount. The geometry of this detector mount is shown in the following figure:

The pseudomotors available are:

- *detector_distance* - This is the length of the line perpendicular to the plane containing the front face of the detector which passed through the center of rotation of the goniometer head.
- *detector_horizontal_angle* - This is the angle between the line used by the detector distance and the horizonal plane.
- *detector_offset* - This is the distance between the centerline of the detector and the line used to define the detector distance above.

There are three constants that describe the system:

- *A* - This is the perpendicular distance between the two vertical supports that hold up the detector.
- *B* - This is the distance along the centerline of the detector from the front face of the detector to the point where a perpendicular from the downstream detector pivot intersects this line.
- *C* - This is the separation between the centerline of the detector and the line defining the detector distance above.

The pseudomotors depend on the positions of three real motors. These are:

- *dv_upstream* - This motor controls the height of the upstream vertical detector support.
- *dv_downstream* - This motor controls the height of the downstream vertical detector support.
- *dh* - This motor controls the horizontal position of the vertical detector supports.

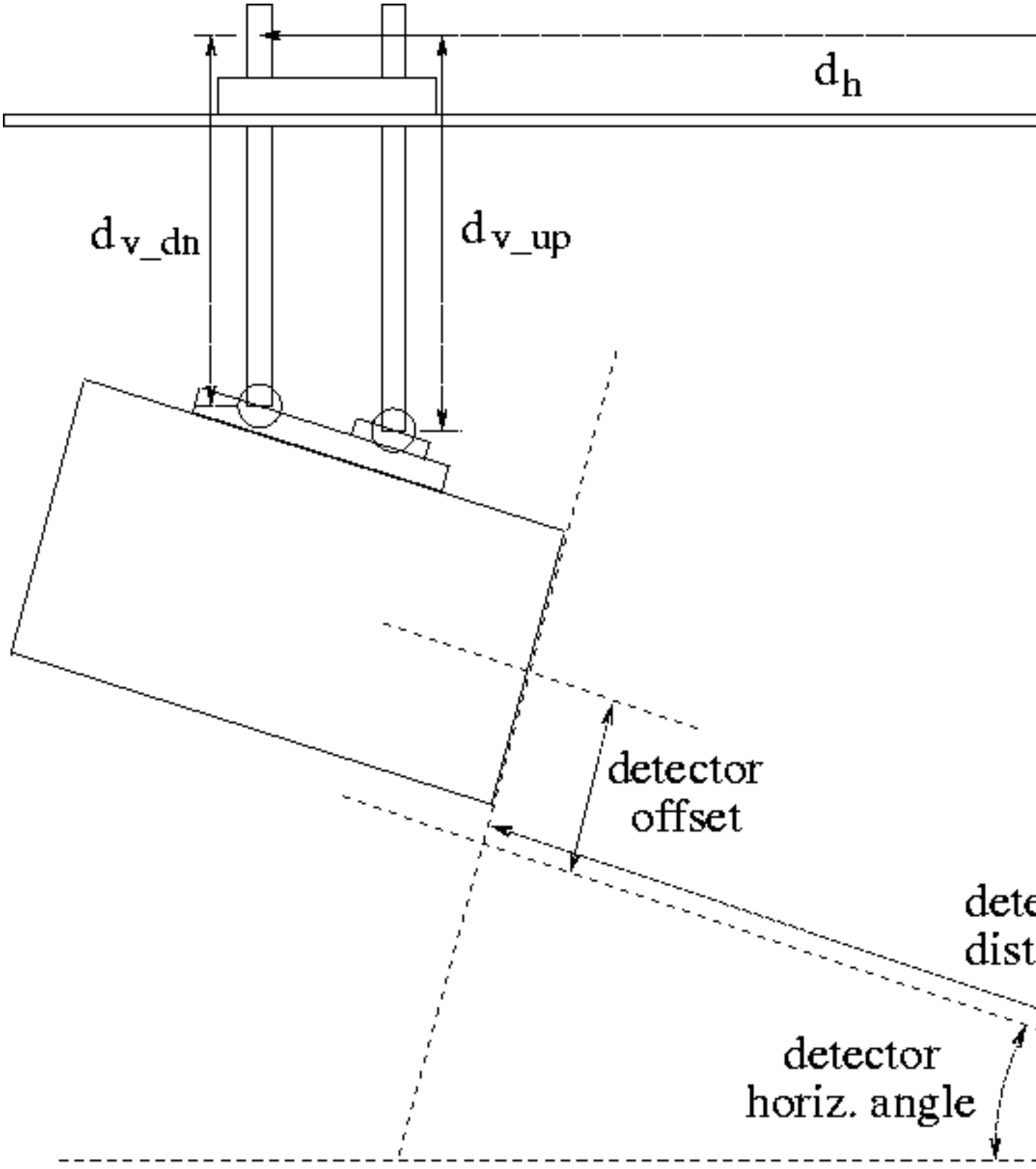


Figure 2.2: A-frame CCD detector mount designed by Gerd Rosenbaum

Confused? I am planning to write a short document that describes the definitions of these parameters in more detail and derives the formulas describing them. If you are reading this text and I have not yet written that document, then pester me until I do write it.

Warning: The detector horizonal angle is expressed internally in radians. If you want to display the angle in degrees, use the scale field of the angle pseudomotor to do the conversion.

2.3.3 ALS Dewar Positioner

2.3.4 APS 18-ID

2.3.5 Delta

2.3.6 Elapsed Time

2.3.7 Energy

2.3.8 Linear Function

An example database for the *linear_function* pseudomotor looks like:

```
cmirror_us   device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 1 0 um 6k_coll 1 3
cmirror_ds   device motor compumotor "" "" 0  0 -1000000 1000000 0 -1 -1 1 0 um 6k_coll 1 2
cmirror_bend device motor linear_function "" "" 0  0 -1000000 1000000 0 -1 -1
1 0 um 0x1 2 cmirror_us cmirror_ds 0.5 0.5 0 0 0.5 0.5
```

2.3.9 Monochromator

The monochromator pseudomotor is implemented using a large collection of MX records. These records can be categorized into several groups:

- The monochromator record with *N* dependencies.
- *N* dependency list records.
- *N* dependency enable records.
- *N* dependency parameter records.
- *N* dependency record list records.
- *N* dependency type records.

where the value of *N* above is set by the value of the *num_dependencies* field in the monochromator record.

Monochromator Record

The MX Motor Driver Support page describes the common motor record description fields. For the *monochromator* driver, the following driver specific fields are present:

Field Name	Field Type	Number of Dimensions	Sizes	Description
See *Common motor field definitions*				
num_dependencies	long	0	0	The number of dependencies for this monochromator pseudomotor.
list_array	record	1	*num_dependencies*	The list of depencency list records.

Example:

```
theta device motor monochromator "" "" 0 0 -10 270 0 -1 -1 1 0 deg 4 theta_list momega_lis
```

The monochromator record is a pseudomotor record which contains a list of the dependencies used by the pseudomotor. In general, one of the dependencies will be a *primary* dependency which describes the primary axis used by the monochromator pseudomotor *(usually theta)*. The rest of the dependencies will be *secondary* dependencies that describe motors that are to be moved to positions that depend on the position of the primary dependency motor. There should only be one primary dependency.

