MX User's Guide

for MX 2.0

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

November 13, 2013

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

```
Copyright 1999 Illinois Institute of Technology

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL ILLINOIS INSTITUTE OF TECHNOLOGY BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of Illinois Institute
of Technology shall not be used in advertising or otherwise to promote
the sale, use or other dealings in this Software without prior written
authorization from Illinois Institute of Technology.
```

Contents

Chapter 1

Introduction

MX is a data acquisition and control toolkit designed for both small to large experimental systems. It provides support for many different kinds of devices such as motor, counter/timers, multichannel analyzers, area detectors and more. It also provides many different scanning modes such as step scanning and slew/quick scanning.

First, let's give some examples of how you can use MX.

1.1 Example 1: Single Process Mode

Suppose you have this situation:

You are performing a powder diffraction measurement in a university laboratory using an X-ray generator. You have a quasi-monochromatic beam that has been created using an appropriate set of filters. This beam is aimed at your powder sample and you want to mount a scintillation counter on a 2θ arm to measure the intensity of the powder pattern reflections as a function of angle. You will move the your stepping motor using an Advanced Control Systems MCU-2 motor controller and count the scintillation pulses using an Ortec 974 NIM module. The simplest possible way of supporting this situation is to use a single process to handle both device control and a text-mode user interface.

The following is an example MX database for this situation. A copy of this database file can be found in the MX source code tree at *mx/test/drivers/muser_example1.dat*. It can be run using the script *mx/test/drivers/muser_example1* **if** you have an MCU-2 and an Ortec 974 connected to a Linux computer.

```
#
# Motion control
#
mcu2_rs232 interface rs232 tty "" "" 19200 8 N 1 N 0x0d 0x0d 5.0 0x0 /dev/ttyS0
theta  device motor mcu2 "" "" 0 0 -180000 180000 0 -1 -1 0.001 0 deg mcu2_rs232 1 0x0
2theta device motor mcu2 "" "" 0 0 -10000 160000 0 -1 -1 0.001 0 deg mcu2_rs232 2 0x0
#
# Counter/timer control
#
ortec_rs232 interface rs232 tty "" "" 9600 8 N 1 S 0xd 0xd 5.0 0x0 /dev/ttyS1
ortec       interface controller ortec974 "" "" ortec_rs232
scaler2     device scaler ortec974_scaler "" "" 0 0 1 timer1 ortec 2
scaler3     device scaler ortec974_scaler "" "" 0 0 1 timer1 ortec 3
```

```
scaler4     device scaler ortec974_scaler "" "" 0 0 1 timer1 ortec 4
timer1      device timer ortec974_timer "" "" ortec
```

The first few lines of the file describe the interface to the motor controller and the motors.

```
mcu2_rs232 interface rs232 tty "" "" 19200 8 N 1 N 0x0d 0x0d 5.0 0x0 /dev/ttyS0
```

On a Linux or Unix computer, this line creates an MX RS-232 record named *mcu2_rs232* that is used to communicate with the MCU-2 controller via a Unix TTY interface. It requests the following settings for this RS-232 interface.

- `19200 8 N 1 N` - These set up the serial port using the traditional set of serial port parameters.
  - 19200 bits per second port speed
  - 8 bit characters.
  - No parity.
  - 1 stop bit.
  - No flow control.
- `0x0d 0x0d` - This specifies that ASCII messages read from the MCU-2 will have a line terminator of a single carriage return character (ASCII 10, Hex 0xD). Similarly, ASCII commands sent to the MCU-2 will also be terminated with a carriage return character.
- `5.0` - This requests an RS-232 timeout of 5 seconds. If the driver software expects a response to a command that MX has sent and does not receive it within 5 seconds, the read in the MX driver would return an MXE_TIMED_OUT error to the user.

  If you do not want a timeout, then just set the timeout to a negative number.
- `0x0` - This is the *rs232_flags* field. Setting this to 0x0 sets a number of internal MX RS-232 parameters to their default values.
- `/dev/ttyS0` - This is the name of the Linux or Unix TTY port used to communicate with the MCU-2 controller.

On a Windows computer, you would replace the above record definition with this slightly different one.

```
mcu2_rs232 interface rs232 win32_com "" "" 19200 8 N 1 N 0x0d 0x0d 5.0 0x0 COM1
```

If you were using a TCP/IP socket to connect to a remote serial port on TCP port 5002 at host 192.168.150.3, then this line would become

```
mcu2_rs232 interface rs232 tcp232 "" "" 19200 8 N 1 N 0x0d 0x0d 5.0 0x0 192.168.150.3 5002
```

There are many other kinds of RS-232 drivers in MX which we do not describe at this point.

... more stuff to go here ...

The current syntax for the database files is relatively terse. There is a plan to add support for a Windows-INI file style syntax has not happened yet.

1.2 Example 2: MX Client/Server Mode

1.3 Example 3: Using a Foreign Control System

Chapter 2

Scans

An MX scan is a database object that sequences through a series of measurements and then writes out the measurement data to an external object such as a file. The measurements can come from almost any type of device supported by MX, such as analog and digital I/O, motor positions, counter/timer readouts, MCA spectra, and so forth.

MX scans currently support three kinds of measurements, which differ depending on the way that the measurements are triggered or gated.

- *preset time* - In this mode, an MX timer device is used to specify the measurement duration in seconds. Typically, but not always, the timer will generate a gate pulse for the duration of the measurement.

