Motor Record Device Support

COLLABORATORS

	TITLE : Motor Record Device Support		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Michael Davidsaver, Daron Chabot	February 2010	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1	January 2010	Initial revision	MAD

Contents

1 Introduction

The EPICS motorRecord is the most widely used of the special (application) record types. It is the most common interface for control of a single axis. This document will describe the device support interface of the motorRecord, how the record uses it, and what it requires of a device support implementation.

The motorRecord is typically used in conjunction with the *common device support*, a library of functions of to help build device support. This library has been extended to work closely with the Asyn hardware access abstraction layer. However, it is a large body of code which contains no small amount of magic. The authors of this document must admit to not being able to understand and use this code base.

Consequently this document will present a minimal complete example of motorRecord device support without using the *common device support*. We hope that it will serve to illistrate what the motorRecord requires, and what the *common device support* must provide.

2 Hardware Model

The Motor record is built around the idea of a device which accepts commands and returns responses serially. Status information may be polled or come asynchronously. Polling the device for status is a costly operation, so the device may only be regularly polled when it is needed, usually while the axis is in motion.

Historically a controller was a RS-232 or similar device with no asynchronous interrupts. Multiple axes shared the same link so bandwidth was an issue.

Functionally, the device is treated as a stepper motor. It is sent move commands as discrete steps and the device (or device support) is expected to maintain a signed counter of the number of commanded steps since IOC initialization. Additionally, the device may provide an encoder for feedback.

Status for two limit switches and a home switch can be monitored.

3 Types of Motion

The motor record supports several different kinds of motion: normal, backlash, home, and jog. All controllers will support normal motions. Backlash motion is implemented by the record as a normal motion so no additional effort is required. Home and Jog motions require that device support implement the required commands.

3.1 Normal

Moves a finite number of steps to a user commanded destination which is known before the move begins.

Required commands: MOVE_ABS, MOVE_REL

Optional commands: SET_VEL_BASE, SET_VELOCITY, SET_ACCEL

3.2 Jog

This is a constant velocity move for a user determined length of time which is not known when the move begins.

Required commands: JOG, STOP_AXIS, JOG_VELOCITY

3.3 Backlash

The motor record will keep track of the direction of the last move. Following the next normal or jog move in the opposite direction, a backlash move is initialized. The backlash moves an additional fixed number of steps (TODO direction?) which are not added to the record's position counter.

3.4 Home

A homing motion moves in the selected direction until the home switch is activated. No backlash motion is used after a homing move.

Required commands: HOME_FOR, HOME_REV

Optional commands: SET_VEL_BASE, SET_VELOCITY, SET_ACCEL

4 Processing

In general an EPICS record will only process on one condition: timer, IRQ, event, or external. However, the Motor record is designed to process on several conditions: timer/IRQ and external. It must respond to external commands, especially *STOP* with low latency, while at the same time providing status updates while it motion.

This is accomplished by requiring that the *SCAN* field always be set to *Passive* to ensure that external actions (CA puts) cause immediate processing. It also requires that device support should provide a timer or interrupt handler which will triggers processing. This is left to device support because the implementation will be device dependent.

5 Device Support

The device support entry table, or dset, for the motor-record contains the standard dset (see "base/include/devSup.h"), plus four fields for pointers to functions that are particular to that record-type. The C-structure therefore appears as defined in "motorApp/MotorSrc/motor.h":

```
/* device support entry table */
struct motor_dset
{
    struct dset base;
    CALLBACK_VALUE (*update_values) (struct motorRecord *);
    long (*start_trans) (struct motorRecord *);
    RTN_STATUS (*build_trans) (motor_cmnd, double *, struct motorRecord *);
    RTN_STATUS (*end_trans) (struct motorRecord *);
};
```

The Motor record requires four additional device support functions. Three to construct and send command sequences (*transactions*), and one to receive updates. Communication from the record to the device support is handled through the *transaction* functions, while communication from device support to the record is handled through the update function.

5.1 Commands

Constructing commands involves the *start_trans*, *build_trans*, and *end_trans* functions. These “transactions” are compound command sequences created from a set of primitive actions. The Motor record will first call *start_trans* to clear any previously unsent commands. This is followed by several calls to *build_trans*. At this point the Motor record will either call *end_trans* to “commit” the transaction by sending it to hardware, or call *start_trans* again to abort it.

The following is an incomplete example of what a motor record transaction will look like.

```
double arg=vel;
start_trans(pmr);
build_trans(SET_VELOCITY, &arg, pmr);
if(use_relative){
  arg=dest-cur;
  build_trans(MOVE_REL, &arg, pmr);
}else{
  arg=dest;
  build_trans(MOVE_ABS, &arg, pmr);
}
build_trans(GO, 0, pmr);
end_trans(pmr);
```

Historically this process was used to construct an ascii string. Successive calls to *build_trans* append to a string buffer, and end_trans sends it to the device. Register based controllers may map the primitive actions as individual register interactions. However, it is important that *build_trans* not perform any I/O operations, only *end_trans*. The record may omit a call to *end_trans* so that the next call to *start_trans* will clear the pending actions with no effect.