In the example above, the dependencies specified are:

- *theta_list* - This is the primary dependency and describes the dependence of the monochromator pseudomotor on the real theta axis of the monochromator.
- *momega_list* - A secondary dependency that controls the angle between the first and second monochromator crystal.
- *id_ev_list* - A secondary dependency that controls the energy of the peak of the undulator spectrum for this beamline.
- *normal_list* - A secondary dependency that controls the perpendicular spacing between the first and second monochromator crystals.

This example does not include all of the available dependency types which are described in more detail below. In addition, the primary dependency does not have to be the first record listed, but it is customary to do so.

Dependency List Records

An example dependency list record looks like

```
momega_list  variable inline record "" "" 1  4  momega_enabled momega_type momega_params m
```

Dependency list records must be four element 1-dimensional arrays of type MXFT_RECORD. The individual elements of this array must be in the following order:

- *Dependency enable record* - used to enable or disable the dependency. (*momega_enabled* in the example above.)
- *Dependency type record* - describes what type of dependency this is. (*momega_type* in the example above.)
- *Dependency parameters record* - describes the parameters used by this dependency. (*momega_params* in the example above.)
- *Dependency record list* - describes the records used by this dependency. (*momega_records* in the example above.)

The individual elements of the array are described in more detail below.

Dependency Enable Records

An example dependency enable record looks like

```
momega_enabled variable net_variable net_int "" "" localhost momega_enabled.value 1  1  0
```

The dependency enable record must be a variable record of type MXFT_INT. It has two legal values:

- *1* - The dependency is enabled and the dependent motor(s) will be moved to positions that correspond to the position of the primary dependency.
- *0* - The dependency is disabled and the dependent motor(s) will not be moved.

Dependency Parameter Records

An example dependency parameters record looks like

```
momega_params variable net_variable net_double "" "" localhost momega_params.value 1 4 0 0
```

The dependency parameters record will be a 1-dimensional variable record of some kind. The particular variable type used will depend on the dependency type as described below. In the example above, the parameters record is a 1-dimensional array of integers that are all initialized to 0.

Dependency Record List Records

An example dependency record list looks like

```
momega_records variable inline record "" "" 1 1 momega
```

The dependency record list record will be a 1-dimensional variable record of type MXFT_RECORD. The number and identity of the listed records will depend on the dependency type as described below. In the example above, the record list array contains only the record *momega*.

Dependency Type Records

An example dependency type record looks like

```
momega_type variable inline int "" "" 1 1 2
```

The dependency type record will be 1-dimensional variable record of type MXFT_INT with only one element, namely, the dependency type. In the example above, the dependency type is 2.

At present, nine different dependency types are available. Dependency types 0 and 1 are *primary* dependency types, while types 2 through 8 are *secondary* dependency types.

Type 0 - Theta dependency

The theta dependency is used to control the position of the primary theta axis. This dependency is normally the *primary* dependency for the monochromator pseudomotor. If this dependency does not exist or is disabled, the real theta axis will not be moved at all.

- **Record list record** - This is a 1-dimensional array with only one element, specifically, the name of the *real* theta motor record.

- **Parameters record** - This record must be present, but its contents are not used by this dependency. Typically, a dummy variable will be used here as in the example below.

An example set of records for the theta dependency looks like

```
theta_list    variable inline record "" "" 1  4  theta_enabled theta_type dummy_params the
theta_type    variable inline int    "" "" 1  1  0
theta_enabled variable inline int    "" "" 1  1  1
theta_records variable inline record "" "" 1  1  theta_real
dummy_params  variable inline double "" "" 1  1  0
```

Normally, there is no reason for the users to disable this dependency, so it is standard to hard code it to 1 as in the example above.

Type 1 - Energy dependency

The energy dependency is a *primary* dependency that computes the monochromator theta angle from a monochromator energy provided by a foreign beamline control system. Use of this dependency is **not recommended**, unless the underlying beamline control software/hardware does not provide a direct way of querying and controlling the theta angle. Use of this dependency is **incompatible** with the type 0 theta dependency specified above. Do not specify both of them in the same MX database.

If you are looking for a way to control a *dependent* motor as a polynomial function of energy, you should be using the type 8 *energy polynomial* dependency described below.

- **Record list record** - This is a 1-dimensional array of type MXFT_RECORD containing two elements:
  - *energy record* - This motor record queries and controls the monochromator energy.
  - *monochromator d spacing record* - This is a variable record that contains the d spacing of the monochromator crystal in angstroms.
- **Parameters record** - This record must be present, but its contents are not used by this dependency. Typically, a dummy variable will be used here as in the example below.

An example set of records for the energy dependency looks like

```
energy_list    variable inline record        "" "" 1  4  energy_enabled energy_type dummy_
energy_type    variable inline int           "" "" 1  1  1
energy_enabled variable net_variable net_int "" "" 1  1  1
energy_records variable inline record        "" "" 1  2  energy d_spacing
dummy_params   variable inline double        "" "" 1  1  0
```

A corresponding d spacing variable would look like

```
d_spacing variable net_variable net_double "" "" localhost d_spacing.value 1  1  3.1355
```

The monochromator theta angle is computed using the standard Bragg equation

theta = asin(12398.5 / (2.0 * d_spacing * energy))

Normally, there is no reason for the users to disable this dependency, so it is standard to hard code it to 1 as in the example above.

Type 2 - Polynomial dependency

This dependency type is a *secondary* dependency that allows a dependent motor to be moved to positions that are a polynomial function of the theta position of the monochromator. The computed position will be of the form

dependent_position = c0 + c1 * theta + c2 * (theta**2) + c3 * (theta**3) + ...

Beamline staff may configure this polynomial to have as few or as many terms in it as they want by changing the number of elements in the parameters array below. For example, a parameters array record with only two array elements will describe a dependent position that has a linear dependence on theta, while a parameters array record with four array elements describes a cubic dependence on theta. It is generally not useful to use a polynomial of higher order than cubic, although there is no limit in the record as to how high the order may be.

- **Record list record** - This is a 1-dimensional array with only one element, specifically, the name of the dependent motor record.
- **Parameters record** - This is a 1-dimensional array of type MXFT_DOUBLE which contains the coefficients of the polynomial. The coefficients are specified in order starting with the constant term and continuing up to the coefficient of the highest order term.

An example set of records for the polynomial dependency looks like

```
momega_list    variable inline record           "" "" 1  4  momega_enabled momega_type mome
momega_type    variable inline int              "" "" 1  1  2
momega_enabled variable net_variable net_int    "" "" localhost momega_enabled.value 1  1
momega_records variable inline record           "" "" 1  1  momega
momega_params  variable net_variable net_double "" "" localhost momega_params.value  1  4
```

In the example above, *momega_params* is a cubic polynomial. If the value of *momega_params* at some particular time was set to something like (0.32, 0.41, -0.02, 0.015), the computed polynomial would have the form

momega = 0.32 + 0.41 * theta - 0.02 * (theta**2) + 0.015 * (theta**3)

Typically, the values of these coefficients will be determined by measuring the location of the peak X-ray intensity as a function of the dependent motor position for several values of theta Then a curve will be fitted to the measurements.

Type 3 - Insertion device energy dependency

This dependency type is a *secondary* dependency that changes the energy of the undulator peak such that the maximum of the undulator spectrum is at the same energy as the monochromator.

- **Record list record** - This is a 1-dimensional array of type MXFT_RECORD containing two elements:
  - *insertion device motor record* - This is a motor record that controls the position of the undulator peak in units of eV.
  - *monochromator d spacing record* - This is a variable record that contains the d spacing of the monochromator crystal in angstroms.
- **Parameters record** - This is a 1-dimensional array of type MXFT_DOUBLE which contains two elements:
  - *gap harmonic* - This is the requested undulator harmonic number and should be a positive odd integer such as 1, 3, 5, etc. This is usually set to 1.