- *preset count* - In this mode, the measurement continues until the specified MX scaler acquires a prespecified number of counts. In general, the scaler will generate a gate pulse until the specified number of counts have occurred.

- *preset pulse period* - Some multichannel scaler devices can be driven by an external pulse generator. Multichannel scaler scans that are driven by an external pulse generator will use this kind of measurement.

Several scan classes are currently available, namely, *motor scans*, *list scans*, *multichannel scaler scans*, *XAFS scans* and *area detector scans*. These scan classes differ in terms of the details of how the scan progresses.

In addition, there are *area detector sequences* which technically are not scans, but they perform actions simular to a scan.

The details of each scan type are described below.

2.1 Linear Scans

An MX *linear scan* takes a sequence of measurements that are *equally spaced* in terms of some independent variable. There are several different types of linear scans.

2.1.1 Motor scans

For *motor scans*, the independent variables are the positions of one or more motors. As an example, suppose you wanted to measure X-ray absorption as a function of X-ray energy over the range from 8000 eV to 9000 eV in 100 eV steps. For this case, you would create an MX motor scan with a start position of 8000 eV with a step size of 100 eV and a total of 11 scan steps.

Another example would be using a temperature controller to change a sample's temperature from 290 K to 310 K in 1 K steps. MX typically supports temperature controllers by treating them as if they were motor controllers, so for this case you would set up an MX motor scan with a start position of 290 K, a step size of 1 K, and with 21 measurement steps.

MX motor scans can also be *multidimensional*. Suppose that you had a large sample attached to an X-Y translation stage, and that you wanted to scan this sample over an area that was 5 cm wide by 10 cm high. If the zeros of X and Y were set at the center of the sample, then you could run a two-dimensional MX scan with the X motor scanned from -2.5 cm to +2.5 cm and the Y motor scanned from -5 cm to +5cm with whatever step size you chose.

In a multidimensional scan, the motor listed first changes position least often while the motor listed last changes position most often. Using the X-Y scan above as an example, if the X step size was 0.5 cm, then the scan would start by moving the motors to X = -2.5 cm and Y = -5 cm. The first linear slice of the scan would be performed with X fixed at -2.5 cm, while Y would step scan from -5 to +5 cm. After the first slice was completed, MX would then move the motors to X = -2.0 cm and Y = -5 cm. Then the second slice of the scan would be performed with X fixed at -2.0 cm and Y step scanning from -5 cm to +5 cm. Subsequent slices would proceed in a similar fashion until the last slice was executed with X = +2.5 cm and Y step scanning from -5 cm to +5 cm.

2.1.2 Pseudomotor scans

MX defines the concept of a *pseudomotor* as a software object that behaves like a motor object, but which are not actual motors. Pseudomotors are described in more detail in the **Motors** chapter, so we will only discuss here their interaction with scans.

An ordinary motor scan can scan both pseudomotors and real motors. The only real difference between a pseudomotor scan and a normal motor scan is that pseudomotor scans make an extra effort to ensure that the pseudomotors are used in a sane manner. In general, this "extra effort" only makes a difference if the position of a pseudomotor depends on the position of more than one real motor.

Example: Scanning slit pseudomotors

Slit pseudomotors are a typical example of a pseudomotor that depends on more than one real motor. An X-ray slit mechanism is used to block the transmission of X-rays everywhere except between the positions of the two slit blades. Such slit blades are typically made of tungsten, since tungsten strongly attenuates X-rays. Essentially, the slit acts as a single point collimator.

FIXME: Show a picture of an X-ray slit here that matches the description below.

Figure *???* shows a pair of slit blades that define the opening for X-rays in one dimension. The metal slit blade on the left marked "L" prevents X-rays from passing through further to the left than the position marked x_{min}, while the slit blade on the right marked "R" prevents X-rays from passing through further to the right than the position marked x_{max}. In general, experimenters want to use an X-ray beam aperture that collimates the beam in both the x and y directions. This is done by adding another pair of slit blades in a horizontal orientation instead of the vertical orientation of the first two slit blades.

From a hardware point of view, the simplest way of implementing a slit mechanism is to provide two independently movable blades for the x direction and another set of two independently movable blades for the y direction. However, experimenters typically do not think in terms of the position of the edges of the individual blades. Instead, they generally think in terms of the width (or height) of the slit opening and the position of the center of the slit opening. Since this is not what the hardware provides, we use pseudomotors to provide the interface that the experimenters expect.

Normally, there will be two pseudomotors that are built out of the two real motors. For the x direction, this will be

$$x_{center} = x_{right} - x_{left}, \quad x_{width} = 0.5 * (x_{right} + x_{left})$$

Similarly for the y direction, the definitions will be

$$y_{center} = y_{top} - y_{bottom}, \quad y_{height} = 0.5 * (y_{top} + y_{bottom})$$

Suppose that we setup and run a motor scan that attempts to step scan the x_{center} pseudomotor while holding the value of x_{width} fixed. Unfortunately, motor scans can only handle this situation correctly if the motor controllers behave in a perfect fashion with infinite resolution. However, real motor controllers have neither of these features.