This can be accomplished by maintaining an in memory list of actions to be performed. The example below demonstrates a simple and effective way of implementing this.

The motor command codes are defined by the *motor_cmnd* enum in "motor.h". The value argument of *build_trans* is a double pointer. This can be an array, but is usually a pointer to a scalar value (position, velocity, . . .).

If a particular command is not meaningful for a controller it should be ignored.

5.1.1 Moves: MOVE_ABS, MOVE_REL

The motor record can use either relative and absolute move commands. Relative moves are used in the presence of an encoder. If the controller does not support both types of moves, then it is fairly simple for software to implement one in terms of the other.

Argument: 1

5.1.2 Homing: HOME_FOR, HOME_REV

Trigger a move in either direction which will stop when the home switch is activated.

Argument: 1 (but value always 0.0)

5.1.3 Set Position Counter: LOAD_POS

This command is used to manually set/reset the controller’s internal absolute position count (if applicable), and the RMP (raw motor position) field.

Argument: 1

5.1.4 Profiled Motion: SET_VEL_BASE, SET_VELOCITY, SET_ACCEL

Many controllers use a parametrized profile when moving. A controller can use whichever of these parameters is appropriate. If a controller supports a single velocity parameter then it must use *SET_VELOCITY* for this.

Argument: 1

5.1.5 Start Motion: GO

This command is the last issued as part of a transaction which will cause motion.

Argument: 0

5.1.6 Configuration: SET_RESOLUTION, SET_ENC_RATIO

TODO

Argument: 1

5.1.7 Request Update: GET_INFO

Sent by record support when it desires an immediate status update from device support. This may be called once at the end of the record processing.

Argument: 0

5.1.8 Immediate Stop: STOP_AXIS

Commands the controller to immediately stop all motion on this axis.

Argument: 0

5.1.9 Jogging: JOG, JOG_VELOCITY

Start a constant velocity move. The sign of the arguement to *JOG_VELOCITY* indicates direction.

Argument: 0 (JOG), 1 (JOG_VELOCITY)

5.1.10 PID parameters: SET_PGAIN, SET_IGAIN, SET_DGAIN

Can be implemented if the controller supports closed loop control.

Argument: 1

5.1.11 Closed Loop Control: ENABLE_TORQUE, DISABL_TORQUE

Used to enable/disable closed loop control.

Argument: 0

5.1.12 PRIMITIVE

Send a raw command string to the controller. No longer used by motor record.

Initialization and configuration actions should be implmented as IOC shell functions.

Argument: ?

5.1.13 Controller Soft Limits: SET_HIGH_LIMIT, SET_LOW_LIMIT

If the controller support software limits. Soft limits are completely independent of the hardware limit switches.

Argument: 1

5.1.14 Required Commands

At a minimum, device support must implement the following commands: MOVE_ABS, MOVE_REL, LOAD_POS, GO, and STOP_AXIS.

5.2 Status Updates

The *update_values* device support function is used by device support as a callback into device support. It is called by the motor record as soon as it is processed. The function may then update certain fields in the record, or not. If any fields is changed then *update_values* must return *CALLBACK_DATA*, or *NOTHING_DONE* otherwise. This allows the record to correctly detect which fields have been changed and to react accordingly (post monitors).

update_values allows the Motor record to poll device support for status updates when the record processes. However, since device support can (and should) trigger processing when it wants the record to be updated, this really allows for periodic and/or interrupt driven status updates from device support.

While there is no technical limitation on which fields can be changed by *update_values*, it is only safe to modify fields: *RMP*, *REP*, and *MSTA*.

5.2.1 RMP Raw Motor Position

This field is the absolute absolute position counter, the number of commanded steps. It is a signed 32-bit integer. Use of this field is required.

5.2.2 REP Raw Encoder Position

If an encoder readback is available it should be placed in this field, otherwise if should not be modified. This field is a signed 32-bit integer.

5.2.3 MSTA Motor Status

This field is a bit array for indicating controller status, some bits are required, while others are optional.

