Argonne NATIONAL LABORATORY

How Motor Support Works (asynMotor)

17 February 2015

Joe Sullivan – EPICS Software Developer
Beamline Controls and Data Acquisition
Advanced Photon Source

U.S. DEPARTMENT OF ENERGY
Office of Science

Outline

- Acknowledgments
- Motor Support Development Framework
- Controller Level Support (asynMotorController)
- Axis Level Support (asynMotorAxis)

Acknowledgments

- Original asynMotor Developers (Motor R5.9 – 2006)
  - Peter Denision (Diamond), Nick Rees (Diamond) and Mark Rivers (APS)
- Current Version (Model 3)
  - Mark Rivers (APS)
- Content references for this class
  - **Model 3 EPICS Motor Driver Support**, 2012 EPICS Collaboration Meeting (Mark Rivers)
    - https://portal.slac.stanford.edu/sites/conf_public/epics_2012_04/presentations/Rivers_Thursday_EPICS_Motor_Device.pdf
  - **asynMotor Class Documentation**, 2012 Doxygen Generated (Mark Rivers)
    - http://cars9.uchicago.edu/software/epics/motorDoxygenHTML/classasyn_motor_controller.html

EPICS Motor Support

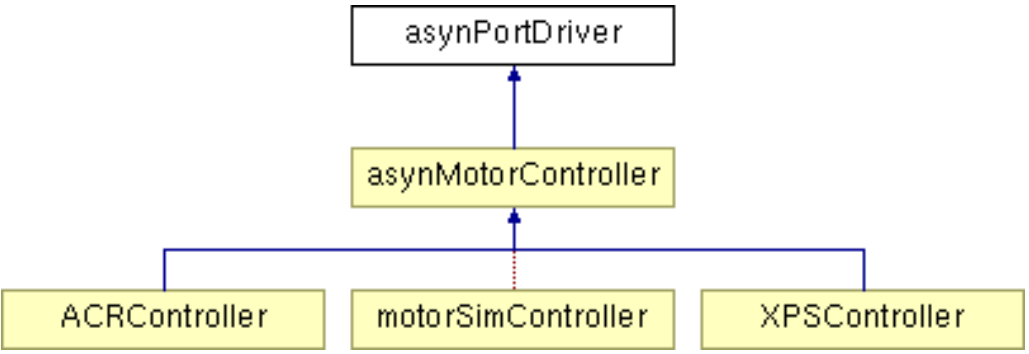
- Top-level object is the EPICS **motor record**
  - Lots of code has been written to this object:
    - spec, IDL and Python classes, etc.
- Next layer is EPICS **device support**
  - Knows about the motor record, talks to the driver
- Lowest layer is EPICS **driver**
  - Knows nothing about motor record, talks to the hardware

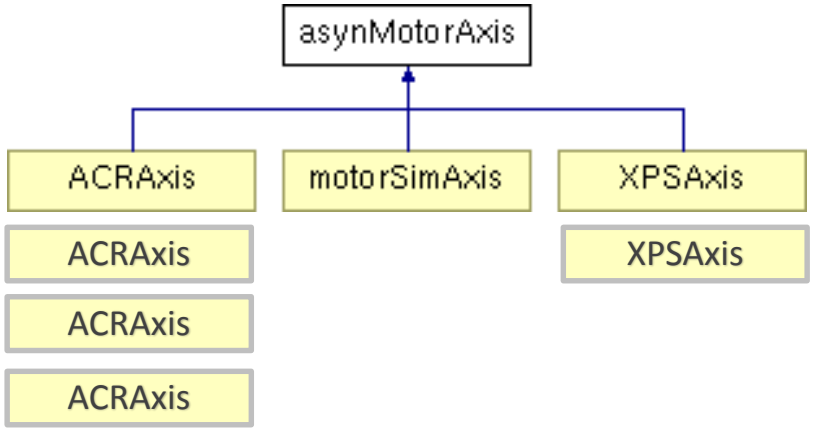
*Current recommended device and driver development framework is **asynMotor** **(Model 3).***

- **asynPortDriver** is used to extend this framework for hardware features not supported by the motorRecord.

asynMotor (Model 3)