- *gap offset* - This is an offset to be added to the computed gap energy.

For a given monochromator *theta* position in degrees, the undulator energy is computed as follows:
mono_energy = 12398.5 / (2.0 * d_spacing * sin(theta))
undulator_energy = (mono_energy + gap_offset) / gap_harmonic
Please note that if the undulator controls provided by your storage ring *also* have a way of setting the gap harmonic in addition to the method provided by MX, then you should only set one of them to the harmonic number and set the other to 1. For example, at the APS, if you set both EPICS's variable for the gap harmonic to 3 *and* MX's variable for the gap harmonic to 3, you would actually end up with the ninth harmonic.

An example set of records for the insertion device energy dependency looks like

```
id_ev_list    variable inline record           "" "" 1  4  id_ev_enabled id_ev_type id_ev_
id_ev_type    variable inline int              "" "" 1  1  3
id_ev_enabled variable net_variable net_int    "" "" localhost id_ev_enabled.value 1  1  0
id_ev_records variable inline record           "" "" 1  2  id_ev d_spacing
id_ev_params  variable net_variable net_double "" "" localhost id_ev_params.value 1  2  1
```

Type 4 - Constant exit Bragg normal dependency

This dependency type is a *secondary* dependency that changes the perpendicular spacing between the first and second monochromator crystals so that the X-ray beam exiting the monochromator stays at a constant height.

- **Record list record** - This is a 1-dimensional array of type MXFT_RECORD containing two elements:
  - *normal record* - This motor record controls the perpendicular spacing between the two monochromators crystals.
  - *beam offset record* - This is a variable record of type MXFT_DOUBLE that contains the desired fixed offset distance of the beam expressed in the same units as the *normal* motor.
- **Parameters record** - This record must be present, but its contents are not used by this dependency. Typically, a dummy variable will be used here as in the example below.

An example set of records for the Bragg normal dependency looks like

```
normal_list    variable inline record        "" "" 1  4  normal_enabled normal_type dummy_
normal_type    variable inline int           "" "" 1  1  4
normal_enabled variable net_variable net_int "" "" localhost normal_enabled.value 1  1  0
normal_records variable inline record        "" "" 1  2  normal beam_offset
dummy_params   variable inline double        "" "" 1  1  0
```

A corresponding beam offset variable would look like

```
beam_offset variable net_variable net_double "" "" localhost beam_offset.value 1  1  -3500
```

The Bragg normal position is computed from the beam offset and the monochromator theta angle via the equation
bragg_normal = beam_offset / (2.0 * cos(theta))
If a positive move of the normal motor at theta = 0 is in the opposite direction from the desired beam offset, then the value of the beam offset must be set to a negative number. For example, this is true of the MX installations at APS sectors 10 and 17 where beam_offset = -35000 um.

Type 5 - Constant exit Bragg parallel dependency dependency

This dependency type is a *secondary* dependency that translates the second crystal parallel to its surface. This dependency should normally be used in combination with the Bragg normal dependency listed above. It is used to ensure that the X-ray beam does not fall off the end of the second crystal.

- **Record list record** - This is a 1-dimensional array of type MXFT_RECORD containing two elements:
  - *parallel record* - This motor record controls the translated position of the second crystal.
  - *beam offset record* - This is a variable record of type MXFT_DOUBLE that contains the desired fixed offset distance of the beam expressed in the same units as the *normal* motor.
- **Parameters record** - This record must be present, but its contents are not used by this dependency. Typically, a dummy variable will be used here as in the example below.

An example set of records for the Bragg parallel dependency looks like

```
parallel_list    variable inline record        "" "" 1  4  parallel_enabled parallel_type c
parallel_type    variable inline int           "" "" 1  1  5
parallel_enabled variable net_variable net_int "" "" localhost parallel_enabled.value 1  1
parallel_records variable inline record        "" "" 1  2  parallel beam_offset
dummy_params     variable inline double        "" "" 1  1  0
```

A corresponding beam offset variable would look like

```
beam_offset variable net_variable net_double "" "" localhost beam_offset.value 1  1  -35000
```

If used in combination with a Bragg normal dependency, the two dependencies should use the same MX variable to control the beam offset.

The Bragg parallel position is computed from the beam offset and the monochromator theta angle via the equation bragg_parallel = beam_offset / (2.0 * sin(theta))

Type 6 - Experiment table height dependency

If a given monochromator does not support fixed exit beam operation, an alternate way to ensure that the X-ray beam hits the desired target is to put the experiment on a table that can be vertically translated to track the beam.

- **Record list record** - This is a 1-dimensional array of type MXFT_RECORD containing three elements:
  - *table height record* - This motor record controls the vertical height of the experiment table.
  - *table offset record* - This is a variable record of type MXFT_DOUBLE that provides a way of adding a constant offset to the computed table height.
  - *crystal separation record* - This is a variable record of type MXFT_DOUBLE that contains the perpendicular crystal separation distance expressed in the same units as the *table height* motor.
- **Parameters record** - This record must be present, but its contents are not used by this dependency. Typically, a dummy variable will be used here as in the example below.

An example set of records for the experiment table height dependency looks like

```
theight_list    variable inline record        "" "" 1  4  theight_enabled theight_type dum
theight_type    variable inline int           "" "" 1  1  6
theight_enabled variable net_variable net_int "" "" localhost theight_enabled.value 1  1
theight_records variable inline record        "" "" 1  3  theight toffset crystal_sep
dummy_params    variable inline double        "" "" 1  1  0
```

A corresponding pair of variables would look like

```
toffset     variable net_variable net_double "" "" localhost toffset.value 1  1  0
crystal_sep variable net_variable net_double "" "" localhost crystal_sep.value 1  1  5000
```

The experiment table height position is computed via the equation
table_height = table_offset + 2.0 * crystal_separation * cos(theta)

Type 7 - Diffractometer theta dependency

The diffractometer theta dependency is used to control the Bragg angle of a diffractometer or goniostat in the experimental hutch so that it is set to the correct angle to diffract the X-ray beam coming from the monochromator. This dependency assumes that, in general, the crystal on the diffractometer will have a different d spacing than that of the monochromator crystal.

If the diffractometer angle is called *htheta* and the diffractometer d spacing is called *hd_spacing*, the diffractometer angle is computed by the equation
sin(htheta) = d_spacing * sin(theta) / hd_spacing
The raw value of *htheta* is adjusted using a linear equation of the form
htheta_adjusted = diffractometer_scale * htheta + diffractometer_offset

- **Record list record** - This is a 1-dimensional array of type MXFT_RECORD containing three elements:
  - *diffractometer theta motor record* - This the a motor record that controls the diffractometer theta angle.
  - *monochromator d spacing record* - This is a variable record that contains the d spacing of the monochromator crystal in angstroms.
  - *diffractometer d spacing record* - This is a variable record that contains the d spacing of the diffractometer crystal in angstroms.
- **Parameters record** - This is a 1-dimensional array of type MXFT_DOUBLE which contains two elements which are used to compute the adjusted diffractometer angle:
  - *diffractometer scale*
  - *diffractometer offset*

An example set of records for the diffractometer theta dependency looks like

```
htheta_list     variable inline record          "" "" 1 4 htheta_enabled htheta_type hthe
htheta_type     variable inline int             "" "" 1 1 7
htheta_enabled  variable net_variable net_int   "" "" localhost htheta_enabled.value 1  1
htheta_records  variable inline record          "" "" 1 3 htheta d_spacing hd_spacing
htheta_params   variable net_variable net_double "" "" localhost htheta_params.value 1  2
```

Type 8 - Energy polynomial dependency

The energy polynomial dependency is similar to the type 2 polynomial dependency described above, except the dependent motor position is a polynomial function of the monochromator X-ray energy. Thus, the dependent position will have the form

dependent_position = c0 + c1 * energy + c2 * (energy**2) + c3 * (energy**3) + ...
This dependency assumes that the energy is expressed in eV.