To make the example more concrete, suppose that we attempt to step scan the pseudomotor x_{center} from -50 mm to +50 mm with a step size of 10 mm. Furthermore, suppose that the x_{width} pseudomotor has a value of 5 mm before the start of the scan. This means that when the scan starts, MX will attempt to move the two real motors, x_{left} and x_{right}, to the starting positions of -52.5 mm and -47.5 mm respectively. As we continue the scan, the x_{left} and x_{right} pseudomotors ideally will go to the following positions:

x_left	x_right
-52.5	-47.5
-42.5	-37.5
-32.5	-27.5
-22.5	-17.5
-12.5	-7.5
-2.5	2.5
7.5	12.5
17.5	22.5
27.5	32.5
37.5	42.5
47.5	52.5

This is in the perfect environment.

In a real environment, stepping motors can lose steps during motion and DC servo motors can end their moves at positions slightly different than the requested position. Take a look at the move from the first step of the scan (-52.5, -47.5) to the second step of the scan (-42.5, -37.5). Suppose that x_{right} loses a few motor steps in its motion to the position $x_{right} = -37.5$ due to imperfections in the motor or its tuning, so that the motor actually ends up at the position $x_{right} = -37.75$. This means that the slit pseudomotors with the incorrect positions of $x_{center} = -40.125$ and $x_{width} = 4.75$. Errors of this sort are never desirable, but under certain circumstances one has to put up with them.

The ***real*** problem is what happens during the move from the second step scan position to the third step scan position. In general, pseudomotors such as the slit pseudomotors do not maintain an elaborate history of all the positions that they were ever at. Instead, all they know are the current positions of the real motors that they depend on.

So think about what would happen during the move from the second step position to the third step position. The MX motor scan knows that the next step is supposed to move the x_{center} pseudomotor in the positive direction by 10 mm. What the motor scan does ***not*** know is the current position of the x_{center} pseudomotor. Instead, it asks the *slit_center* pseudomotor driver for the current position of x_{center} and gets back the answer $-40.125mm$. The motor scan cheerfully believes this and then commands a move of x_{center} to $-30.125mm$.

The textitslit_center driver looks at the request to move the pseudomotor by $10mm$ and interprets that as a request to move the real motors each by $10mm$. Thus, x_{left} is commanded to move to $-32.5mm$ while x_{right} is commanded to move to $-27.75mm$. Remember that on this step, the x_{right} motor should have been commanded to move to $-27.5mm$, but it was actually commanded to move to $-27.75mm$. If the same kind of error occurs in the move from the second step to the third step, then x_{right} may actually end up at $-28.0mm$. If we propagate this error, through the rest of the scan, then after the last step x_{left} is at the correct position of $47.5mm$, but x_{right} is at the significantly incorrect position of $50.0mm$ instead of $52.5mm$.

Such a result is obviously unacceptable. If the scan started with $x_{width} = 5mm$, then the scan ended with $x_{width} = 2.5mm$. If the slit blades are collimating a uniform beam, then this error will result in a 50 percent reduction in beam intensity, which cannot be tolerated.

This kind of error was the inspiration for pseudomotor scans. Pseudomotor scans and ordinary motor scans behave the same for real motors. But for certain types of pseudomotors, the pseudomotor scan records the starting position for each real motor. Then for each subsequent step of the step scan the destinations for the real motors are computed by looking at the starting positions of each real pseudomotor and then adding the value $step\_size * number\_of\_steps$ to each of those positions. Thus, when the example step scan above attempts to move the real motors to the positions for the third step in the scan, it will attempt to move x_{right} to the *correct* position of $-27.5mm$ instead of the incorrect position of $-27.75mm$.

The changes necessary to support this special pseudomotor scan behavior are currently (June 2008) only implemented for the *network_motor*, *slit_motor*, and *translation_mtr* drivers. In general, this kind of modification is only useful for pseudomotors that have internal dependencies on the relative positions of the real motors that are not obvious from the reported position of the pseudomotor. For example, this feature is totally irrelevant for a pseudomotor such as the *energy_motor* driver. An *energy_motor* pseudomotor only depends on one real motor, so there is no internal state about the relative positions of real motors that has to be preserved. On the other hand, this feature is only implementable for pseudomotors whose positions depend in a simple way on the positions of the real motors. An example of a pseudomotor driver that would have this problem is the *table_motor* driver. A *table_motor* pseudomotor reports the x, y, z, roll, pitch, or yaw positions for a table such as a diffractometer table. In general, rotation matrices must be used to compute the pseudomotor positions from the real motors positions for a diffractometer table, which means that the *table_motor* driver is not a good candidate for support by pseudomotor scans.

2.1.3 Slit Scans

Slit scans were the ancestor of pseudomotor scans. A pseudomotor scan can do everything a slit scan could do and can also handle translation pseudomotors, as well as forwarding such requests to remote MX servers. Some future version of MX may automatically replace all existing slit scans with pseudomotor scans.

2.1.4 Input Scans

Input scans are 1-dimensional scans that do not use a motor as the independent variable. Instead, the independent variable is the measurement count. This gives the experimenter a convenient way to run a scan that monitors a set of MX input devices without requiring the use of a motor. An alternate way of getting similar functionality is to run a motor scan using a motor controlled by any one of the *soft_motor*, *disabled_motor*, or *elapsed_time* motor drivers.

