

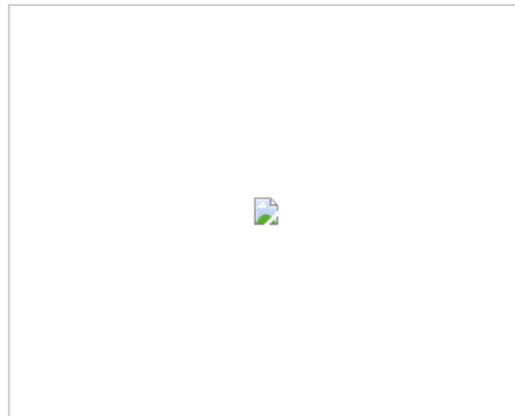
Writing a Motor Controller Driver

Kevin M. Peterson

2015-02-17

Software used during this presentation

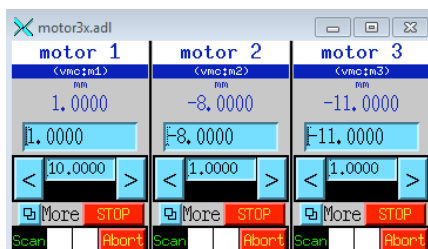
- Prebuilt IOC (Windows, OS X, Linux)
- Virtual Motor Controller (requires Python 2.7)
 - Provides 8 axes (400 steps per EGU)



- Software setup instructions:
 - <http://www.xray.aps.anl.gov/~kpetersn/motorClass.html>

Prerequisites

■ Familiarity with the motor record



■ Some programming experience

ssh

Installing created dbd file ../../dbd/iocvmcLinux.dbd

```
/usr/bin/g++ -c -D_POSIX_C_SOURCE=199506L -D_POSIX_THREADS -D_XOPEN_SOURCE=500 -D_X86_64 -DUNIX -D_BSD_SOURCE -Dlinux -D_REENTRANT -O3 -Wall -m64 -MM -I. -I../0.Common -I. -I.. -I.././././include/os/Linux -I.././././include -I/APSShare/epics/base-3.14.12.3/include/os/Linux -I/APSShare/epics/base-3.14.12.3/include -I/APSShare/epics/synApps_5_7/support/async-4-21/include -I/APSShare/epics/synApps_5_7/support/autosave-5-5/include/os/Linux -I/APSShare/epics/synApps_5_7/support/autosave-5-5/include -I/APSShare/epics/synApps_5_7/support/busy-1-6/include -I/APSShare/epics/synApps_5_7/support/calc-3-2/include -I/APSShare/epics/synApps_5_7/support/seq-2-1-13/include -I/APSShare/epics/synApps_5_7/support/sscan-2-9/include -I/APSShare/epics/synApps_5_7/support/std-3-2/include ..../VirtualMotorDriver.cpp
../VirtualMotorDriver.cpp: In member function 'virtual asyncStatus VirtualMotorAxis::poll(bool*)':
../VirtualMotorDriver.cpp:428: error: expected ';' before 'comStatus'
../VirtualMotorDriver.cpp:435: error: expected ';' before 'direction'
../VirtualMotorDriver.cpp:447: error: expected ';' before 'limit'
../VirtualMotorDriver.cpp:457: error: expected ';' before 'skip'
../VirtualMotorDriver.cpp:421: error: label 'skip' used but not defined
make[3]: *** [VirtualMotorDriver.o] Error 1
make[3]: Leaving directory `/tmp/vmc/vmcApp/src/0.linux-x86_64'
make[2]: *** [install.linux-x86_64] Error 2
make[2]: Leaving directory `/tmp/vmc/vmcApp/src'
make[1]: *** [src.install] Error 2
make[1]: Leaving directory `/tmp/vmc/vmcApp'
make: *** [vmcApp.install] Error 2
[bdadp14 /tmp/vmc]$
```