- Available in the synApps **motor** module since 2011.
- C++ model.
- Two base classes, asynMotorController and asynMotorAxis.
- Base classes provide much functionality, only need to write device-specific implementations.
- Easy to support controller-specific features
- Direct support for coordinated profile moves in the driver API.





synApps based motor driver development

- synApps motor module (requires ASYN, BUSY, IPAC and SNCSEQ)
  - motorRecord ($MOTOR/motorApp/MotorSrc)
  - asynMotor Device Support (devMotorAsyn.c)
  - asynMotor Driver Base Classes
    - asynMotorController.h, .cpp
    - asynMotorAxis.h , .cpp

- Motor driver code to implement controller specific methods from asynMotor base classes
  - ***<Name>***MotorDriver.cpp
  - ***<Name>***MotorDriver.h
  - ***<Name>***MotorSupport.dbd

What does the ‘Controller’ constructor do?

- Creates a Controller object (asynPortDriver) that implements a ‘real’ motor controller.
  - Necessary initializing of motor controller hardware.
- Call the ‘Axis’ constructor for every motor channel requested in the ‘CreateController’ argument for number of axis.
- Starts the Poller task that will update the status of all the motor channels.
- Implement non-motorRecord features.
  - asynPortDriver methods for additional controller parameters.
  - Built-in ‘Profile Move’ (coordinated multi-axis) methods.

ACRController - Constructor

Called before iocInit() - from st.cmd

- asynMotor.cmd (BCDA standard IOC startup file)

ACR Example: motor/iocBoot/iocWithAsyn/st.cmd.acr

portName, asynPort, number of axes, active poll period (ms), idle poll period (ms)

*ACR*CreateController("ACR1", "ARIES", 1, 20, 1000)

```
extern "C" int ACRCreateController(const char *portName,
                                   const char *ACRPortName,
                                   int numAxes,
                                   int movingPollPeriod,
                                   int idlePollPeriod)
{
  new ACRController(portName, ACRPortName, numAxes,
                    movingPollPeriod/1000., idlePollPeriod/1000.);
```

- *<synApps>/motor/motorApp/ACRSrc/ACRMotorDriver.cpp, M. Rivers*

ACRController Constructor

```
new ACRController(portName, ACRPortName, numAxes,
                  movingPollPeriod/1000., idlePollPeriod/1000.);
```

ACRController::ACRController (const char * portName,
const char * ACRPortName,
int numAxes,
double movingPollPeriod,
double idlePollPeriod
)

Creates a new **ACRController** object.

Parameters:

[in]	**portName**	The name of the asyn port that will be created for this driver
[in]	**ACRPortName**	The name of the drvAsynIPPPort that was created previously to connect to the ACR controller
[in]	**numAxes**	The number of axes that this controller supports
[in]	**movingPollPeriod**	The time between polls when any axis is moving
[in]	**idlePollPeriod**	The time between polls when no axis is moving

- *doxygen asynMotor page generated 2012, M. Rivers*

ACRController Constructor

```
ACRController::ACRController(const char *portName, const char *ACRPortName, int numAxes,
                             double movingPollPeriod, double idlePollPeriod)
  :  asynMotorController(portName, numAxes, NUM_ACR_PARAMS,
                         asynUInt32DigitalMask,
                         asynUInt32DigitalMask,
                         ASYN_CANBLOCK | ASYN_MULTIDEVICE,
                         1, // autoconnect
                         0, 0)  // Default priority and stack size
{
```

asynMotorController::asynMotorController (const char * portName,
int numAxes,
int numParams,
int interfaceMask,
int interruptMask,
int asynFlags,
int autoConnect,
int priority,
int stackSize
)

Creates a new **asynMotorController** object.