2.1.5 Theta-2 Theta Scans

For this type of scan, the experimenter specifies the names of two motors. Although two motors are specified, this is really a 1-dimensional scan in which the second motor is always moved to a position that is twice as large as the position of the first motor. For example, if the user specified the names of two motors, *alpha* and *beta*, then the scan range woult be specified in terms of the positions of the motor named *alpha*. During this can, the motor *beta* would always be move to a position with a numerical value that was always twice that of the position of *alpha*. Obviously, for this kind of scan to do the right thing, the zeros of the *alpha* and *beta* motors must be set to appropriate values such that *alpha* and *beta* are lined up with each other when they are both at positions with a numerical value of 0.

An alternate way of getting similar functionality is to use the *theta_2theta* pseudomotor driver. This driver has the advantage that it can be used seamlessly as part of an ordinary motor scan. However, one advantage of the scan is that the user can specify the names of the two motors to use without having to change the MX motor database.

2.2 List Scans

If the scan you want to perform has motor positions that are not evenly spaced or measurement times that are not all the same, then you may want to use an MX *list scan*. An MX list scan does not compute the needed motor positions itself. Instead, it fetches the requested measurement time and motor positions from some external source. At present, the only supported external source is a user-specified file, but it would be easy to add support for reading the positions from somewhere else such as an in-memory array.

2.2.1 File List Scans

For a file list scan, the user is expected to provide the name of the timing source, the names of the scan motors, and the name of a file containing the requested timing information and the motor positions. List scans support all of the standard measurement types, namely, *preset time*, *preset count*, and *preset pulse period* (for pulse generators). The file should contain columns of numbers with the first column corresponding to the measurement interval and the rest of the columns corresponding to the different motors.

As an example, suppose we want to scan the motors *energy*, *x*, and *y* along some path and the measurement times at the different scan positions are not all the same. Then, we can specify the time and the positions using an external file such as the following:

```
0.5   8900  4.3  3.1
0.6   8908  4.7  3.2
1.4   8940  5.1  2.9
0.7   8932  5.8  2.4
```

The list scan, when run, would then perform the following measurements:

- A 0.5 second measurement at *energy* = 8900 eV, *x* = 4.3 mm, and *y* = 3.1 mm.
- A 0.6 second measurement at *energy* = 8908 eV, *x* = 4.7 mm, and *y* = 3.2 mm.
- A 1.4 second measurement at *energy* = 8940 eV, *x* = 5.1 mm, and *y* = 2.9 mm.
- A 0.7 second measurement at *energy* = 8932 eV, *x* = 5.8 mm, and *y* = 2.4 mm.

In general, it is better that motors not change direction during a scan because of backlash, but the list scan does not require this as you can see above.

2.3 Quick Scans

A disadvantage of linear and list scans is that between each step of the measurement, the motors have to accelerate up to speed and then decelerate back down to zero before taking the next data point. The acceleration and deceleration can actually end up taking a significant amount of time, so some effort has been put into finding a way around this. One solution is to acquire data without stopping the motors. For example, suppose you command a rotary stage to move at 1 degree/second and then tell your measurement equipment to acquire 0.5 second measurements. Then all of your measurements will be averaged over 0.5 degree regions. In general, it is best for the dead time between measurements to be as short as possible (ideally zero).

Quick scans are also known as fast scans or slew scans in other control systems. In retrospect, the name *slew scan* would have been more appropriate, since many "quick" scans actually run the motor at a very slow speed.

At present, there are two kinds of quick scans available:

- Multichannel scaler (MCS) quick scans - These scans use an MX multichannel scaler device to acquire measurements over equally spaced intervals.
- APS insertion device quick scans - This kind of scan is only useable at the Advanced Photon Source. It attempts to synchronize the motion of an X-ray monochromator with a continuous change to the X-ray energy of a wiggler or undulator device. In practice, the APS insertion device scans have been little used, since the minimum speed of the undulator motors is too high for many experiments.

For the rest of this discussion, we will focus on multichannel scaler (MCS) scans, since most quick scans are of that type.

A multichannel scaler is a device that contains multiple scaler channels and which takes a sequence of measurements without stopping. Ideally, the sequence timing is handled entirely by the multichannel scaler itself, so that one can be sure of getting deterministic timing for the measurement sequence. A commonly used multichannel scaler is the Struck SIS-3801 which contains 32 input scaler channels. It can be commanded to take a series of measurements using either an internal clock or a sequence of external triggers.

An MCS quick scan typically looks like this:

- Move the motor to the start position of the scan.
- Start the multichannel scaler.
- Command the motor to move to the end position of the scan.
- Wait for the measurement sequence to complete.
- Read out the measured data points from the multichannel scaler.

2.3.1 Recording Motor Positions

In general, the multichannel scaler is completely capable of handling the acquisition of periodic scaler measurements. However it is also necessary to correlate these measurements with the corresponding motor positions. The strategy we have chosen in MX is to convert motor position measurements into something that can be recorded by a multichannel scaler.

In MX, we have defined a device class we call a "multichannel encoder (MCE)" whose job is to record the incremental position change of the motor for each measurement by the multichannel scaler. If you move your motors to the start position, start the multichannel scaler, and then command the motors to move, you then have a complete record of the motor position by using the incremental position changes as an offset to the original start position.