■ Knowledge of controller commands

MM4005

Command List — Alphabetical

Command	Description	MM4005	Command	Description	MM4005
AB	Abort motion	■	BQ	Generate service request (SRQ)	■
AC	Set acceleration	■	BS	Reset controller	■
AD	Define the maximum allowed angle of discontinuity	■	IS	Set I/O output bits	■
AM	Set analog input mode	■	SC	Set control loop type	■
AP	Abort program	■	SD	Speed scaling	■
AQ	Axis position acquisition	■	SE	Start synchronized motion	■
AS	Axis string	■	SI	Set axis mechanical motion device	■
AT	Tell the element number under execution	■	SH	Set home preset position	■
AX	Assign a physical axis as X geometric axis	■	SL	Set left travel limit	■
AY	Assign a physical axis as Y geometric axis	■	SM	Save program	■
BA	Set backlash compensation	■	SN	Set axis displacement units	■
CA	Define sweep angle and build an arc of circle r (C/Cr, CA)	■	SO	Set I/O output byte	■
CB	Clear I/O output bits	■	SP	Set trace sample rate	■
CD	Set cycle value and activate periodic display mode	■	SQ	Set global sample rate	■
CM	Change communication mode	■	SR	Set right travel limit	■
CP	Complete program	■	SS	Set master-slave mode	■
CR	Define radius for arcs of circle r (C/Cr, CA)	■	ST	Stop motion	■
CS	Concatenate two strings	■	SY	Axis synchronization	■
CT	Define X position to reach and build an arc of circle r (C/Cr, CT)	■	TA	Read motion device	■
CY	Define Y position to reach and build an arc of circle r (C/Cr, CY)	■	TE	Read error message	■
DA	Read desired acceleration	■	TC	Read control loop type	■
DB	Read following error	■	TD	Read error last of program	■
DI	Define home	■	TE	Read error code	■
DL	Define label	■	TF	Read filter parameters	■
DM	Read manual velocity	■	TG	Toggle I/O output bits	■
DO	Read home search velocity	■	TH	Read historical position	■
DP	Read desired position	■	TL	Read left travel limit	■
DS	Read desired velocity	■	TM	Set trace mode	■
DT	Display program error	■	TN	Read displacement units	■
DU	Display a variable	■	TP	Read actual position	■
EV	Automatically execute on power on	■	TR	Read global trace data	■
EL	Erase the last element of trajectory	■	TS	Read right travel limit	■
ED	Execute a program	■	TX	Read encoder resolution	■
EN	Define the tangent angle for the first point	■	TY	Read controller activity	■
EP	Label function key	■	TX	Read controller extended status	■
ER	Clear function key for the first point	■	TY	Read a variable	■
FD	Display function keys	■	UF	Update servo filter	■
FE	Set maximum master slave following error	■	UH	Wait for I/O high	■
FF	Set output frequency	■	UL	Wait for I/O low	■
GG	Set global trace mode	■	VA	Set velocity	■
GR	Set master-slave reduction ratio	■	VB	Set base velocity (Shopper motor only)	■
H	Jump to label	■	VE	Read controller version	■
IC	Abort command line	■	VN	Define the vector acceleration on trajectory (trajectory acceleration)	■
IP	Set integral gain	■	VV	Define the vector velocity on trajectory (trajectory velocity)	■
IS	Set proportional gain	■	W	Wait	■
IS	Set integration of integral factor	■	WE	End while loop	■
IP	in position loop PID controller	■	WF	Wait for function key	■
LI	List program	■	WG	While variable is greater	■
LE	Extended list of the trajectory	■	WL	Wait for a trajectory (curvilinear) length	■
LS	Define X position and build a line segment r (L/L, LS)	■	WK	Wait for key	■
LV	Define Y position and build a line segment r (L/L, LV)	■	WL	While variable is less	■
M	Set manual mode	■	WN	Wait for a element of trajectory	■
MP	Motor OFF	■	WP	Wait for position	■
MS	Set manual velocity	■	WS	Wait for motion stop	■
ML	Set local mode	■	W	Wait	■
MO	Motor ON	■	WT	While variable is different	■
MP	Download EEPROM to RAM	■	XA	Tell the current maximum allowed angle of discontinuity	■
MS	Read motor status	■	XB	Read backlash compensation	■
MT	Move to travel limit switch	■	XD	Read derivative gain factor	■
MX	Define X position for a line segment r (L/L, MX)	■	XE	Tell the last element	■
MY	Define Y position and build a line segment r (L/L, MY)	■	XF	Read maximum following error	■
N	Set trajectory element where the generation of pulses starts	■	XG	Read home preset position	■
NE	Set trajectory element where the generation of pulses ends	■	XI	Read integral gain factor	■
NI	Set step (curvilinear distance) between synchronization pulses	■	XL	Delete one line of program	■
NN	Set number of synchronization pulses to generate	■	XM	Read available memory	■
NP	Set decimal digits number of position display	■	XN	Read number of acquisitions	■
NQ	Other generation of pulses on interpolation	■	XP	Read proportional gain factor	■
NT	Start definition of a new trajectory	■	XQ	Read global sample rate	■
OE	Test I/O output	■	XR	Read trace sample rate	■
OL	Set home search low velocity	■	XT	Tell number of elements in the trajectory	■
OR	Search low velocity	■	Y	Tell the vector acceleration on trajectory (trajectory acceleration)	■
PA	Move to absolute position	■	YV	Tell the vector velocity on trajectory (trajectory velocity)	■
PE	Set start position of generation of pulses of synchronization	■	ZA	Erase program	■
PF	Set end position of generation of pulses of synchronization	■	ZB	Add to variable	■
PI	Set step of generation of pulses of synchronization	■	ZC	Negate variable	■
PP	Other generation of pulses on motion	■	ZD	Add variables	■
PT	Quit program mode	■	ZE	Divide variables	■
QW	Save parameters	■	ZF	F variable is equal	■
RA	Read analog input	■	ZG	Scale variable	■
RD	Read I/O input	■	ZH	F variable is greater	■
RE	Enable display refresh	■	ZI	Read key to variable	■
RO	Read I/O output	■	ZJ	Multiply variables	■
RP	Repeat command line	■	ZK	F variable is different	■
			ZL	Send a value to an user analog port	■
			ZM	Set theoretical position in variable	■
			ZN	Set current position in variable	■
			ZO	Read a value from an user analog port and affect variable	■
			ZO	Initialize variable	■
			ZP	Wait and read key	■
			ZQ	Read value from keyboard in a variable	■
			ZR	Wait and read key	■
			ZS	Copy variable	■
			ZT	Read Axis/General parameters configuration	■

EDB162E1040 - 05/99

II



Writing a model 3 driver:

The standard approach

- Obtain documentation for the new controller
- Find a similar controller that already has EPICS support
- Use the similar controller's driver as a starting point
- Implement the necessary `asynMotor{Controller,Axis}` methods



Writing a model 3 driver:

The standard approach

- Obtain documentation for the new controller
- Find a similar controller that already has EPICS support
- Use the similar controller's driver as a starting point
- Implement the necessary `asynMotor{Controller,Axis}` methods

How to draw an owl

1.



1. Draw some circles

2.

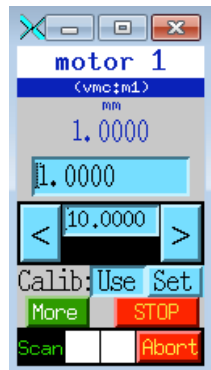


2. Draw the rest of the owl

References

- asynMotor documentation in the motor module
 - motor/documentation/motorDeviceDriver.html
 - Suggested example motor drivers:
 - ACS MCB-4B (simple driver, no additional asyn params)
 - Parker ACR (simple driver, adds few asyn params)
 - Newport XPS (complex driver, implements profile moves)
 - motor/documentation/motorDoxygenHTML/index.html
 - Install doxygen (if not already available)
 - `cd motor/documentation`
 - `make` (creates motorDoxygenHTML)
 - comments in asynMotor source code (if unable to build doxygen documentation)
 - `motor/motorApp/MotorSrc/asynMotor{Controller,Axis}.{cpp,h}`
- Motor record source code

Road Map



???



How does the IOC associate the model-3 motor driver with the motor record?