All of the arguments are simply passed to the constructor for the asynPortDriver base class. After calling the base class constructor this method creates the motor parameters defined in asynMotorDriver.h.

asynPortDriver – Constructor (review)

asynPortDriver Parameters:

[in]	**portName**	The name of the asyn port that will be created for this driver
[in]	**numAxes**	The number of axes that this controller supports
[in]	**numParams**	Number of parameters unique to the controller (do not exist in the motorRecord).
[in]	**interfaceMask**	asyn interfaces unique to the controller (bit mask)
[in]	**interruptMask**	enable callbacks for unique asyn interfaces (bit mask)
[in]	**asynFlags**	ASYN_CANBLOCK(asynchronous), ASYN_MULTIDEVICE(addressed)
[in]	**autoConnect**	Yes/No
[in]	**priority**	For port thread if ASYN_CANBLOCK
[in]	**stackSize**	For port thread if ASYN_CANBLOCK

ACRController Constructor (cont)

```
ACRController::ACRController(const char *portName, const char *ACRPortName, int numAxes,
                             double movingPollPeriod, double idlePollPeriod)
  :  asynMotorController(portName, numAxes, NUM_ACR_PARAMS,
                         asynUInt32DigitalMask,
                         asynUInt32DigitalMask,
                         ASYN_CANBLOCK | ASYN_MULTIDEVICE,
                         1, // autoconnect
                         0, 0)  // Default priority and stack size
{
```

```
// Create controller-specific parameters
createParam(ACRJerkString,         asynParamFloat64,       &ACRJerk_);
createParam(ACRReadBinaryIOString, asynParamInt32,         &ACRReadBinaryIO_);

/* Connect to ACR controller */
status = pasynOctetSyncIO->connect(ACRPortName, 0, &pasynUserController_, NULL);

// Turn off command echoing
sprintf(outString_, "ECHO 4");
writeController();
```

- *<synApps>/motor/motorApp/ACRSrc/ACRMotorDriver.cpp, M. Rivers*

ACRController Constructor (cont.)

```
// Create the axis objects
for (axis=0; axis<numAxes; axis++) {
  new ACRAxis(this, axis);
}
```

ACRAxis::ACRAxis (class ACRController * pC,
int axisNo
)

Creates a new **ACRAxis** object.

Parameters:

[in]	**pC**	Pointer to the **ACRController** to which this axis belongs.
[in]	**axisNo**	Index number of this axis, range 0 to pC->numAxes_-1.

Initializes register numbers, etc.

- *doxygen asynMotor page generated 2012, M. Rivers*
- *<synApps>/motor/motorApp/ACRSrc/ACRMotorDriver.cpp, M. Rivers*

ACRController Constructor (cont)

```
  startPoller(movingPollPeriod, idlePollPeriod, 2);
}
```

asynStatus asynMotorController::startPoller (double movingPollPeriod,
double idlePollPeriod,
int forcedFastPolls
) [virtual]

Starts the motor poller thread.

Derived classes will typically call this at near the end of their constructor. Derived classes can typically use the base class implementation of the poller thread, but are free to reimplement it if necessary.

Parameters:

[in]	**movingPollPeriod**	The time between polls when any axis is moving.
[in]	**idlePollPeriod**	The time between polls when no axis is moving.
[in]	**forcedFastPolls**	The number of times to force the movingPollPeriod after waking up the poller. This can need to be non-zero for controllers that do not immediately report that an axis is moving after it has been told to start.

- *<synApps>/motor/motorApp/ACRSrc/ACRMotorDriver.cpp, M. Rivers*

asynMotorController Method ACRController Custom Parameter

```
asynStatus ACRController::writeFloat64(asynUser *pasynUser, epicsFloat64 value)
{
  int function = pasynUser->reason;
  asynStatus status = asynSuccess;
  ACRAxis *pAxis = getAxis(pasynUser);
  static const char *functionName = "writeFloat64";
```

asynStatus ACRController::writeFloat64 (asynUser * pasynUser,
epicsFloat64 value
) [virtual]

Called when asyn clients call pasynFloat64->write().

Extracts the function and axis number from pasynUser. Sets the value in the parameter library. If the function is ACRJerk_ it sets the jerk value in the controller. Calls any registered callbacks for this pasynUser->reason and address. For all other functions it calls **asynMotorController::writeFloat64**.

Parameters:

[in]	**pasynUser**	asynUser structure that encodes the reason and address.
[in]	**value**	Value to write.

