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:
Prerequisites

- Familiarity with the motor record
- Knowledge of controller commands
- Some programming experience

Installing created dlib file: ./.../abd/localLinux.x64.dlib

Anything else?

Familiarity with the motor record
- Some programming experience
- Knowledge of controller commands

Prerequisites

- Familiarity with the motor record
- Some programming experience
- Knowledge of controller commands
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. Draw some circles
2. Draw the rest of the owl
References

- asynMotor documentation in the motor module
  - [motor/documentation/motorDeviceDriver.html](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](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,Axes}.cpp,h](motor/motorApp/MotorSrc/asynMotor{Controller,Axes}.cpp,h)

- Motor record source code
Road Map
How does the IOC associate the model-3 motor driver with the motor record?

```c
+vmc.cmd

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")
```

```c
+vmc.substitutions

1 file "/$(TOP)/db/asyn_motor.db"
2 {
3   P, N, M, DTPY, PORT, ADDR,
4   (vmc, 1, "$M(N)"), "asynMotor", "VMC1, 0",
5   (vmc, 2, "$M(N)"), "asynMotor", "VMC1, 1",
6   (vmc, 3, "$M(N)"), "asynMotor", "VMC1, 2",
7   (vmc, 4, "$M(N)"), "asynMotor", "VMC1, 3",
8   (vmc, 5, "$M(N)"), "asynMotor", "VMC1, 4",
9   (vmc, 6, "$M(N)"), "asynMotor", "VMC1, 5",
10  (vmc, 7, "$M(N)"), "asynMotor", "VMC1, 6",
11  (vmc, 8, "$M(N)"), "asynMotor", "VMC1, 7",
12 } motorSupport.db
```

```c
+asyn_motor.db

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

---

EPICS Training 2015 – Writing a Motor Controller Driver, Kevin M. Peterson, 2015-02-17
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

motor record

asynMotor device support

asynMotorController & asynMotorAxis base classes

controller-specific asynMotor driver
Road Map

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

controller-specific
asynMotor driver

asynMotorAxis
base classes

asynMotorController
& asynMotorAxis

asynMotor device support

motor record

Road Map
Road Map

- **motor record**
- **asynMotor device support**
- **asynMotorController & asynMotorAxis base classes**
- **controller-specific asynMotor driver**

```c
switch (command) {
    case LOAD_POS:
        pmsg->command = motorPosition;
        pmsg->dvalue = *param;
        pPvt->moveRequestPending++;
        break;
}
```
else if (function == motorPosition) {
    status = pAxis->setPosition(value);
    pAxis->callParamCallbacks();
}
Road Map

motor record

asynMotor device support

asynMotorController & asynMotorAxis base classes

controller-specific asynMotor driver

```cpp
asynStatus VirtualMotorAxis::setPosition(double position) {
    asynStatus status;
    sprintf(pC_->outString_, "%d POS %d", axisIndex_, NINT(position));
    status = pC_->writeReadController();
    return status;
}
```
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 `asynMotorController,Axis` methods need to be implemented?

- There are many methods that could be implemented

```c
asynMotorController::asynMotorController(const char*, 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, 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
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) : asynStatus
asynMotorAxis::home(double, double, double, int) : asynStatus
asynMotorAxis::stop(double) : asynStatus
asynMotorAxis::poll(bool) : asynStatus
asynMotorAxis::setIgain(double) : asynStatus
asynMotorAxis::setDgain(double) : asynStatus
asynMotorAxis::setCloseLoop(bool) : asynStatus
asynMotorAxis::setEncoderRatio(double) : asynStatus
asynMotorAxis::report(FILE*, int) : void
asynMotorAxis::doMoveToHome() : asynStatus
asynMotorAxis::setReferencingMove(int) : void
asynMotorAxis::getReferencingMove() : 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

```c
typedef enum {
    MOVE_ABS, /* Absolute Move. */
    MOVE_REL, /* Relative Move. */
    HOME_FWD, /* Home Forward. */
    HOME_REV, /* Home Reverse. */
    LOAD_POS, /* Load Position. */
    SET_VEL_BASE, /* Set Minimum Velocity. */
    SET_VELOCITY, /* Set Jog and Trajectory Velocity. */
    SET_ACCEL, /* Set Acceleration. */
    GO, /* Start previously programmed move. */
    SET_ENC_RATIO, /* Set Encoder Ratio. */
    GET_INFO, /* Update Motor Status. */
    STOP_AXIS, /* Stop Axis Motion. */
    JOG, /* Momentary Jog. */
    SET_PGAII, /* Set Proportional Gain. */
    SET_I GAIN, /* Set Integral Gain. */
    SET_DGAII, /* Set Derivative Gain. */
    ENABLE_TORQUE, /* Enable Servo Closed-Loop Control. */
    DISABLE_TORQUE, /* Disable Servo Closed-Loop Control. */
    PRIMITIVE, /* Primitive Controller command. */
    SET_HIGH_LIMIT, /* Set High Travel Limit. */
    SET_LOW_LIMIT, /* Set Low Travel Limit. */
    JOG_VELOCITY, /* Change Jog velocity. */
    SET_RESOLUTION /* Set resolution */
} motor_cmd;
```
Which asynMotor{Controller,Axis} methods need to be implemented?

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

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

```cpp
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 \texttt{asynMotor\{Controller,Axis\}} methods need to be implemented?

- The Virtual Motor Controller driver implements these methods

\begin{verbatim}
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
\end{verbatim}
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
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.
 */

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

```cpp
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

```cpp
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

```cpp
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

```cpp
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

```c
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

```cpp
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 ? 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) {
        printf(pC_->outString_, "%d MR %d", axisIndex_, NINT(position));
    } else {
        printf(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_-&gt;outString_, "%d BAS %f", axisIndex_, baseVelocity);
    status = pC_-&gt;writeReadController();

    // Send the velocity
    sprintf(pC_-&gt;outString_, "%d VEL %f", axisIndex_, velocity);
    status = pC_-&gt;writeReadController();

    // Send the acceleration
    sprintf(pC_-&gt;outString_, "%d ACC %f", axisIndex_, acceleration);
    status = pC_-&gt;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 */

```c
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_-&gt;outString_, "%d JOG %f", axisIndex_, maxVelocity);
    status = pC_-&gt;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);

    snprintf(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 (steps/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.