Typically, we construct multichannel encoders by arranging for the motor controller to generate an output signal that depends on the motor position. One way is to use a quadrature encoder signal output, if available. Another way (for motor controllers that support slaving) is to slave a second axis to the moving axis and then programming the slave axis to generate step and direction output signals. Either way, the next step is to add a bit of electronics that converts the quadrature signal or the step and direction signal into a pair of pulse trains typically called "clockwise" and "counter-clockwise". As you can guess, clockwise pulses appear when the motor is moving in one direction, while counter-clockwise pulses appear when the motor is moving in the other direction. These clockwise and counter-clockwise pulse trains are designed to be easily recordable in a pair of multichannel scaler channels. If you then start the multichannel scaler before you start the motor, you can compute the incremental position shift of the motor for each MCS measurement by simply taking the difference between the number of clockwise and counter-clockwise pulses for each measurement.

Some motor controllers, such as the Delta Tau PMAC, allow you to slave any motor axis to any other motor axis. For such controllers, if you dedicate one motor axis to generating the output for the multichannel encoder, you can then quick scan all of the other axes in the controller by arranging to make the MCE axis be a slave to the scan motor axis before starting the scan.

2.3.2 Quick Scanning Pseudomotors

... TBD ...

2.4 XAFS Scans

XAFS scans are a specialized kind of motor scan used to examine the behavior of the absorption of X-rays by a material near one of the absorption edges of the material. They are generally only of interest to experimenters doing X-ray absorption or X-ray fluorescence experiments.

An X-ray absorption edge is a fairly sharp discontinuity in the absorption of X-rays by the material. Absorption edges are a consequence of quantum mechanics and are due to the fact that electrons in an atom are arranged in shells. For a given shell of electrons, if an X-ray comes into the atom with slightly more energy than the absorption edge energy, then an electron can be ejected from the atom. If the X-ray has slightly less energy than the absorption edge energy, then no electron will be ejected. Each atomic element has its own characteristic absorption edge energies and one way to measure the atomic composition of a material is to look for the presence of absorption edges. The absorption spectrum shows some amount of oscillation just above the edge which is collectively called X-ray absorption fine structure (XAFS).

2.4.1 XAFS Scan Regions

In MX, XAFS scans consist of a series of regions that are contiguous with each other. If we define E_0 as the absorption edge energy, then regions that are below that energy, or which overlap that energy are specified in terms of the difference between the X-ray energy and the absorption edge energy. This quantity is typically called $E - E_0$ and is represented in the MX database by a pseudomotor called `e_minus_e0`. Energies above the absorption energies are typically specified in terms of the quantum mechanical wavenumber of the ejected electron. The wavenumber k is defined by the equation

$$E = E_0 + \frac{\hbar^2 k^2}{2m_e}$$

where E is the energy of the incoming photon, E_0 is the absorption edge energy, $\hbar$ is Planck's constant h divided by 2π, and m_e is the mass of the electron. The k value is represented in the MX database by a pseudomotor that is also called `k`.

A typical XAFS scan will contain a few scan regions before the absorption edge and a few after the edge. Here is an example scan:

- `e_minus_e0` region 1
  Scan from $E - E_0$ = -1500 eV to -100 eV in 200 eV steps with a 0.1 second measurement time.
- `e_minus_e0` region 2
  Scan from $E - E_0$ = -100 eV to -20 eV in 5 eV steps with a 2 second measurement time.
- `e_minus_e0` region 3
  Scan from $E - E_0$ = -20 eV to 15 eV in 0.3 eV steps with a 4 second measurement time.
- `e_minus_e0` region 4
  Scan from $E - E_0$ = 15 eV to 30 eV in 0.5 eV steps with a 6 second measurement time.
- `k` region 1
  Scan from $E - E_0$ = 30 eV to k = 8 Å^{-1} in 0.07 Å^{-1} steps with a 8 second measurement time.
- `k` region 2
  Scan from k = 8 Å^{-1} to k = 12 Å^{-1} in 0.07 Å^{-1} steps with a 10 second measurement time.

The choice of regions depends on the type of experiment being performed and is typically chosen to make most effective use of the (typically limited) available X-ray time.

2.4.2 XAFS Scan-related Database Records

XAFS scans expect the presence of a number of MX database records with specific names and types.

Required Records

- `edge_energy` - This is an MX database variable that is expected to contain the value of the X-ray absorption edge energy expressed in electron volts (eV).
- `e_minus_e0` - This is a pseudomotor record that is expected to control the offset between the current energy of the X-ray beam and the X-ray absorption edge energy. It is normally implemented using a *delta_motor* pseudomotor.
- `k` - This is a pseudomotor record of type *xafs_wavenumber* which expresses the wavenumber of the ejected electron in inverse Angstroms. The driver for this pseudomotor **assumes** the presence of the database variable `edge_energy` without actually mentioning the variable in its database record.

Here is an example database that demonstrates the use of these records using an *epics_motor* record for the θ axis.

```
edge_energy variable inline double "" "" 1 1 8980.0
d_spacing   variable inline double "" "" 1 1 3.13555
theta       device motor epics_motor "" "" 0 0 -100000 100000 0 -1 -1 0.001 0 deg wml:m1
energy      device motor energy_motor "" "" 0 0 0 1e+08 0 -1 -1 1 0 eV theta d_spacing 0.0
e_minus_e0  device motor delta_motor "" "" 0 0 -1e+08 1e+08 0 -1 -1 1 0 eV energy edge_ene
k           device motor xafs_wavenumber "" "" 0 0 0 1e+08 0 -1 -1 1 0 A-1 energy
```

The `theta`, `energy`, and `d_spacing` records are not required to have the names shown above, but those names are conventional.