Reimplemented from **asynMotorController**.

asynMotorController Method ACRController Custom Parameter (cont)

```
asynStatus ACRController::writeFloat64(asynUser *pasynUser, epicsFloat64 value)
{
  int function = pasynUser->reason;
  asynStatus status = asynSuccess;
  ACRAxis *pAxis = getAxis(pasynUser);
  static const char *functionName = "writeFloat64";


if (function == ACRJerk_)
{
  sprintf(outString_, "%s JOG JRK %f", pAxis->axisName_, value);
  status = writeController();

} else {
  /* Call base class method */
  status = asynMotorController::writeFloat64(pasynUser, value);
}



/* Do callbacks so higher layers see any changes */
pAxis->callParamCallbacks();
```

asynMotorAxis – Constructor

- Creates a Axis object that is customized to a particular motor controller.
  - Necessary initialization of motor axis in controller hardware.
  - Characterize the axis with respect to the motorRecord
    - GainSupport, HasEncoder,…
  - Implement methods to provide all available motorRecord functionality.
    - Move, MoveVelocity, Home, SetPosition, …
- Implement ‘report’ method.

asynMotorAxis Constructor
ACRAxis

```
ACRAxis::ACRAxis(ACRController *pC, int axisNo)
  : asynMotorAxis(pC, axisNo),
    pC_(pC)
{
```

asynMotorAxis::asynMotorAxis (class asynMotorController * pC,
int axisNo
)

Creates a new **asynMotorAxis** object.

Parameters:

[in] **pC** Pointer to the **asynMotorController** to which this axis belongs.

[in] **axisNo** Index number of this axis, range 0 to pC->numAxes_-1.

Checks that pC is not null, and that axisNo is in the valid range. Sets a pointer to itself in pC->pAxes[axisNo_]. Connects pasynUser_ to this asyn port and axisNo.

- *<synApps>/motor/motorApp/ACRSrc/ACRMotorDriver.cpp, M. Rivers*

asynMotorAxis Methods
ACRAxis Move

```
asynStatus ACRAxis::move(double position, int relative,
                         double minVelocity, double maxVelocity,
                         double acceleration)
{
```

asynStatus ACRAxis::move (double position,
int relative,
double minVelocity,
double maxVelocity,
double acceleration
) [virtual]

Move the motor to an absolute location or by a relative amount.

Parameters:

[in]	**position**	The absolute position to move to (if relative=0) or the relative distance to move by (if relative=1). Units=steps.
[in]	**relative**	Flag indicating relative move (1) or absolute move (0).
[in]	**minVelocity**	The initial velocity, often called the base velocity. Units=steps/sec.
[in]	**maxVelocity**	The maximum velocity, often called the slew velocity. Units=steps/sec.
[in]	**acceleration**	The acceleration value. Units=steps/sec/sec.

Reimplemented from **asynMotorAxis**.

asynMotorAxis Methods
ACRAxis Move

```
asynStatus ACRAxis::move(double position, int relative,
                         double minVelocity, double maxVelocity,
                         double acceleration)
{

sprintf(pC_->outString_, "%s JOG ACC %f", axisName_, acceleration/pulsesPerUnit_);
status = pC_->writeController();
sprintf(pC_->outString_, "%s JOG VEL %f", axisName_, maxVelocity/pulsesPerUnit_);
status = pC_->writeController();
// Note, the CtlY being send below clears the kill for all axes, in case they had hit a l
if (relative) {
  sprintf(pC_->outString_, "%c:%s JOG INC %f", CtlY, axisName_, position/pulsesPerUnit_);
  status = pC_->writeController();
} else {
  sprintf(pC_->outString_, "%c:%s JOG ABS %f", CtlY, axisName_, position/pulsesPerUnit_);
  status = pC_->writeController();
}
```

asynMotorAxis Methods
ACRAxis MoveVelocity

```
asynStatus ACRAxis::moveVelocity ( double minVelocity,
                                   double maxVelocity,
                                   double acceleration
                                 )       [virtual]
```

Move the motor at a fixed velocity until told to stop.

Parameters:

[in]	**minVelocity**	The initial velocity, often called the base velocity. Units=steps/sec.
[in]	**maxVelocity**	The maximum velocity, often called the slew velocity. Units=steps/sec.
[in]	**acceleration**	The acceleration value. Units=steps/sec/sec.