The *msta_field* union defined in *motor.h* must be used when filling this field.

```
typedef union
{
    unsigned long All;
    struct {
      unsigned int na                :17;/* N/A bits  */
      unsigned int RA_HOMED        :1; /* Axis has been homed.*/
        unsigned int RA_MINUS_LS     :1; /* minus limit switch has been hit */
        unsigned int CNTRL_COMM_ERR :1; /* Controller communication error. */
        unsigned int GAIN_SUPPORT    :1; /* Motor supports closed-loop position control. */
        unsigned int RA_MOVING       :1; /* non-zero velocity present */
        unsigned int RA_PROBLEM      :1; /* driver stopped polling */
        unsigned int EA_PRESENT      :1; /* encoder is present */
        unsigned int EA_HOME         :1; /* encoder home signal on */
        unsigned int EA_SLIP_STALL   :1; /* slip/stall detected */
        unsigned int EA_POSITION     :1; /* position maintenence enabled */
        unsigned int EA_SLIP         :1; /* encoder slip enabled */
        unsigned int RA_HOME         :1; /* The home signal is on */
        unsigned int RA_PLUS_LS      :1; /* plus limit switch has been hit */
        unsigned int RA_DONE         :1; /* a motion is complete */
        unsigned int RA_DIRECTION    :1; /* (last) 0=Negative, 1=Positive */d
    } Bits;
} msta_field;
```

Note: The above representation corresponds to a big-endian IOC. See "motorApp/MotorSrc/motor.h" for details.

The *RA_DONE*, *RA_DIRECTION*, *RA_PLUS_LS*, and *RA_MINUS_LS* bits will be used by all device support.

5.2.3.1 RA_DIRECTION

Used to compute the TDIR (direction of travel) field. Not used internally.

Value: 0 - Positive, 1 - Negative

5.2.3.2 RA_DONE

Used to compute the MOVN (moving) field. This field is used internally to indicate if the axis is in motion. This should be cleared when the axis begins moving, and set when it has stopped.

Must be initialized to 1.

Value: 0 - Moving, 1 - Stopped

5.2.3.3 RA_PLUS_LS

Indicates limit switch status in the raw positive direction.

Value: 0 - Normal, 1 - At limit

5.2.3.4 RA_HOME

Used to set the ATHM (at home) field if no encoder is present, or it is not used.

Value: 0 - Normal, 1 - At home

5.2.3.5 EA_SLIP

Not used.

5.2.3.6 EA_POSITION

Indicates status of closed loop control (position maintenance).

Value: 0 - Open loop, 1 - Closed loop

5.2.3.7 EA_SLIP_STALL

Used to signal an axis slip or stall, if supported by the controller. Causes a Major alarm on the record.

Value: 0 - Normal, 1 - Alarm

5.2.3.8 EA_HOME

Used to set the ATHM (at home) field if an encoder is present, and it is used.

Value: 0 - Normal, 1 - At home

5.2.3.9 EA_PRESENT

If the controller reports an encoder position in the REP (raw encoder position) field it must also set this flag.

Value: 0 - No encoder, 1 - Encoder present

5.2.3.10 RA_PROBLEM

Used to signal a failure effecting the motor. Causes a Major alarm on the record and resets axis status to stopped. If this flag is set, it is assumed that the controller has attempted to stop the axis.

Value: 0 - Normal, 1 - Alarm

5.2.3.11 RA_MOVING

Not used. (See RA_DONE).

5.2.3.12 GAIN_SUPPORT

Set to indicate support for closed loop control.

Value: 0 - Not supported, 1 - Closed loop available

5.2.3.13 CNTRL_COMM_ERR

Used to signal a failure in communications with the motor. Causes an Invalid alarm on the record.

Value: 0 - Normal, 1 - Alarm

5.2.3.14 RA_MINUS_LS

Indicates limit switch status in the raw negative direction.

Value: 0 - Normal, 1 - At limit

5.2.3.15 RA_HOMED

Not used.

6 Criticism

There are several problematic points in the design of the motorRecord which should be pointed out.

6.1 Multiple Coordinate Systems

The motor record contains fields with position values in no less then four coordinate systems: raw motor, raw encoder, dial, and user. This in turn necessitates three sets of parameters to control the coordinate transforms. At the same time it restricts the form these transforms to simple linear relations.

However, device support is only allowed to see that raw (integer) coordinates. This makes it difficult to create synthetic axes which appear as normal motors.

6.2 Processing Ambiguity

In order to be responsive to both internal (hardware) and external (channel access) events the motorRecord must process in unusual ways. To be responsive to user requests it must use passive processing. However, to provide updates during motion it must process periodically.

To accomplish both it assumes that device support will trigger processing internally. The result of the *update_values* device support function to defermine which condition caused processing. Different actions are taken for each case.

6.3 Device Support Interface

The motorRecord device support functions reflect its bidirectional nature. It has both write (*start_trans*, *build_trans*, and *end_trans*) as well as read (*update_values*) functions.

The transaction based interface of the **_trans* functions is a much more general interface then what is needed to impliment the transactions currently defined. Also, the (almost) unused abort transaction mechanism adds significant complexity when interactioning with register based devices.