- **Record list record** - This is a 1-dimensional array containing two elements:
  - *dependent motor record* - This motor record controls the dependent motor whose position is determined by the energy polynomial.
  - *monochromator d spacing record* - This is a variable record that contains the d spacing of the monochromator crystal in angstroms. This record is used to convert the theta angle in degrees to energy in eV.
- **Parameters record** - This is a 1-dimensional array of type MXFT_DOUBLE which contains the coefficients of the polynomial. The coefficients are specified in order starting with the constant term and continuing up to the coefficient of the highest order term.

An example set of records for the energy polynomial dependency looks like

```
focus_list     variable inline record           "" "" 1 4 focus_enabled focus_type focus_pa
focus_type     variable inline int              "" "" 1 1 8
focus_enabled  variable net_variable net_int    "" "" localhost focus_enabled.value 1  1  0
focus_records  variable inline record           "" "" 1 2 focus d_spacing
focus_params   variable net_variable net_double "" "" localhost focus_params.value 1  4  0
```

Type 9 - Option selector dependency

The option selector dependency is used together with a "position_select" calculation record to switch an external value between multiple settings depending on the current value of the monochromator theta position. The "position_select" variable has an integer value from 1 to N. For example, this could be used to automatically switch between mirror stripes at certain X-ray energies.

- **Record list record** - This is a 1-dimensional array of type MXFT_RECORD containing two elements:
  - *option selector record* - This should be a calculation variable record of type "position_select" that controls the external value and contains a list of allowed values for the external value.
  - *option range record* - This is the name of the parameters record below.
- **Parameters record** - This is a 2-dimensional Nx2 array of type MXFT_DOUBLE which contains pairs of elements that describe the limits for each allowed theta range. The two values for each theta range are
  - *theta range lower limit*
  - *theta range upper limit*

  The number N is the number of theta ranges.

An example set of records for the option selector dependency looks like

```
stripe_list    variable inline record "" "" 1  4  stripe_enabled stripe_type dummy_params
stripe_type    variable inline int "" "" 1  1  9
stripe_enabled variable inline int "" "" 1  1  1
stripe_records variable inline record "" "" 1  2  stripe_select stripe_params
stripe_params  variable inline double "" "" 2  4  2  0 5  4.5 8.5  8 11  10.5 15
stripe_select  variable calc position_select "" "" stripe 4 300 600 900 1200 1 1 -1
stripe         device motor soft_motor "" "" 0 0 -1000000000 1000000000 0 -1 -1 0.01 0 um
```

The example above is for an X-ray mirror whose transverse position is determined by a motor record called *stripe*. There are 4 allowed positions for *stripe*, namely, 300, 600, 900, and 1200. The allowed *theta* ranges corresponding to the allowed positions are 0 to 5 degrees, 4.5 to 8.5 degrees, 8 to 11 degrees, and 10.5 to 15 degrees. The *stripe* position to be selected is determined by comparing the current position of *theta* to the parameter ranges in the variable *stripe_params*. According to *stripe_params*, the *stripe* motor is allowed to be at 300 if *theta* is between 0 and 5, or it is allowed to be at 600 if *theta* is between 4.5 and 8.5, and so forth. If *theta* moves outside the allowed range of positions for the current selection, the option selector will switch to the next selection.

Notice that the allowed ranges for *theta* overlap. This is to provide a deadband for switches between option selector ranges. As an example, suppose *theta* is currently at 7 degrees. This means that *theta* is within the second option selector range of 4.5 to 8.5 degrees and that the *stripe* motor should currently be at 600. However, if *theta* is moved to 9 degrees, this is outside the current option selector range, so *stripe* will be moved to the next allowed position of 900. However, the lower end of the new range, namely, 8 degrees, is below the upper end of the original range, namely, 8.5 degrees. This means that *theta* must be moved below 8 degrees before the *stripe* position will be moved back to the previous value of 600.

Note that if *theta* is below 0 degrees, the *stripe* motor will be sent to 300, while if *theta* is above 15 degrees, the *stripe* motor will be moved to 1200.

Bugs

The MX database for the monochromator pseudomotor system has turned out to be much more complex to set up than I had originally wanted. It is my intention to revise this pseudomotor to be easier to configure, but I have no time estimate as to when that will happen.

Some of the dependencies are user configurable via the parameters record while others are configured via additional records added to the record list.

2.3.10 Q Motor

This is an MX pseudomotor driver for the momentum transfer parameter q, which is defined as

$$q = \frac{4\pi \sin(\theta)}{\lambda} = \frac{2\pi}{d}$$

where λ is the wavelength of the incident X-ray, θ is the Bragg angle for the analyzer arm and d is the effective crystal d-spacing that is currently being probed.

Warning: θ above must not be set to the angle of the analyzer arm. Instead, it must be set to half of that angle which is the nominal Bragg angle.

2.3.11 Record Field Motor

2.3.12 Segmented Move

2.3.13 Slit Motor

slit_type field - This integer field select the type of slit pseudomotor that this record represents. There are four possible values for the *slit_type* field, which come in two groups of two, namely, the SAME case and the OPPOSITE case. The allowed values are:

- 1 - *(MXF_SLIT_CENTER_SAME)*
- 2 - *(MXF_SLIT_WIDTH_SAME)*
- 3 - *(MXF_SLIT_CENTER_OPPOSITE)*
- 4 - *(MXF_SLIT_WIDTH_OPPOSITE)*

The SAME cases above are for slit blade pairs that move in the same physical direction when they are both commanded to perform moves of the same sign. The OPPOSITE cases are for slit blade pairs that move in the opposite direction from each other when they are both commanded to perform moves of the same sign.

As an example, for a top and bottom slit blade pair, if a positive move of each causes both blades to go up, you use the SAME case. On the other hand, if a positive move of each causes the top slit blade to go up and the bottom slit blade to go down, you use the OPPOSITE case.

2.3.14 Table Motor

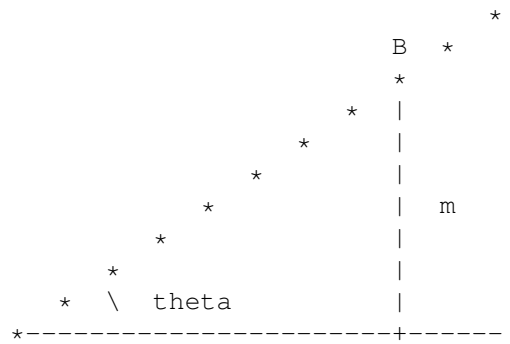
2.3.15 Tangent Arm/Sine Arm

This is an MX motor driver to move a tangent arm or sine arm pseudomotor.