Reimplemented from **asynMotorAxis**.

asynMotorAxis Methods
ACRAxis Poll

```
asynStatus ACRAxis::poll(bool *moving)
{
```

asynStatus ACRAxis::poll (bool * moving) [virtual]

Polls the axis.

This function reads the controller position, encoder position, the limit status, the moving status, and the drive power-on status. It does not current detect following error, etc. but this could be added. It calls **setIntegerParam()** and **setDoubleParam()** for each item that it polls, and then calls **callParamCallbacks()** at the end.

Parameters:

[out] **moving** A flag that is set indicating that the axis is moving (1) or done (0).

Reimplemented from **asynMotorAxis**.

asynMotorAxis Methods
ACRAxis Poll (cont)

```
asynStatus ACRAxis::poll(bool *moving)
{

// Read the current encoder position
sprintf(pC_->outString_, "?P%d", encoderPositionReg_);
comStatus = pC_->writeReadController();
if (comStatus) goto skip;
encoderPosition_ = atof(pC_->inString_);
setDoubleParam(pC_->motorEncoderPosition_,encoderPosition_);

// Read the current flags
sprintf(pC_->outString_, "?P%d", flagsReg_);
comStatus = pC_->writeReadController();
if (comStatus) goto skip;
currentFlags_ = atoi(pC_->inString_);
done = (currentFlags_ & 0x1000000)?0:1;
setIntegerParam(pC_->motorStatusDone_, done);
*moving = done ? false:true;
```

asynMotorAxis Methods
ACRAxis Poll (cont)

```
asynStatus ACRAxis::poll(bool *moving)
{
// Read the current limit status
sprintf(pC_->outString_, "?P%d", limitsReg_);
comStatus = pC_->writeReadController();
if (comStatus) goto skip;
currentLimits_ = atoi(pC_->inString_);
limit = (currentLimits_ & 0x1)?1:0;
setIntegerParam(pC_->motorStatusHighLimit_, limit);
limit = (currentLimits_ & 0x2)?1:0;
setIntegerParam(pC_->motorStatusLowLimit_, limit);
limit = (currentLimits_ & 0x4)?1:0;
setIntegerParam(pC_->motorStatusAtHome_, limit);
```

```
  skip:
  setIntegerParam(pC_->motorStatusProblem_, comStatus ? 1:0);
  callParamCallbacks();
  return comStatus ? asynError : asynSuccess;
}
```

asynMotorAxis Methods
ACRAxis Home

asynStatus ACRAxis::home (double minVelocity,
double maxVelocity,
double acceleration,
int forwards
) [virtual]

Move the motor to the home position.

Parameters:

[in]	**minVelocity**	The initial velocity, often called the base velocity. Units=steps/sec.
[in]	**maxVelocity**	The maximum velocity, often called the slew velocity. Units=steps/sec.
[in]	**acceleration**	The acceleration value. Units=steps/sec/sec.
[in]	**forwards**	Flag indicating to move the motor in the forward direction(1) or reverse direction(0). Some controllers need to be told the direction, others know which way to go to home.

Reimplemented from **asynMotorAxis**.

asynMotorAxis Methods
ACRAxis Report

void ACRAxis::report (FILE * fp,
int level
) [virtual]

Reports on status of the driver.

Parameters:

[in]	**fp**	The file pointer on which report information will be written
[in]	**level**	The level of report detail desired

If details > 0 then information is printed about each axis. After printing controller-specific information calls **asynMotorController::report()**

Reimplemented from **asynMotorAxis**.

asynMotorAxis Methods
ACRAxis setPosition

asynStatus ACRAxis::setPosition (double position) [virtual]

Set the current position of the motor.

Parameters:

[in] **position** The new absolute motor position that should be set in the hardware. Units=steps.

Reimplemented from **asynMotorAxis**.

asynMotorAxis Methods
ACRAxis Stop

asynStatus ACRAxis::stop (double acceleration) [virtual]

Stop the motor.

Parameters:

[in] **acceleration** The acceleration value. Units=steps/sec/sec.

Reimplemented from **asynMotorAxis**.

Class Class Method
Stop