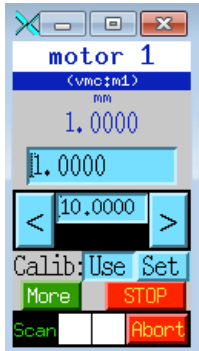
```
*vmc.cmd X
23 drvAsynIPPortConfigure("VMC_ETH","127.0.0.1:${VMC_PORT1}", 0, 0, 0)
24
25 # VirtualMotorController(
26 #   portName           The name of the asyn port that will be created for this driver
27 #   VirtualMotorPortName The name of the drvAsynSerialPort that was created previously
28 #   numAxes             The number of axes that this controller supports
29 #   movingPollPeriod    The time between polls when any axis is moving
30 #   idlePollPeriod      The time between polls when no axis is moving
31 VirtualMotorCreateController("VMC1", "VMC_ETH", 3, 250, 10000)
32
33 dbLoadTemplate("vmc.substitutions")
34
```

```
vmc.substitutions X
1 file "${TOP}/db/asyn_motor.db"
2 {
3 pattern
4 {P,      N,      M,      DTYP,      PORT,  ADDR,
5 {vmc:,  1,      "m$(N)",  "asynMotor",  VMC1,  0,
6 {vmc:,  2,      "m$(N)",  "asynMotor",  VMC1,  1,
7 {vmc:,  3,      "m$(N)",  "asynMotor",  VMC1,  2,
8 {vmc:,  4,      "m$(N)",  "asynMotor",  VMC1,  3,
9 {vmc:,  5,      "m$(N)",  "asynMotor",  VMC1,  4,
10 {vmc:,  6,      "m$(N)",  "asynMotor",  VMC1,  5,
11 {vmc:,  7,      "m$(N)",  "asynMotor",  VMC1,  6,
12 {vmc:,  8,      "m$(N)",  "asynMotor",  VMC1,  7,
13 }
```

```
motorSupport.dbd X
6 registrar(motorRegister)
7 registrar(asynMotorControllerRegister)
8 device(motor, INST_I0, devMotorAsyn, "asynMotor")
9
```

```
asyn_motor.db X
6 record(motor, "${P}${M}") {
7   field(DESC, "${DESC}")
8   field(DTYP, "${DTYP}")
9   field(DIR, "${DIR}")
10  field(VELO, "${VELO}")
11  field(VBAS, "${VBAS}")
12  field(ACCL, "${ACCL}")
13  field(BDST, "${BDST}")
14  field(BVEL, "${BVEL}")
15  field(BACC, "${BACC}")
16  field(OUT, "@asyn$(PORT),$(ADDR)")
17  field(MRES, "${MRES}")
18  field(PREC, "${PREC}")
19  field(EGU, "${EGU}")
20  field(DHLM, "${DHLM}")
21  field(DLLM, "${DLLM}")
22  field(INIT, "${INIT}")
23  field(RTRY, "${RTRY=10}")
24  field(TWV, "1")
25 }
```

Road Map



motor
record

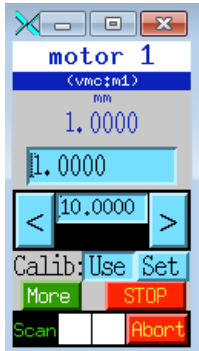
asynMotor
device support

???

controller-specific
asynMotor driver



Road Map



motor
record

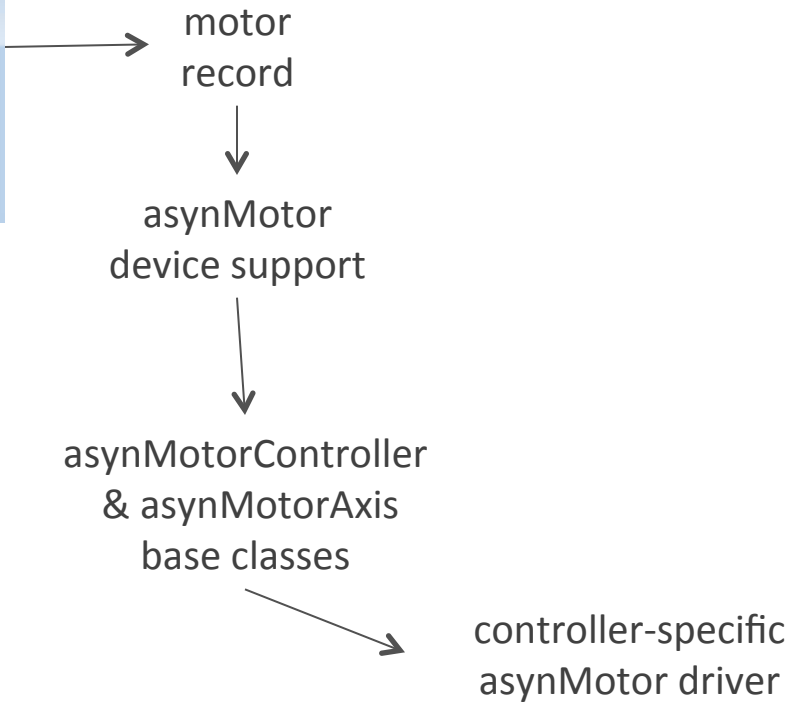
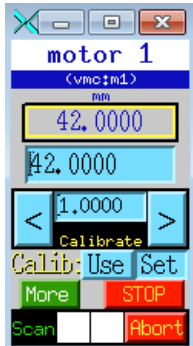
asynMotor
device support

asynMotorController
& asynMotorAxis
base classes

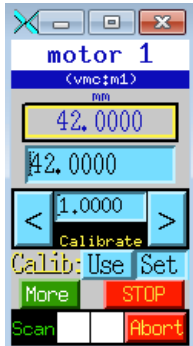
controller-specific
asynMotor driver



Road Map



Road Map



motor
record

↓
asynMotor
device support

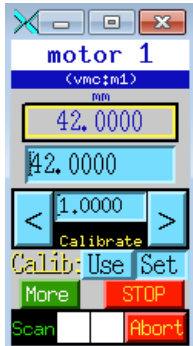
↓
asynMotorController
& asynMotorAxis
base classes

↘
controller-specific
asynMotor driver

```
/* Load pos. into motor controller.  
INIT_MSG();  
WRITE_MSG(LOAD_POS, &newpos);  
SEND_MSG();
```



Road Map



motor
record
↓
asynMotor
device support

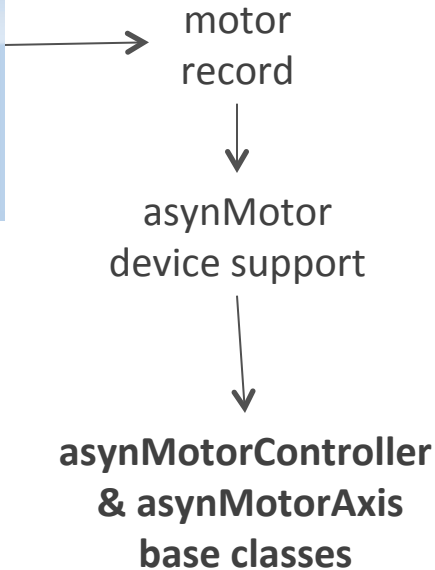
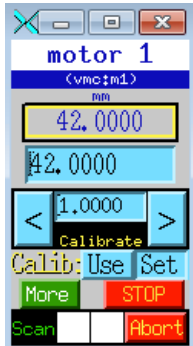
asynMotorController
& asynMotorAxis
base classes

controller-specific
asynMotor driver

```
switch (command) {  
case LOAD_POS:  
    pmsg->command = motorPosition;  
    pmsg->dvalue = *param;  
    pPvt->moveRequestPending++;  
    break;  
}
```