IMPORTANT: The moving motor position and the arm length must both be specified using the *same* user units, while the angle offset must be specified in radians. Thus, if the moving motor position is specified in micrometers, then the arm length must be specified in micrometers as well.

If you want to specify the angle offset in degrees rather than in radians, the simplest way is to create a 'translation_mtr' record containing only the raw angle offset motor. Then, set a scale factor in the translation motor record that converts from radians to degrees.

A tangent arm consists of two arms that are connected at a pivot point as follows:

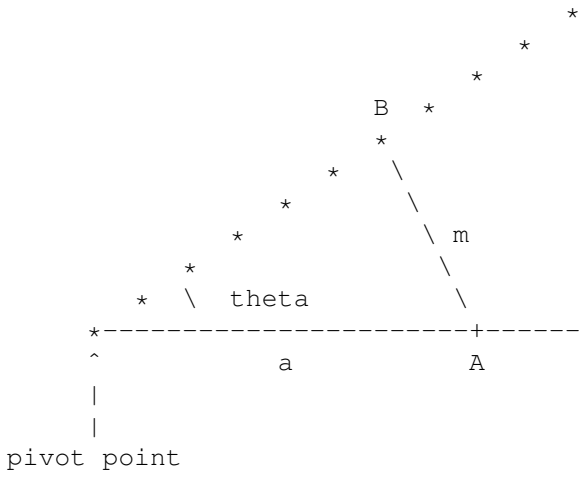


A linear motor is attached to the fixed arm at point A and then moves a push rod that pushes the moving arm at point B. In this geometry, the position of the moving motor, m, is related to the angle theta by the relationship

$$\tan(\theta) = \frac{m}{a}$$

where a is the distance of the motor on the arm it is attached to from the pivot point. The important consideration here is that the linear motion is perpendicular to the fixed arm.

A sine arm is similar except that the linear motion is now perpendicular to the moving arm:



This changes the equation to the form

$$\sin(\theta) = \frac{m}{a}$$

hence the name “sine arm”.

Note: The ’tangent_arm’ and ’sine_arm’ drivers share the same code and distinguished in the driver code by their different driver types, namely, MXT_MTR_TANGENT_ARM and MXT_MTR_SINE_ARM.

2.3.16 Theta-Two Theta

2.3.17 Translation

2.3.18 Wavelength

2.3.19 Wavenumber

2.3.20 XAFS Wavenumber

This MX pseudomotor driver controls another MX motor in units of XAFS electron wavenumber.

The XAFS electron wavenumber k is computed using the equation:

$$E_{photon} = E_{edge} + \frac{\hbar^2 k^2}{2m_{electron}}$$

Chapter 3

Counter/Timers

3.1 Am9513

The following is an example database for the IIT BCPS setup for Am9513 boards:

```
ports   interface portio linux_portio "" "" /dev/portio
am9513  interface generic am9513 "" "" ports 0x284 0x1b0
i8255   interface generic i8255 "" "" ports 0x280
#
# Motor 1 uses Am9513 counters 1 & 2 to generate the motor step pulses while
# 8255 output bit 2 of port C is used to generate the direction signal.
#
motor1  device motor am9513_motor "" "" 0 0 -1000000 1000000 0 -1 -1 0.005 0 um 2 am9513:1
portc   device digital_output i8255_out "" ""  0 i8255 C
#
# Scaler 1 is a 32 bit scaler created using Am9513 counters 4 & 5.  The
# counter is gated by the gate input for its low order counter (GATE4),
# while external pulses to be counted are fed to the source input for
# its low order counter (SRC4).
#
scaler1 device scaler am9513_scaler "" "" 0 0 0 2 am9513:4 am9513:5 0x4 0x4
#
# Timer 1 is a 16 bit timer created using Am9513 counter 3.  It is using
# a 5 MHz clock signal.
#
timer1  device timer am9513_timer "" "" 1  am9513:3 5000000
```

Warning: The *am9513* interface driver has only been fully implemented and tested for Am9513-based systems using 8-bit bus access.

At present, MX Am9513 timers can only use one 16-bit counter. Also note that the timer driver relies on the output for the timer's counter being connected to its own gate input. That is, OUT(n) must be connected to GATE(n)

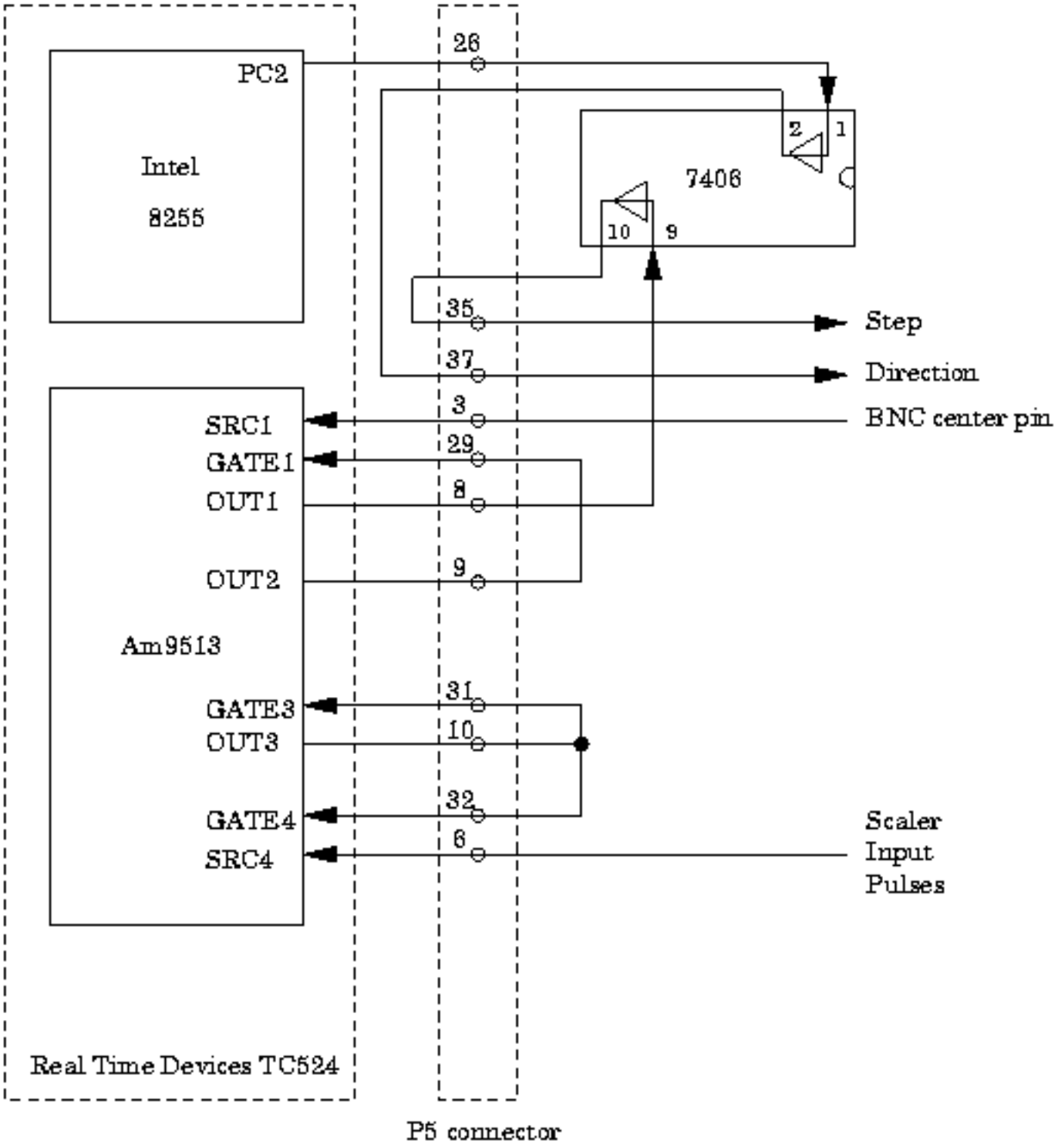


Figure 3.1: Wiring diagram used by the IIT BCPS department

for the timer to work. Of course, OUT(n) is also connected to the GATE inputs of the scalers that this timer is gating.