Note: Some future version of MX may make the required names configurable. However, this is only likely to happen if some future beamline has an experiment that needs to simultaneously use multiple edge energies.

Required for *XAFS* Format Datafiles

If you save the data from your XAFS scan using MRCAT's *xafs* datafile format, then you need several more records in your MX database which are used to generate the *xafs* format header for the data files. Here is an example database of that:

```
beamline_name variable inline string "" "" 1 20 "APS Sector 10-ID"
ring_energy variable inline double "" "" 1 1 7
#
amplifier_list variable inline string "" "" 2 3 17 keithley1 keithley2 keithley3
keithley1      device amplifier soft_amplifier "" "" 10000000 0 0
keithley2      device amplifier soft_amplifier "" "" 10000000 0 0
keithley3      device amplifier soft_amplifier "" "" 10000000 0 0
#
xafs_header1   variable inline string "" "" 1 81 "Header line 1"
xafs_header2   variable inline string "" "" 1 81 "Header line 2"
xafs_header3   variable inline string "" "" 1 81 "Header line 3"
```

The XAFS header file variables are described in the datafile format section of this document that follows.

2.5 Scan Datafile Formats

2.5.1 *Text* Format

2.5.2 *SFF* Format

2.5.3 *XAFS* Format

2.6 Area Detector Scans

2.6.1 Wedge Scans

If you are acquiring a data set of area detector images using step scans, the starting and stopping of motor moves between measurements can introduce a significant time overhead. This can lead to a significant increase in radiation dose to your protein crsytal.

More to come ...

2.7 Area Detector Sequences

Area detector sequences are not really MX scans, but they resemble MX scans in many ways. The primary difference is that an MX area detector record is in control of the image acquisition process, instead of an MX scan record.

More to come ...

Chapter 3

Building and Installing MX

3.1 Prerequisites

The first step in building MX is to set up the environment in which to compile and install the source code. For Microsoft Windows, this is a somewhat complicated and long process. For most of the other MX build targets, the process is much, much simpler.

3.1.1 Linux

Due to the available package managers and distribution repositories, Linux is one of the easier systems on which to set up an MX build environment. For many Linux distributions, all of the prerequisite packages will already be available from your package repositories.

The most commonly used Linux distributions for MX are:

- **Debian** - Debian GNU/Linux is the primary development platform for MX on Linux.

- **RHEL, CentOS, Scientific Linux** - There are a number of sites that run MX on this family of distributions. If you want to use any of the MX GUI programs, then you should plan on adding the "Extra Packages for Enterprise Linux (EPEL)" repository to your list of repositories. RHEL, for example, definitely does not contain all of the packages needed to build the wxPython GUIs in the base system.

Linux Installation

We will use the Debian 7.0 (amd64) distribution as our example in the following description. For the most parts, the details for other distributions should be similar.

MX is most commonly used on the "amd64", "x86", and "powerpc" Debian ports. The "armel" and "armhf" ports are also likely to be used in the future. A small amount of testing has also been done for various versions of the "alpha", "ia64", "m68k", "mips", "mipsel", "s390", "sh", and "sparc" ports, but this has typically been to just verify that the code compiles, with brief testing of the binaries.

The first step is to install Debian (or another Linux distribution) on your computer. There are many online references and books that describe this process, so I will not go into it here. Early on in the installation process, Debian will display a “Software selection” screen. At this screen, I will generally select at least the “SSH server” task, since that makes remote administration much easier. For servers, I often install little more than this, while for clients you can install whatever GUI environments that they are most comfortable with.

However, in the following description, I assume that you only installed the base Debian distribution, with only the OpenSSH programs added. This makes it easier to enumerate the list of packages that are actually required for various MX components.

Packages Needed to Compile MX (C/C++ support)

The source code to the core of MX is generally distributed either as a compressed tar file from the MX web site *http://mx.iit.edu* with a name like *mx-2.0.0.tar.gz* or by downloading the source code using Subversion from the MX Subversion repository like this

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

The source code in the *mx/modules* subdirectory is for optionally compiling dynamically loadable modules that make use of external software that may not always be present such as EPICS. However, the rest of the MX core is intended to be built on all MX installations. Compiling modules is described later in this manual.

In order to build the MX core, you need at least a compiler and GNU make. On Wheezy (Debian 7), the following set of packages should be sufficient for that

- **gcc** - The GNU C compiler. Selecting this package should automatically result in the installation of a number of other needed packages like **binutils**, **cpp**, **libc6-dev**, and so forth, but the dependency handling of the Debian *apt-get* or *aptitude* commands should take care of automatically selecting all of the other needed packages for you.
- **make** - The GNU make program.

Other things that are convenient to have, but not absolutely required, include

- **gdb** - The GNU debugger.
- **subversion** - Used to download MX source code from the Subversion repositories.

Installation of the above packages and their dependencies, should be enough to build and install a completely functional version of the MX core.

Packages Needed to Compile Mp (Python support)

Mp provides the Python API for MX. At present, Python 3.x is not supported and will not be until after MX 2.0.0 is released.

The following packages are required to build Mp.

- **python** - Python 2.7 is best for now.
- **python-dev** - Provides the C API for Python extension modules such as Mp.
- **python-numpy** - It is possible to build Mp without Numpy support. However, array and image operations will run much, much slower, so you are strongly encouraged to install **python-numpy**.