Road Map

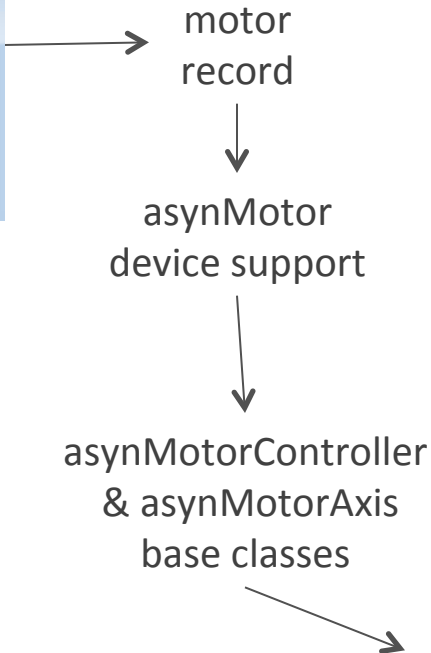
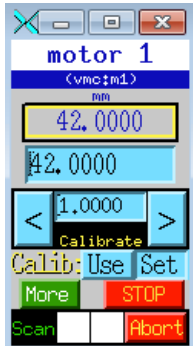


controller-specific
asynMotor driver

```
else if (function == motorPosition_) {  
    status = pAxis->setPosition(value);  
    pAxis->callParamCallbacks();  
}
```



Road Map



```
asynStatus VirtualMotorAxis::setPosition(double position)
{
    asynStatus status;

    sprintf(pC_>outString_, "%d POS %d", axisIndex_, NINT(position));
    status = pC_>writeReadController();
    return status;
}
```

**controller-specific
asynMotor driver**



Why are these details important?

- Understanding the source of the arguments passed to the driver makes writing simple model-3 drivers easier
 - Variables in motor drivers don't always have the most helpful names
- Understanding the role the `asynMotorController` base class makes it easier to write complex model-3 drivers
 - Adding new features requires adding parameters and implementing additional methods from the base classes
- It makes debugging less confusing
 - Knowing that motor commands get placed in asyn queues is crucial for following code execution from the motor record to the model-3 driver

Which `asynMotor{Controller,Axis}` methods need to be implemented?

- There are many methods that could be implemented

```
asynMotorController::asynMotorController(const char*, int, int, int, int, int, int, int, int)
asynMotorController::report(FILE*, int) : void
asynMotorController::writeInt32(asynUser*, epicsInt32) : asynStatus
asynMotorController::writeFloat64(asynUser*, epicsFloat64) : asynStatus
asynMotorController::writeFloat64Array(asynUser*, epicsFloat64*, size_t) : asynStatus
asynMotorController::readFloat64Array(asynUser*, epicsFloat64*, size_t, size_t*) : asynStatus
asynMotorController::readGenericPointer(asynUser*, void*) : asynStatus
asynMotorController::getAxis(asynUser*) : asynMotorAxis*
asynMotorController::setDeferredMoves(bool) : asynStatus
asynMotorController::getAxis(int) : asynMotorAxis*
asynMotorController::startPoller(double, double, int) : asynStatus
asynMotorController::wakeupPoller() : asynStatus
asynMotorController::poll() : asynStatus
asynMotorController::asynMotorPoller() : void
asynMotorController::startMoveToHomeThread() : asynStatus
asynMotorController::asynMotorMoveToHome() : void
asynMotorController::writeController() : asynStatus
asynMotorController::writeController(const char*, double) : asynStatus
asynMotorController::writeReadController() : asynStatus
asynMotorController::writeReadController(const char*, char*, size_t, size_t*, double) : asynStatus
asynMotorController::initializeProfile(size_t) : asynStatus
asynMotorController::buildProfile() : asynStatus
asynMotorController::executeProfile() : asynStatus
asynMotorController::abortProfile() : asynStatus
asynMotorController::readbackProfile() : asynStatus
asynMotorController::setMovingPollPeriod(double) : asynStatus
asynMotorController::setIdlePollPeriod(double) : asynStatus
setMovingPollPeriod(const char*, double) : asynStatus
setIdlePollPeriod(const char*, double) : asynStatus
asynMotorEnableMoveToHome(const char*, int, int) : asynStatus

asynMotorAxis::asynMotorAxis(class asynMotorController*, int)
asynMotorAxis::move(double, int, double, double, double) : asynStatus
asynMotorAxis::moveVelocity(double, double, double, double) : asynStatus
asynMotorAxis::home(double, double, double, int) : asynStatus
asynMotorAxis::stop(double) : asynStatus
asynMotorAxis::poll(bool*) : asynStatus
asynMotorAxis::setPosition(double) : asynStatus
asynMotorAxis::setEncoderPosition(double) : asynStatus
asynMotorAxis::setHighLimit(double) : asynStatus
asynMotorAxis::setLowLimit(double) : asynStatus
asynMotorAxis::setPGain(double) : asynStatus
asynMotorAxis::setIGain(double) : asynStatus
asynMotorAxis::setDGain(double) : asynStatus
asynMotorAxis::setClosedLoop(bool) : asynStatus
asynMotorAxis::setEncoderRatio(double) : asynStatus
asynMotorAxis::report(FILE*, int) : void
asynMotorAxis::doMoveToHome() : asynStatus
asynMotorAxis::setReferencingModeMove(int) : void
asynMotorAxis::getReferencingModeMove() : int
asynMotorAxis::setIntegerParam(int, int) : asynStatus
asynMotorAxis::setDoubleParam(int, double) : asynStatus
asynMotorAxis::callParamCallbacks() : asynStatus
asynMotorAxis::initializeProfile(size_t) : asynStatus
asynMotorAxis::defineProfile(double*, size_t) : asynStatus
asynMotorAxis::buildProfile() : asynStatus
asynMotorAxis::executeProfile() : asynStatus
asynMotorAxis::abortProfile() : asynStatus
asynMotorAxis::readbackProfile() : asynStatus
```

Which `asynMotor{Controller,Axis}` methods need to be implemented?