3.2 Black Cat Systems GM-xx

3.3 Blu-Ice Timer

3.4 DSP QS450 or Kinetic Systems 3610

3.5 EPICS Scaler

The MX EPICS scaler support optionally can make use of globally visible dark current values. This is done by loading an additional EPICS database file in "st.cmd" that can be found in the MX base distribution in the file *mx/driver_info/epics_scaler/Jscaler_dark.db*. This EPICS database implements two additional records per EPICS scaler channel. For example, for scaler channel 2 the records are

- (P)(S)_Dark2.VAL - Dark current per second for scaler channel 2.

- (P)(S)_SD2.VAL - The dark current subtracted value for scaler 2.

where $(P) and $(S) are defined to have the same values as in the standard Jscaler.db database. The database is loaded into the EPICS VME crate by adding a line to the 'st.cmd' startup script that looks like

```
dbLoadRecords("iocBoot/ioc1/Jscaler_dark.db","P=s10id:,S=scaler1,C=0", top)
```

Please note that this database contains a definition for the scaler record $(P) and $(S) itself and thus is not immediately compatible with the standard *Jscaler.db* database. This is due to the fact that EPICS does not supply any way for an add-on database to add forward links to existing records. If you wish to combine *Jscaler.db* and *Jscaler_dark.db*, the simplest way is to merely move the FLNK field whose value is "(P)(S)_cts1.PROC" in Jscaler.db to the LNK4 field of Fanout record "(P)(S)_fan0" defined in *Jscaler_dark.db*.

Hopefully, something equivalent to the dark current fields in *Jscaler_dark.db* will be added to some future version of *Jscaler.db*.

3.6 EPICS Timer

3.7 Interval Timer

3.8 Joerger VSC8/16

3.9 MCA Timer

3.10 MCS Timer

3.11 Network Scaler

3.12 Network Timer

3.13 Ortec 974

3.14 Prairie Digital Model 45

3.15 PFCU Shutter Timer

3.16 Radix Databox Scaler/Timer

3.17 RTC-018

3.18 SCIPE Scaler

3.19 SCIPE Timer

3.20 Soft Scaler

3.21 Soft Timer

3.22 Spec Scaler

3.23 Spec Timer

3.24 XIA DXP Timer

3.25 XIA Handel Timer

3.26 Pseudoscalers

3.26.1 Autoscale Related Pseudoscalers

3.26.2 MCA Related Pseudoscalers

3.26.3 MCS Scaler

3.26.4 Scaler Function

WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING

This driver does **NOT** attempt to ensure that all of the timers start at exactly the same time. This means that devices gated by different timers may not be gated on for exactly the same timer interval, although the lengths of time they are gated on for should be the same. The result is that you may get **SYSTEMATIC ERRORS** if you do not use this driver intelligently. It is up to you to decide whether or not this makes a difference to the experiment you are performing. The *best* solution is to make sure that all of your measuring devices are gated by the same hardware timer, but if that is not possible, then this driver may be useful as a stopgap.

Caveat emptor.

Chapter 4

Pulse Generator

4.1 Network Pulse Generator

4.2 Prairie Digital Model 45 Pulse Generator

4.3 Struck SIS3801

4.4 Struck SIS3807

Chapter 5

Encoder

5.1 Kinetic Systems 3640

Chapter 6

Analog I/O

6.1 APS ADCMOD2 Analog I/O

6.2 Data Track Tracker Analog I/O

6.3 Kinetic Systems 3112 Analog Output

6.4 Kinetic Systems 3512 Analog Input

6.5 MODBUS Analog I/O

6.6 Multichannel Analog Input Function

6.7 Network Analog I/O

6.8 Newport Electronics Iseries Analog I/O

6.9 Prairie Digital Model 45 Analog I/O

6.10 Soft Analog I/O

6.11 Stanford Research Systems SR-630

The SR-630 is a 16-channel thermocouple readout controller.

An example database for the SR-630 looks like:

```
sr630_rs232 interface rs232 tty "" "" 9600 8 N 1 N 0x0a 0x0a 10 0x0 /dev/ttyS7
sr630 interface generic sr630 "" "" sr630_rs232
temp1  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 1
```

```
temp2  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 2
temp3  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 3
temp4  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 4
temp5  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 5
temp6  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 6
temp7  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 7
temp8  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 8
temp9  device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 9
temp10 device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 10
temp11 device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 11
temp12 device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 12
temp13 device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 13
temp14 device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 14
temp15 device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 15
temp16 device analog_input sr630_ainput "" "" 0 1 0 C 0x0 0 "" sr630 16
```

6.12 Wago 750 Series MODBUS Analog Output

6.13 Motor Controller Analog I/O

6.14 Other Controller Type Analog I/O

Chapter 7

Multichannel Analog Input

7.1 Keithley 2700

7.2 Oxford Danfysik QBPM

Chapter 8

Digital I/O

Chapter 9

Relays

9.1 Binary Relay

9.2 Blind Relay

9.3 Blu-Ice Shutter

9.4 Generic Relay

9.5 MarCCD Relay

9.6 MarDTB Shutter

9.7 Network Relay

9.8 PFCU Filter and Shutter

9.9 Pulsed Relay

Chapter 10

Amplifiers

Chapter 11

Single Channel Analyzers

11.1 Network SCA

11.2 Oxford Danfysik Cyberstar X1000

11.3 Soft SCA

Chapter 12

Multichannel Analyzers

12.1 EPICS MCA

12.2 Network MCA

12.3 Ortec UMCBI (Trump)

12.4 Röntec RCL-22 MCA

12.5 Soft MCA

12.6 X-Ray Instrumentation Associates (XIA)

12.7 MCA Associated Records

12.7.1 MCA Alternate Time

12.7.2 MCA Channel

12.7.3 MCA Region of Interest Integral

12.7.4 MCA Value

Chapter 13

Multichannel Encoders

Chapter 14

Multichannel Scalers

14.1 EPICS MCS

The MX EPICS MCS support optionally can make use of globally visible dark current values. This is done by loading an additional EPICS analog output record per MCS channel which is used to store the dark current value. This is most easily described by giving an example.