Packages Needed to Support WxPython-based GUIs

The following package is needed for WxPython support

- **python-wxgtk2.8** - Installing this package should pull in all of the other packages needed to support basic Mp GUI functionality.
- **python-imaging** - Python Imaging Library (PIL) is required by Mp GUIs that display two dimensional area detector images such as *mpad*.

Other useful (but not required) packages include

- **python-matplotlib** - An excellent two-dimensional graphing package.

3.1.2 Mac OS X

In many ways, Mac OS X is much easier to set up for MX than Linux. If first you install Xcode, which is Apple's software development toolkit, almost everything that you need to develop MX software will already be installed. You will already have a C/C++/Objective C compiler, Python, and WxPython. The only item missing is the Python Imaging Library (PIL). That is needed for Mp area detector image display GUIs, so you will need to download PIL separately and install it.

Warning: I have not yet tested with the new LLVM-based compiler called *Clang*. It will probably require its own separate build target for it to work, so for now stick with GCC.

3.1.3 Microsoft Windows

MX can be built for almost all 32-bit and 64-bit x86 versions of Windows starting with Windows 95 and Windows NT 3.51. Versions of MX running on older versions of Windows may be missing some features (especially for the Windows 9x series), but most core functionality still works. At present (2013), the most commonly used version for MX is 64-bit Windows 7 followed by 32-bit Windows XP.

MX can use a variety of different compilers on Microsoft Windows. Here they are, listed in order from most used to least used.

- **Microsoft Visual C++** - The following versions have been tested with MX: Visual C++ 4.0, Visual C++ 5.0, Visual C++ 6.0, Visual C++ 2003, Visual C++ 2005 Express, Visual C++ 2008, Visual C++ 2010 (currently most used), and Visual C++ 2012. Visual C++ 2013 has not yet been tested.
- **Cygwin GCC**
- **MinGW GCC** - Only tested with MSYS.
- **Borland C++** - Only version 5.5 has really been tested.
- **DJGPP** - This does not run on 64-bit versions of Windows.

The vast majority of build targets in MX support Unix or Posix-like programming environments either in a self hosted or a cross-compiled system. However, Microsoft Windows does not. Rather than have an entirely separate build system for Microsoft Windows, I have chosen to require the installation of a copy of GNU make on Windows and its associated build environment in order to build MX on Windows, even when using the Microsoft or Borland compilers.

Here is a list of the availables ways of providing such an environment are listed in order from most used to least used.

- **Cygwin** - The Cygwin environment is my preferred environment for building MX on Windows, both for Visual C++ and GCC.
- **MSYS** - This is a fork of an old version of Cygwin that is typically used with the MinGW version of GCC.
- **UnxUtils** - Not tested recently. It probably does not run on modern versions of Windows.

I do already know about Microsoft's *nmake* program. However, the MX makefiles make heavy use of GNU make style "include" directives, Unix-style shell programs like *cp, rm, mv* as well as other GNU make features, so supporting *nmake* would be difficult. In case you are thinking of suggesting it, I am totally uninterested in adding an entirely separate build system just for Microsoft Windows.

Given the above, setting up a build environment for Microsoft Windows is more complicated than for any other build target supported by MX. In the following sections, I describe how to do that.

Cygwin build environment with Visual C++ compiler

This is probably the most common environment for building MX on Windows. Setting up the prerequisites is a multistep process. Many of the items on this list are only suggestions and not required.

1. **Install Cygwin from *www.cygwin.com*.**
   The basic steps for installing Cygwin are explained quite well on the Cygwin web site. A few things to look out for:

   - Recent releases of Cygwin provide both a 32-bit version and a 64-bit version. Make sure that you are installing the Cygwin version that matches the 32/64-bit size of the Windows version you have installed.
   - The Windows 9x series is not supported by the modern version of Cygwin 1.7.x. But if you can find a copy of the old Cygwin 1.5.x, then it can be used to build MX.

   Along with the Cygwin Base packages, I normally install the following optional packages on Cygwin: **bash**, **binutils**, **bzip2**, **cpio**, **curl**, **cygrunsrv**, **cygutils**, **dash**, **diffutils**, **dos2unix**, **enscript**, **file**, **gcc4-core**, **gcc4-g++**, **gccmakedep**, **gdb**, **groff**, **less**, **links**, **make**, **makedepend**, **man**, **mintty**, **nc6**, **ncftp**, **ncurses**, **netpbm**, **openssh**, **p7zip**, **perl**, **ping**, **procps**, **psmisc**, **python**, **python-numpy**, **rebase**, **screen**, **shutdown**, **subversion**, **sunrpc**, **tcl**, **tcl-tk**, **texinfo**, **unzip**, **util-linux**, **wget**, **which**, and **zip**. Not everything in the preceding list is strictly required, but some dependencies can be non-obvious. Note that the above list also gives you enough packages to compile the Cygwin version of MX, but we are focusing on the Microsoft Visual Studio version for now.

2. **Create Mintty Shortcut on Desktop** (optional)

   - Create a desktop shortcut to *C:\cygwin\bin\mintty.exe*.
   - Edit the properties of the shortcut as follows:

     Target: C:\cygwin\bin\mintty -t mintty -e /usr/bin/bash –login -i
     Start In: %USERPROFILE% environment variable

3. **Setup /home Mount Point for Cygwin** (optional)
   This change makes it so that user files for both Windows and Cygwin are saved in the same actual directory. The following commands should be executed from the Cygwin bash shell.