- But the motor record only uses a limited number of commands

```
motor.h X
92 typedef enum {
93     MOVE_ABS,      /* Absolute Move. */
94     MOVE_REL,      /* Relative Move. */
95     HOME_FOR,      /* Home Forward. */
96     HOME_REV,      /* Home Reverse. */
97     LOAD_POS,      /* Load Position. */
98     SET_VEL_BASE,  /* Set Minimum Velocity. */
99     SET_VELOCITY,  /* Set Jog and Trajectory Velocity. */
100    SET_ACCEL,      /* Set Acceleration. */
101    GO,             /* Start previously programmed move. */
102    SET_ENC_RATIO,  /* Set Encoder Ratio. */
103    GET_INFO,       /* Update Motor Status. */
104    STOP_AXIS,      /* Stop Axis Motion. */
105    JOG,            /* Momentary Jog. */
106    SET_PGAIN,      /* Set Proportional Gain. */
107    SET_IGAIN,      /* Set Integral Gain. */
108    SET_DGAIN,      /* Set Derivative Gain. */
109    ENABLE_TORQUE,   /* Enable Servo Closed-Loop Control. */
110    DISABL_TORQUE,  /* Disable Servo Closed-Loop Control. */
111    PRIMITIVE,      /* Primitive Controller command. */
112    SET_HIGH_LIMIT, /* Set High Travel Limit. */
113    SET_LOW_LIMIT,  /* Set Low Travel Limit. */
114    JOG_VELOCITY,   /* Change Jog velocity. */
115    SET_RESOLUTION  /* Set resolution */
116 } motor_cmd;
```

Which `asynMotor{Controller,Axis}` methods need to be implemented?

- Limiting the driver to motor-record commands shrinks the list

```
asynMotorController::asynMotorController  
asynMotorController::report  
asynMotorController::getAxis
```

```
asynMotorAxis::asynMotorAxis  
asynMotorAxis::move  
asynMotorAxis::moveVelocity  
asynMotorAxis::home  
asynMotorAxis::stop  
asynMotorAxis::poll  
asynMotorAxis::setPosition  
asynMotorAxis::setHighLimit  
asynMotorAxis::setLowLimit  
asynMotorAxis::setPGain  
asynMotorAxis::setIGain  
asynMotorAxis::setDGain  
asynMotorAxis::setClosedLoop  
asynMotorAxis::setEncoderRatio  
asynMotorAxis::report
```

Which `asynMotor{Controller,Axis}` methods need to be implemented?

- The Virtual Motor Controller driver implements these methods

```
asynMotorController::asynMotorController  
asynMotorController::report  
asynMotorController::getAxis
```

```
asynMotorAxis::asynMotorAxis  
asynMotorAxis::move  
asynMotorAxis::moveVelocity  
asynMotorAxis::home  
asynMotorAxis::stop  
asynMotorAxis::poll  
asynMotorAxis::setPosition  
asynMotorAxis::setHighLimit  
asynMotorAxis::setLowLimit  
asynMotorAxis::setPGain  
asynMotorAxis::setIGain  
asynMotorAxis::setDGain  
asynMotorAxis::setClosedLoop  
asynMotorAxis::setEncoderRatio  
asynMotorAxis::report
```

Which `asynMotor{Controller,Axis}` methods need to be implemented?

- A driver will be usable* if only these methods are implemented

`asynMotorController::asynMotorController`
`asynMotorController::report`
`asynMotorController::getAxis`

`asynMotorAxis::asynMotorAxis`
`asynMotorAxis::move`
`asynMotorAxis::moveVelocity`
`asynMotorAxis::home`
`asynMotorAxis::stop`
`asynMotorAxis::poll`
`asynMotorAxis::setPosition*`
`asynMotorAxis::setHighLimit`
`asynMotorAxis::setLowLimit`
`asynMotorAxis::setPGain`
`asynMotorAxis::setIGain`
`asynMotorAxis::setDGain`
`asynMotorAxis::setClosedLoop`
`asynMotorAxis::setEncoderRatio`
`asynMotorAxis::report`

* `motorAxis::setPosition` is needed for autosave to restore a position to the controller

Virtual Motor Controller Commands

- The commands can be found in the vmc documentation directory
 - `vmc/documentation/VirtualMotorControllerCommands.txt`

Seeing controller communication

```
cal C:\Windows\system32\cmd.exe
2 POS?\\r
2015/02/16 23:28:12.874 127.0.0.1:31337 read 7
-4400\\r\\n
2015/02/16 23:28:12.876 127.0.0.1:31337 write 6
2 ST?\\r
2015/02/16 23:28:12.877 127.0.0.1:31337 read 3
3\\r\\n
2015/02/16 23:28:12.878 127.0.0.1:31337 write 7
3 POS?\\r
2015/02/16 23:28:12.880 127.0.0.1:31337 read 7
-4400\\r\\n
2015/02/16 23:28:12.881 127.0.0.1:31337 write 6
3 ST?\\r
2015/02/16 23:28:12.883 127.0.0.1:31337 read 3
3\\r\\n
2015/02/16 23:28:22.898 127.0.0.1:31337 write 7
1 POS?\\r
2015/02/16 23:28:22.900 127.0.0.1:31337 read 7
11296\\r\\n
2015/02/16 23:28:22.901 127.0.0.1:31337 write 6
1 ST?\\r
2015/02/16 23:28:22.902 127.0.0.1:31337 read 3
3\\r\\n
2015/02/16 23:28:22.904 127.0.0.1:31337 write 7
2 POS?\\r
2015/02/16 23:28:22.905 127.0.0.1:31337 read 7
-4400\\r\\n
2015/02/16 23:28:22.907 127.0.0.1:31337 write 6
2 ST?\\r
2015/02/16 23:28:22.908 127.0.0.1:31337 read 3
3\\r\\n
2015/02/16 23:28:22.909 127.0.0.1:31337 write 7
3 POS?\\r
2015/02/16 23:28:22.911 127.0.0.1:31337 read 7
-4400\\r\\n
2015/02/16 23:28:22.912 127.0.0.1:31337 write 6
3 ST?\\r
2015/02/16 23:28:22.914 127.0.0.1:31337 read 3
3\\r\\n
```

asynRecord.adl

vmc:asyn1

Port: Address:

Connected

drvInfo: Reason:

Interface:

Error:

Connected Enabled autoConnect

traceMask	traceIOMask
0x8	0x2
Off On traceError	Off On traceIOASCII
Off On traceIODevice	Off On traceIOEscape
Off On traceIOFilter	Off On traceIOHex
Off On traceIODriver	80 Truncate size
Off On traceFlow	
Trace file: Unknown	

Implementing VirtualMotorController methods: constructor, report, getAxis

- constructor
 - Connects to the controller
 - Initializes the controller
 - Querying version strings usually occurs here
 - **Creates the VirtualMotorAxis objects**
 - Starts the poller
- report
 - Prints the values that were passed to VirtualMotorCreateController
 - Calls the report method of the base class (asynMotorController)
- getAxis (both forms)
 - Return VirtualMotorAxis pointers

Implementing VirtualMotorAxis methods: constructor, report

- constructor
 - Initializes the axis
 - May involve querying axis settings
 - Sets internal variables
 - 1-based index instead of zero
 - **Sets asyn parameters**
 - **Some MSTA bits need to be set here (GAIN_SUPPORT, ENCODER_PRESENT)**
 - Calls callParamCallbacks() to make changed parameters take effect
- report
 - Prints the values that were passed to VirtualMotorAxis constructor
 - 1-based index value for each axis
 - Calls the report method of the base class (asynMotorAxis)

Implementing VirtualMotorAxis methods: stop

```
/*
 * stop() is called by asynMotor device support whenever a user presses the stop button.
 * It is also called when the jog button is released.
 *
 * Arguments in terms of motor record fields:
 *   acceleration = ???
 */
asynStatus VirtualMotorAxis::stop(double acceleration)
{
    asynStatus status;

    sprintf(pC_->outString_, "%d AB", axisIndex_);
    status = pC_->writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: stop

```
/*
 * stop() is called by asynMotor device support whenever a user presses the stop button.
 * It is also called when the jog button is released.
 *
 * Arguments in terms of motor record fields:
 *   acceleration = ???
 */
asynStatus VirtualMotorAxis::stop(double acceleration)
{
    asynStatus status;

    sprintf(pC_->outString_, "%d AB", axisIndex_);
    status = pC_->writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: stop

```
/*  
 * stop() is called by asynMotor device support whenever a user presses the stop button.  
 * It is also called when the jog button is released.  
 *  
 * Arguments in terms of motor record fields:  
 *   acceleration = ???  
 */  
asynStatus VirtualMotorAxis::stop(double acceleration)  
{  
    asynStatus status;  
  
    sprintf(pC_->outString_, "%d AB", axisIndex_);  
    status = pC_->writeReadController();  
    return status;  
}
```

Implementing VirtualMotorAxis methods:

poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods: poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods:

poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods:

poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods:

poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods: poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods: poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods:

poll

```
asynStatus VirtualMotorAxis::poll(bool *moving)
{
    // Read the current motor position
    sprintf(pC_->outString_, "%d POS?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "0.00000"
    position = atof((const char *) &pC_->inString_);
    setDoubleParam(pC_->motorPosition_, position);

    // Read the status of this motor
    sprintf(pC_->outString_, "%d ST?", axisIndex_);
    comStatus = pC_->writeReadController();

    // The response string is of the form "1"
    status = atoi((const char *) &pC_->inString_);
```

```
// Read the direction
direction = (status & 0x1) ? 1 : 0;
setIntegerParam(pC_->motorStatusDirection_, direction);

// Read the moving status
done = (status & 0x2) ? 1 : 0;
setIntegerParam(pC_->motorStatusDone_, done);
setIntegerParam(pC_->motorStatusMoving_, !done);
*moving = done ? false:true;

// Read the limit status
limit = (status & 0x8) ? 1 : 0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (status & 0x10) ? 1 : 0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);

callParamCallbacks();
return comStatus ? asynError : asynSuccess;
}
```

Implementing VirtualMotorAxis methods: move

```
/* Arguments in terms of motor record fields:
 *   position (steps) = RVAL = DVAL / MRES
 *   baseVelocity (steps/s) = VBAS / abs(MRES)
 *   velocity (step/s) = VELO / abs(MRES)
 *   acceleration (step/s/s) = (velocity - baseVelocity) / ACCL */
asynStatus VirtualMotorAxis::move(double position, int relative, double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;
    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    if (relative) {
        sprintf(pC_>outString_, "%d MR %d", axisIndex_, NINT(position));
    } else {
        sprintf(pC_>outString_, "%d MV %d", axisIndex_, NINT(position));
    }
    status = pC_>writeReadController();

    // If controller has a "go" command, send it here
    return status;
}
```

Implementing VirtualMotorAxis methods: move

```
/* Arguments in terms of motor record fields:
 *   position (steps) = RVAL = DVAL / MRES
 *   baseVelocity (steps/s) = VBAS / abs(MRES)
 *   velocity (step/s) = VELO / abs(MRES)
 *   acceleration (step/s/s) = (velocity - baseVelocity) / ACCL */
asynStatus VirtualMotorAxis::move(double position, int relative, double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;
    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    if (relative) {
        sprintf(pC_>outString_, "%d MR %d", axisIndex_, NINT(position));
    } else {
        sprintf(pC_>outString_, "%d MV %d", axisIndex_, NINT(position));
    }
    status = pC_>writeReadController();

    // If controller has a "go" command, send it here
    return status;
}
```

Implementing VirtualMotorAxis methods:

move

/* Arguments in terms of motor record fields:

* position (steps) = RVAL = DVAL / MRES

* **baseVelocity (steps/s) = VBAS / abs(MRES)**

* velocity (step/s) = VELO / abs(MRES)

* acceleration (step/s/s) = (velocity - baseVelocity) / ACCL */

asynStatus VirtualMotorAxis::move(double position, int relative, double **minVelocity**, double maxVelocity, double acceleration)

{

asynStatus status;

status = sendAccelAndVelocity(acceleration, maxVelocity, **minVelocity**);

if (relative) {

 sprintf(pC_->outString_, "%d MR %d", axisIndex_, NINT(position));

} else {

 sprintf(pC_->outString_, "%d MV %d", axisIndex_, NINT(position));

}

status = pC_->writeReadController();

// If controller has a "go" command, send it here

return status;