Suppose you have a set of MCS records loaded in the EPICS "st.cmd" script that look like

```
dbLoadRecords("mcaApp/Db/mca.db","P=s10id:,M=mcs1,CARD=0,SIGNAL=0,DTYPE=Struck STR7201 MCS,
dbLoadRecords("mcaApp/Db/mca.db","P=s10id:,M=mcs2,CARD=0,SIGNAL=1,DTYPE=Struck STR7201 MCS,
dbLoadRecords("mcaApp/Db/mca.db","P=s10id:,M=mcs3,CARD=0,SIGNAL=2,DTYPE=Struck STR7201 MCS,
dbLoadRecords("mcaApp/Db/mca.db","P=s10id:,M=mcs4,CARD=0,SIGNAL=3,DTYPE=Struck STR7201 MCS,
```

Then, all that you need to add to support dark currents for these channels is to add something like the following lines to "st.cmd".

```
dbLoadRecords("iocBoot/ioc1/mcs_dark.db","P=s10id:,M=mcs1", top)
dbLoadRecords("iocBoot/ioc1/mcs_dark.db","P=s10id:,M=mcs2", top)
dbLoadRecords("iocBoot/ioc1/mcs_dark.db","P=s10id:,M=mcs3", top)
dbLoadRecords("iocBoot/ioc1/mcs_dark.db","P=s10id:,M=mcs4", top)
```

The *mcs_dark.db* database file is extremely simple. The entire contents of the file is:

```
grecord(ao,"$(P)$(M)_Dark") {
        field(PREC,"3")
}
```

A copy of this file may be found in the MX base source distribution in the file *mx/driver_info/epics_mcs/mcs_dark.db*.

14.2 Network MCS

14.3 Radix Databox MCS

14.4 Scaler Function MCS

14.5 SIS3801

14.6 Soft MCS

14.7 X-ray Instrumentation Associates MCS

Chapter 15

CCD

15.1 Network CCD

15.2 Remote MarCCD

Chapter 16

Goniostat/Diffractometer Tables

16.1 IMCA-CAT ADC Table at APS Sector 17

The ADC table is designed to support a standard crystallography goniostat. At present, it is the support for an ADSC Quantum 105 detector system. The geometry of the table is shown by the figure below: The geometry is described further in a technical note in PDF format. The note is also available in Postscript if you prefer.

The MX table support for ADC tables uses two different kinds of records, namely, an ADC specific table record described here and the generic table motor record described in the motor section.

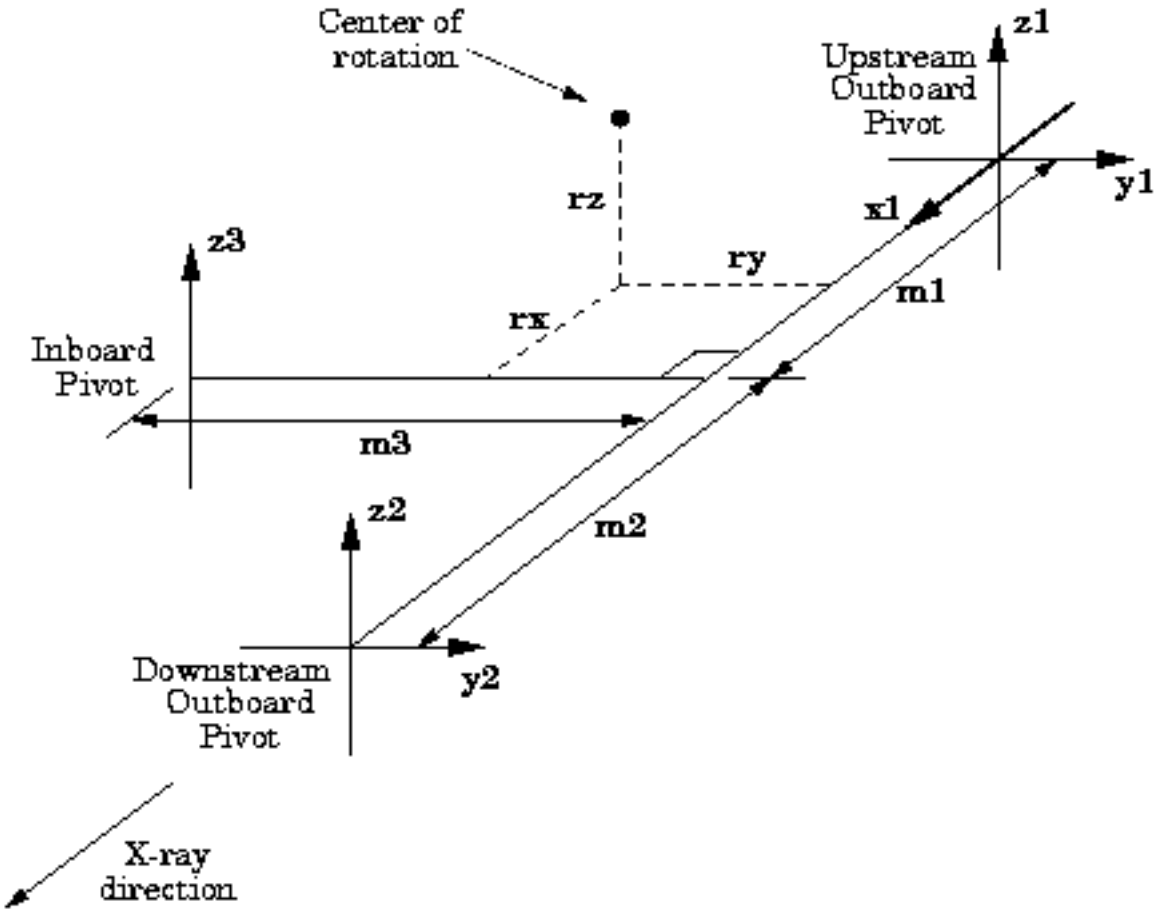


Figure 16.1: IMCA-CAT ADC table geometry

16.1.1 Record Fields in the Record Description

Field Name	Field Type	Number of Dimensions	Sizes	Description
name	string	1	16	The name of the record
mx_superclass	recordtype	0	0	The string "device"
mx_class	recordtype	0	0	The string "table"
mx_type	recordtype	0	0	The string "adc_table"
label	string	1	40	A verbose description of the record.
acl_description	string	1	40	Placeholder for an access control list *(not yet implemented)*.
motor_record_array	record	1	6	The names of the raw motor records used by this table record listed in the order X1, Y1, Y2, Z1, Z2, and Z3.
m1	double	0	0	Distance from the table zero point to the Z1 pivot point.
m2	double	0	0	Distance from the table zero point to the Z2 pivot point.
m3	double	0	0	Distance from the table zero point to the Z3 pivot point.
rx	double	0	0	X component of the distance from the table zero point to the rotation center.
ry	double	0	0	Y component of the distance from the table zero point to the rotation center.
rz	double	0	0	Z component of the distance from the table zero point to the rotation center.

An example database for this table type would look like:

```
adsc_table device table adc_table "" ""  x1 y1 y2 z1 z2 z3 0.4 0.6 0.75 0.25 0.25 0.5
x1         device motor soft_motor "" "" 0 0 -20000000 20000000 0 -1 -1 0.001 0 um
y1         device motor soft_motor "" "" 0 0 -20000000 20000000 0 -1 -1 0.001 0 um
y2         device motor soft_motor "" "" 0 0 -20000000 20000000 0 -1 -1 0.001 0 um
z1         device motor soft_motor "" "" 0 0 -20000000 20000000 0 -1 -1 0.001 0 um
z2         device motor soft_motor "" "" 0 0 -20000000 20000000 0 -1 -1 0.001 0 um
z3         device motor soft_motor "" "" 0 0 -20000000 20000000 0 -1 -1 0.001 0 um
tx         device motor table_motor "" "" 0 0 -20000000 20000000 0 -1 -1 1 0 um adsc_table
ty         device motor table_motor "" "" 0 0 -20000000 20000000 0 -1 -1 1 0 um adsc_table
tz         device motor table_motor "" "" 0 0 -20000000 20000000 0 -1 -1 1 0 um adsc_table
troll      device motor table_motor "" "" 0 0 -20000000 20000000 0 -1 -1 1 0 urad adsc_tabl
tpitch     device motor table_motor "" "" 0 0 -20000000 20000000 0 -1 -1 1 0 urad adsc_tabl
tyaw       device motor table_motor "" "" 0 0 -20000000 20000000 0 -1 -1 1 0 urad adsc_tabl
```

In this example, the *adsc_table* record is the actual table record itself. It is configured to use soft motors *x1*, *y1*, *y2*, *z1*, *z2*, and *z3* as the raw motors. The table parameters for the example are set to *m1 = 0.4*, *m2 = 0.6*, *m3 = 0.75*, *rx = 0.25*, *ry = 0.25*, and *rz = 0.5*.