   - Move the /home directory that Cygwin created for you from /home to /home.old.

- mkdir /home
- Add a line to /etc/fstab to mount the Windows equivalent of a home directory on the Cygwin /home.

  For Vista and above: `C:/Users /home ntfs binary 0 0`

  For XP and before: `C:/Documents\040and\040Settings /home ntfs binary 0 0`
- Run the *"mount -a"* command to mount the Windows user directory at the location Cygwin expects.

4. **/etc/passwd** (optional for SSHD)
   Check /etc/passwd to see if it is set up correctly. This is only needed if you intend to run an OpenSSH server. Users not listed in /etc/passwd cannot login via the Cygwin OpenSSH server.

5. **/usr/bin/ssh-host-config** (optional for SSHD)
   This is only necessary if you intend to run an OpenSSH server.

   - Start Mintty from its desktop shortcut by right clicking on the Mintty icon and selecting "Run as Administrator".
   - Run the */usr/bin/ssh-host-config* script and give it the following answers. If an answer is not specified in the following list, then you should use the default for that and press the Enter key.

     Use privilege separation.
     Create new local account "sshd".
     Install sshd as a service.
     Leave the CYGWIN environment set to empty (or maybe set it to 'nodosfilewarning').
     Use the name "cyg_server".
     Create "cyg_server" account.
     Give "cyg_server" a password.

     At this point the script should end.
   - Start the Cygwin OpenSSH server with this command: "cygrunsrv -S sshd".

6. **Allow inbound SSH access** (optional for SSHD)
   The following procedure is for Windows 7. Other versions of Windows will be somewhat different.

   - Start the Windows Control Panel.
   - Select "Control Panel > All Control Panel Items > Windows Firewall > Allowed Programs
   - Click "Change Settings".
   - Browse to "C:\cygwin\usr \sbin\sshd.exe"
   - Allow "Home/Work" and "Public".

7. **Install TightVNC** (optional)
   Used for remote adminstration. TightVNC is my preferred VNC server, but you can also use other VNC servers, Remote Desktop, LogMeIn, or whatever. Here are some recommendations for TightVNC on how to ensure that VNC traffic does not flow over the Internet unencrypted.

   - Select Typical options for TightVNC, but **DO NOT** add a TightVNC exception to the Windows firewall.
   - In "TightVNC Service Configuration > Access Control", select "Allow loopback connections".

- When you want to connect remotely to VNC, create an SSH tunnel hosted on the TightVNC server computer that connects port 5900 on localhost (the TightVNC server computer's localhost) to port 5901 on **your** local computer.
- Make a VNC connection to :1 on your localhost (which is really port 5901). If you do this, your VNC traffic will go through the SSH tunnel rather than unencrypted over the Internet.

8. **Install Virtual Clone Drive**
   Virtual Clone Drive is freeware that comes from *www.slysoft.com*. This software provides a way of mounting ISO images as if they were CD/DVD drives. This will be used for the Visual C++ installation that follows.

   If you have one, you may use a different software package to mount ISO images. Alternately, if you have a **real** install DVD, you can just use that in your CD/DVD drive and skip installing something like Virtual Clone Drive.

9. **Install Visual C++**
   Currently (2013), I normally use Microsoft Visual Studio 2010, but many other versions of Visual C++ (either Express, non-Express, or Windows SDK) can be used to build MX. Regardless of which version you use, you will only need the C/C++ compiler component and not the other components (Visual Basic, SQL, etc.). MX does not make use of Visual Studio project files, so I normally only use the Visual Studio IDE for its debugger.

   If you are using an Express version of Visual C++, then this only provides a 32-bit version of the compiler. This is fine on a 32-bit version of Windows. For 64-bit versions of Windows, you *may* be able to get a 64-bit compiler by downloading and installing the Windows SDK. You can also get a debugger by downloading and installing the "Debugging Tools for Windows".

10. **Run Microsoft Update** (optional)
    It is strongly recommended that you install all of the updates for the version of Visual C++ that you installed. This can be best handled by upgrading "Windows Update" to "Microsoft Update" and then telling "Microsoft Update" to update "Other Microsoft Programs".

Chapter 4

The *Mxmotor* Program

Chapter 5

Building and Installing MX Modules

MX normally tries to compile the MX drivers for every device supported by MX for your build target. Some drivers only work on some MX build targets, like the *tty* and *win32_com* RS-232 drivers. But the MX build system attempts to handle this kind of driver automatically, so you should never have to do anything special to get them compiled.

However, there are other drivers that depend on the presence of external software packages such as the EPICS control system, the XIA Handel multichannel analyzer system, and the DALSA Sapera LT video camera/frame grabber software. For this type of external software package, MX uses dynamically loadable "MX modules" that bundle one or more drivers into one single binary object.

Note: If you used older versions of MX like MX 1.5.4 and before, you may be used to editing the *mx/libMx/mxconfig.h* file and the *mx/libMx/Makehead.???* that matches your platform. In MX 2.0 and above, the file *mx/libMx/mxconfig.h* no longer exists and you no longer edit *mx/libMx/Makehead.???* for this purpose.

... More to come ...