}

Implementing VirtualMotorAxis methods:

move

```
/* Arguments in terms of motor record fields:
 *   position (steps) = RVAL = DVAL / MRES
 *   baseVelocity (steps/s) = VBAS / abs(MRES)
 *   velocity (step/s) = VELO / abs(MRES)
 *   acceleration (step/s/s) = (velocity - baseVelocity) / ACCL */
asynStatus VirtualMotorAxis::move(double position, int relative, double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;
    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    if (relative) {
        sprintf(pC_>outString_, "%d MR %d", axisIndex_, NINT(position));
    } else {
        sprintf(pC_>outString_, "%d MV %d", axisIndex_, NINT(position));
    }
    status = pC_>writeReadController();

    // If controller has a "go" command, send it here
    return status;
}
```

Implementing VirtualMotorAxis methods:

move

```
/* Arguments in terms of motor record fields:
 *   position (steps) = RVAL = DVAL / MRES
 *   baseVelocity (steps/s) = VBAS / abs(MRES)
 *   velocity (step/s) = VELO / abs(MRES)
 *   acceleration (step/s/s) = (velocity - baseVelocity) / ACCL */
asynStatus VirtualMotorAxis::move(double position, int relative, double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;
    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    if (relative) {
        sprintf(pC_>outString_, "%d MR %d", axisIndex_, NINT(position));
    } else {
        sprintf(pC_>outString_, "%d MV %d", axisIndex_, NINT(position));
    }
    status = pC_>writeReadController();

    // If controller has a "go" command, send it here
    return status;
}
```

Implementing VirtualMotorAxis methods:

move

```
/* Arguments in terms of motor record fields:
 *   position (steps) = RVAL = DVAL / MRES
 *   baseVelocity (steps/s) = VBAS / abs(MRES)
 *   velocity (step/s) = VELO / abs(MRES)
 *   acceleration (step/s/s) = (velocity - baseVelocity) / ACCL */
asynStatus VirtualMotorAxis::move(double position, int relative, double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;
    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    if (relative) {
        sprintf(pC_>outString_, "%d MR %d", axisIndex_, NINT(position));
    } else {
        sprintf(pC_>outString_, "%d MV %d", axisIndex_, NINT(position));
    }
    status = pC_>writeReadController();

    // If controller has a "go" command, send it here
    return status;
}
```

Implementing VirtualMotorAxis methods: sendAccelAndVelocity

/* sendAccelAndVelocity() is called by VirtualMotorAxis methods that result in the motor moving: move(), moveVelocity(), home()

* Arguments in terms of motor record fields:

* baseVelocity (steps/s) = VBAS / abs(MRES)

* velocity (step/s) = depends on calling method

* acceleration (step/s/s) = depends on calling method */

asynStatus VirtualMotorAxis::sendAccelAndVelocity(double acceleration, double velocity, double baseVelocity)

{

asynStatus status;

// Send the base velocity

sprintf(pC_>outString_, "%d BAS %f", axisIndex_, baseVelocity);

status = pC_>writeReadController();

// Send the velocity

sprintf(pC_>outString_, "%d VEL %f", axisIndex_, velocity);

status = pC_>writeReadController();

// Send the acceleration

sprintf(pC_>outString_, "%d ACC %f", axisIndex_, acceleration);

status = pC_>writeReadController();

return status; }

Implementing VirtualMotorAxis methods: sendAccelAndVelocity

/* sendAccelAndVelocity() is called by VirtualMotorAxis methods that result in the motor moving: move(), moveVelocity(), home()

* Arguments in terms of motor record fields:

* **baseVelocity (steps/s) = VBAS / abs(MRES)**

* velocity (step/s) = depends on calling method

* acceleration (step/s/s) = depends on calling method */

asynStatus VirtualMotorAxis::sendAccelAndVelocity(double acceleration, double velocity, double **baseVelocity**)

{

asynStatus status;

// Send the base velocity

sprintf(pC_>outString_, "%d BAS %f", axisIndex_, **baseVelocity**);

status = pC_>writeReadController();

// Send the velocity

sprintf(pC_>outString_, "%d VEL %f", axisIndex_, velocity);

status = pC_>writeReadController();

// Send the acceleration

sprintf(pC_>outString_, "%d ACC %f", axisIndex_, acceleration);

status = pC_>writeReadController();

return status; }

Implementing VirtualMotorAxis methods: sendAccelAndVelocity

/* sendAccelAndVelocity() is called by VirtualMotorAxis methods that result in the motor moving: move(), moveVelocity(), home()

* Arguments in terms of motor record fields:

* baseVelocity (steps/s) = VBAS / abs(MRES)

* **velocity (step/s) = depends on calling method**

* acceleration (step/s/s) = depends on calling method */

asynStatus VirtualMotorAxis::sendAccelAndVelocity(double acceleration, double **velocity**, double baseVelocity)

{

asynStatus status;

// Send the base velocity

sprintf(pC_>outString_, "%d BAS %f", axisIndex_, baseVelocity);

status = pC_>writeReadController();

// Send the velocity

sprintf(pC_>outString_, "%d VEL %f", axisIndex_, **velocity**);

status = pC_>writeReadController();

// Send the acceleration

sprintf(pC_>outString_, "%d ACC %f", axisIndex_, acceleration);

status = pC_>writeReadController();

return status; }

Implementing VirtualMotorAxis methods: sendAccelAndVelocity

/* sendAccelAndVelocity() is called by VirtualMotorAxis methods that result in the motor moving: move(), moveVelocity(), home()

* Arguments in terms of motor record fields:

* baseVelocity (steps/s) = VBAS / abs(MRES)

* velocity (step/s) = depends on calling method

* **acceleration (step/s/s) = depends on calling method** */

asynStatus VirtualMotorAxis::sendAccelAndVelocity(double **acceleration**, double velocity, double baseVelocity)

{

asynStatus status;

// Send the base velocity

sprintf(pC_>outString_, "%d BAS %f", axisIndex_, baseVelocity);

status = pC_>writeReadController();

// Send the velocity

sprintf(pC_>outString_, "%d VEL %f", axisIndex_, velocity);

status = pC_>writeReadController();

// Send the acceleration

sprintf(pC_>outString_, "%d ACC %f", axisIndex_, **acceleration**);

status = pC_>writeReadController();

return status; }

Implementing VirtualMotorAxis methods: setPosition

```
/*
 * setPosition() is called by asynMotor device support when a position is redefined.
 * It is also required for autosave to restore a position to the controller at iocInit.
 *
 * Arguments in terms of motor record fields:
 *   position (steps) = DVAL / MRES = RVAL
 */
asynStatus VirtualMotorAxis::setPosition(double position)
{
    asynStatus status;

    sprintf(pC_>outString_, "%d POS %d", axisIndex_, NINT(position));
    status = pC_>writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: setPosition