Chapter 17

Autoscale Devices

17.1 Autoscale Amplifier

17.2 Autoscale Filter

17.3 Autoscale Filter and Amplifier

17.4 Related Devices

17.4.1 Autoscale Scaler

17.4.2 Gain Tracking Scaler

MX scaler driver to control MX gain tracking scalers. Gain tracking scalers are pseudoscalers that rescale their reported number of counts to go up and down when an associated amplifier changes its gain.

For example, suppose that the real scaler was reading 1745 counts, the amplifier was set to 10^8 gain and the gain tracking scale factor was 10^{10}. Then, the gain tracking scaler would report a value of 174500 counts. If the amplifier gain was changed to 10^9, then the gain tracking scaler would report a value of 17450 counts.

Gain tracking scalers are intended to be used in combination with autoscaling scalers, so that when the autoscaling scaler changes the gain of the amplifier, the values reported by gain tracking scalers will change to match.

Chapter 18

Sample Changers

18.1 Network

18.2 Sercat ALS Robot

Chapter 19

Pan-Tilt-Zoom Controllers

Chapter 20

Video Input Devices

20.1 EPIX XCLIB

20.2 Network Video Input

20.3 Soft Video Input

20.4 Video4Linux 2

Chapter 21

Area Detector

21.1 Aviex PCCD-170170

21.2 Network Area Detector

21.3 Soft Area Detector

Chapter 22

RS-232

Chapter 23

GPIB

23.1 EPICS GPIB

23.2 Iotech Micro488EX GPIB

23.3 Keithley 500-SERIAL

23.4 Linux GPIB

23.5 Linux Lab Project GPIB

Warning: This MX driver was originally developed for the Linux 2.0.x device driver provided by the Linux Lab Project, but that project seems to have stalled. Fortunately, the code seems to have been picked up by the Linux GPIB Package Homepage *(http://linux-gpib.sourceforge.net/)* which supports Linux 2.4.x. However, this MX driver has not yet been tested with the new Linux 2.4.x version of the device driver.

The Linux Lab Project GPIB driver does not provide an equivalent to the ibdev() function that finds a GPIB device by address. That makes the Linux Lab Project driver the only one that does not provide such an interface, so it is easier just to provide one for it. This is handled by adding to */etc/gpib.conf* the contents of the file *mx/driver_info/llp_gpib/gpib.conf_addon* from the MX base source distribution, so that there is a way to find a given GPIB device by its primary address. The contents of the file *gpib.conf_addon* is shown in the following figure:

23.6 National Instruments GPIB

23.7 Network GPIB

```
/* Devices for use with the MX Linux GPIB driver. */

device { name = gpib0.1     pad=1    sad=0 }
device { name = gpib0.2     pad=2    sad=0 }
device { name = gpib0.3     pad=3    sad=0 }
device { name = gpib0.4     pad=4    sad=0 }
device { name = gpib0.5     pad=5    sad=0 }
device { name = gpib0.6     pad=6    sad=0 }
device { name = gpib0.7     pad=7    sad=0 }
device { name = gpib0.8     pad=8    sad=0 }
device { name = gpib0.9     pad=9    sad=0 }
device { name = gpib0.10    pad=10   sad=0 }
device { name = gpib0.11    pad=11   sad=0 }
device { name = gpib0.12    pad=12   sad=0 }
device { name = gpib0.13    pad=13   sad=0 }
device { name = gpib0.14    pad=14   sad=0 }
device { name = gpib0.15    pad=15   sad=0 }
device { name = gpib0.16    pad=16   sad=0 }
device { name = gpib0.17    pad=17   sad=0 }
device { name = gpib0.18    pad=18   sad=0 }
device { name = gpib0.19    pad=19   sad=0 }
device { name = gpib0.20    pad=20   sad=0 }
device { name = gpib0.21    pad=21   sad=0 }
device { name = gpib0.22    pad=22   sad=0 }
device { name = gpib0.23    pad=23   sad=0 }
device { name = gpib0.24    pad=24   sad=0 }
device { name = gpib0.25    pad=25   sad=0 }
device { name = gpib0.26    pad=26   sad=0 }
device { name = gpib0.27    pad=27   sad=0 }
device { name = gpib0.28    pad=28   sad=0 }
device { name = gpib0.29    pad=29   sad=0 }
device { name = gpib0.30    pad=30   sad=0 }
device { name = gpib0.31    pad=31   sad=0 }
```

Figure 23.1: *gpib.conf_addon* for the MX Linux Lab Project GPIB driver

Chapter 24

Port I/O

24.1 DriverLINX Port I/O

This MX driver is an interface to the DriverLINX port I/O driver for Windows NT/98/95 written by Scientific Software Tools, Inc. The DriverLINX package may be downloaded from http://www.sstnet.com/dnload/dnload.htm. This driver is primarily intended for use under Windows NT, since the 'dos_portio' driver already handles Windows 98/95, but it should work on all three operating systems.

Warning: These drivers have not yet been tested with Windows 2000 or Windows XP.

24.2 MSDOS Port I/O

24.3 Linux iopl() and ioperm() drivers

24.4 Linux portio driver

24.5 VxWorks Port I/O

Chapter 25

USB

25.1 Libusb

Chapter 26

VME

Chapter 27

MODBUS

27.1 MODBUS Serial RTU

27.2 MODBUS/TCP

Chapter 28

CAMAC

28.1 DSP6001

28.2 ESONE

28.3 Soft CAMAC

Chapter 29

Camera Link

29.1 Camera Link API

29.2 EPIX Camera Link

29.3 Soft Camera Link

Chapter 30

Variables

30.1 EPICS Variables

30.2 Inline Variables

30.3 Network Variables

30.4 PMAC Variables

30.5 Spec Variables

30.6 Calculation Variables

30.6.1 APS Topup Time to Inject

30.6.2 APS Topup Interlock

30.6.3 Mathop Variables

30.6.4 Polynomial

30.6.5 Position Select

Chapter 31

Servers

31.1 TCP/IP Servers

31.2 Unix Domain Socket Servers

Chapter 32

Scans

32.1 Linear Scans

32.1.1 Input Scans

32.1.2 Motor Scans

32.1.3 Pseudomotor Scans

32.1.4 Slit Scans

32.1.5 Theta-Two Theta Scans

32.2 List Scans

32.2.1 File List Scans

32.3 XAFS Scans

32.4 Quick Scans *(also known as Fast or Slew Scans)*

32.4.1 Joerger Quick Scans

32.4.2 MCS Quick Scans

Chapter 33

Interfaces to Other Control Systems