```
/*
 * setPosition() is called by asynMotor device support when a position is redefined.
 * It is also required for autosave to restore a position to the controller at iocInit.
 *
 * Arguments in terms of motor record fields:
 *   position (steps) = DVAL / MRES = RVAL
 */
asynStatus VirtualMotorAxis::setPosition(double position)
{
    asynStatus status;

    sprintf(pC_>outString_, "%d POS %d", axisIndex_, NINT(position));
    status = pC_>writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: moveVelocity (jog)

```
/*
 * moveVelocity() is called by asynMotor device support when a jog is requested.
 * If a controller doesn't have a jog command (or jog commands), this a jog can be simulated here.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = (jog_direction == forward) ? (JVEL * DIR / MRES) : (-1 * JVEL * DIR / MRES)
 *   acceleration (step/s/s) = JAR / abs(EGU)
 */
asynStatus VirtualMotorAxis::moveVelocity(double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;

    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    sprintf(pC_>outString_, "%d JOG %f", axisIndex_, maxVelocity);
    status = pC_>writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: moveVelocity (jog)

```
/*
 * moveVelocity() is called by asynMotor device support when a jog is requested.
 * If a controller doesn't have a jog command (or jog commands), this a jog can be simulated here.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) =  $VBAS / \text{abs}(MRES)$ 
 *   maxVelocity (step/s) = (jog_direction == forward) ? (JVEL * DIR / MRES) : (-1 * JVEL * DIR / MRES)
 *   acceleration (step/s/s) = JAR / abs(EGU)
 */
asynStatus VirtualMotorAxis::moveVelocity(double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;

    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    sprintf(pC_>outString_, "%d JOG %f", axisIndex_, maxVelocity);
    status = pC_>writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: moveVelocity (jog)

```
/*
 * moveVelocity() is called by asynMotor device support when a jog is requested.
 * If a controller doesn't have a jog command (or jog commands), this a jog can be simulated here.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = (jog_direction == forward) ? (JVEL * DIR / MRES) : (-1 * JVEL * DIR / MRES)
 *   acceleration (step/s/s) = JAR / abs(EGU)
 */
asynStatus VirtualMotorAxis::moveVelocity(double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;

    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    sprintf(pC_>outString_, "%d JOG %f", axisIndex_, maxVelocity);
    status = pC_>writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: moveVelocity (jog)

```
/*
 * moveVelocity() is called by asynMotor device support when a jog is requested.
 * If a controller doesn't have a jog command (or jog commands), this a jog can be simulated here.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = (jog_direction == forward) ? (JVEL * DIR / MRES) : (-1 * JVEL * DIR / MRES)
 *   acceleration (step/s/s) = JAR / abs(EGU)
 */
asynStatus VirtualMotorAxis::moveVelocity(double minVelocity, double maxVelocity, double acceleration)
{
    asynStatus status;

    status = sendAccelAndVelocity(acceleration, maxVelocity, minVelocity);

    sprintf(pC_>outString_, "%d JOG %f", axisIndex_, maxVelocity);
    status = pC_>writeReadController();
    return status;
}
```

Implementing VirtualMotorAxis methods: home

```
/*
 * home() is called by asynMotor device support when a home is requested.
 * Note: forwards is set by device support, NOT by the motor record.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = HVEL / abs(MRES)
 *   acceleration (step/s/s) = (maxVelocity - minVelocity) / ACCL
 *   forwards = 1 if HOMF was pressed, 0 if HOMR was pressed
 */
/*
asynStatus VirtualMotorAxis::home(double minVelocity, double maxVelocity, double acceleration, int forwards)
{

    // Homing isn't currently implemented

    return asynSuccess;
}
*/
```

Implementing VirtualMotorAxis methods: home

```
/*
 * home() is called by asynMotor device support when a home is requested.
 * Note: forwards is set by device support, NOT by the motor record.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = HVEL / abs(MRES)
 *   acceleration (step/s/s) = (maxVelocity - minVelocity) / ACCL
 *   forwards = 1 if HOMF was pressed, 0 if HOMR was pressed
 */
/*
asynStatus VirtualMotorAxis::home(double minVelocity, double maxVelocity, double acceleration, int forwards)
{

    // Homing isn't currently implemented

    return asynSuccess;
}
*/
```

Implementing VirtualMotorAxis methods: home

```
/*
 * home() is called by asynMotor device support when a home is requested.
 * Note: forwards is set by device support, NOT by the motor record.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = HVEL / abs(MRES)
 *   acceleration (step/s/s) = (maxVelocity - minVelocity) / ACCL
 *   forwards = 1 if HOMF was pressed, 0 if HOMR was pressed
 */
/*
asynStatus VirtualMotorAxis::home(double minVelocity, double maxVelocity, double acceleration, int forwards)
{

    // Homing isn't currently implemented

    return asynSuccess;
}
*/
```

Implementing VirtualMotorAxis methods: home

```
/*
 * home() is called by asynMotor device support when a home is requested.
 * Note: forwards is set by device support, NOT by the motor record.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = HVEL / abs(MRES)
 *   acceleration (step/s/s) = (maxVelocity - minVelocity) / ACCL
 *   forwards = 1 if HOMF was pressed, 0 if HOMR was pressed
 */
/*
asynStatus VirtualMotorAxis::home(double minVelocity, double maxVelocity, double acceleration, int forwards)
{

    // Homing isn't currently implemented

    return asynSuccess;
}
*/
```

Implementing VirtualMotorAxis methods: home

```
/*
 * home() is called by asynMotor device support when a home is requested.
 * Note: forwards is set by device support, NOT by the motor record.
 *
 * Arguments in terms of motor record fields:
 *   minVelocity (steps/s) = VBAS / abs(MRES)
 *   maxVelocity (step/s) = HVEL / abs(MRES)
 *   acceleration (step/s/s) = (maxVelocity - minVelocity) / ACCL
 *   forwards = 1 if HOMF was pressed, 0 if HOMR was pressed
 */
/*
asynStatus VirtualMotorAxis::home(double minVelocity, double maxVelocity, double acceleration, int forwards)
{

    // Homing isn't currently implemented

    return asynSuccess;
}
*/
```

Final Thought

The model-3 driver for the Virtual Motor Controller was written to handle every value sent by the motor record without having to convert it. This makes it an ideal starting point for writing a simple model-3 driver, since there is very little code to remove before implementing support for a new controller.