6.4 One Record per Axis

Most of the inflexibility and strange behaviors of the motorRecord come from the decision to place all the logic for an axis in a single record. The decision has lead to an attempt to create a one-size-fits-all solution with little expandability in the cases (eg multi-axis motion) where it does not fit.

Additionally, providing access through fields instead of seperate records has lead to a sort of lock-in. Client software working with *rec.DVAL* and *rec.VAL* must always interact with the same record. It is not possible to use a calcRecord or other mechanisms in the process database to modify these values.

7 Debugging

The following points may be useful in debugging mis-behaving motor device support:

- To enable debugging print-messages add the following line to "motorApp/MotorSrc/Makefile":

```
USR_CPPFLAGS += =-DDEBUG
```

- When using the standard IOC shell add the following to the application's startup script:

```
var(motorRecordDebug,6)
```

- Those using the RTEMS Cexp shell will instead add the following:

```
motorRecordDebug=6
```

- Set the Motor Record's field, RTRY (number of retries), to zero. This will prevent the record-support from "hunting" for the correct position, should device support be misbehaving.
- Set the Motor Record's field, BDST (backlash distance) to zero. Initially this will only confuse the situation.

8 An Example

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include <dbDefs.h>
#include <ellLib.h>
#include <devSup.h>
#include <recSup.h>
#include <recGbl.h>
```

```
#include <callback.h>
#include <initHooks.h>
#include <dbAccess.h>
#include <epicsExport.h>
#include <cantProceed.h>
#include <devLib.h>
#include <iocsh.h>
#include <epicsTime.h>

#include <motorRecord.h>
#include <motor.h>
```

8.1 Data Structures

This example will use a simulated motor. It is a linear stage which moves at a constant rate with zero acceleration time. The simulated "controller" implements an absolute position counter, and positive and negative limit switches. However, it was no home switch or encoder, and does not support additional soft limits.

The position of the limit switches with respect to the motorRecord soft limits is arbitrary. However, normal operation the record soft limits would be inside the bounds of the limit switches.

8.1.1 Simulation State

The *hardware* structure is the state of this simulation. It must hold the motor's settings and its current state. State includes the motor's current position and the position of the hard limits. These are distinct from what is represented in the motorRecord itself. It also holds the number of steps to move (signed), and the velocity.

In order to simplify this example the motor simulation will not run concurrently. Its state will instead be updated when the motorRecord polls its state.

```
struct hardware {
  epicsInt32 pos; /* signed */
  double vel;

  epicsInt32 lim_h_val;
  epicsInt32 lim_l_val;
  int lim_h:1;
  int lim_l:1;

  int moving:1;
  epicsInt32 remaining;

  epicsTimeStamp last;
};
```

Both counts remaining and velocity will have the same sign.

8.1.2 Transactions

The motorRecord sends commands to the axis as transactions. These are sets of primitive commands assembled and committed (sent) in groups. Historically these were ascii strings (ie *VEL4.2;ACC0.2;REL4000;GO*).

Since our simulated motor does not deal in ascii strings we will instead keep a list of actions to be performed when the transaction is committed. This will simply be a list of actions (function pointers) values for them.

Currently the motorRecord does not use actions with more then two arguments.

```
enum {max_targs=2};

typedef void (*trans_proc_t)();

/* List entry to hold transactions */
struct trans {
  ELLNODE node; /* Must be first */
  size_t nargs;
  double args[max_targs];

  trans_proc_t trans_proc;
};
```

8.1.3 Device Support Private

Now we come to the device support private structure for motorRecord.

```
struct devsim {
  ELLNODE node;

  struct hardware hw;
```

A IOC global list of device support instances will be needed later. Also the "hidden" state for the simulation.

```
  int id;
```

A IOC global unique identifier marking this instance of the device support. Used to match an instance created with a IOCsh function to a record.

```
  double rate; /* poll rate */

  int updateReady;
  CALLBACK updatecb;
```

A callback used to implement the internal polling of device state. This will be timer running at a fixed rate.

```
  ELLLIST transaction; /* list of struct trans */
};
```

The list of uncommitted transactions for this motor.

8.2 Simulation

When the simulation is running we will periodically advance the state.

```
static
void update_motor(struct hardware *hw)
{
  epicsTimeStamp now;
  double ellapsed, moved;

  if(hw->moving){

    epicsTimeGetCurrent(&now);

    ellapsed = epicsTimeDiffInSeconds(&now, &hw->last);

    moved = ellapsed * hw->vel;
```

By looking at the time since the last update, determine how far the motor has moved.

```
    if( fabs(moved) >= fabs(hw->remaining) ){
      moved = hw->remaining;
      hw->moving = 0;
    }
```

The simulated motor is well behaved. It will always stop after moving exactly the requested number of steps.

```
    hw->pos += (epicsInt32)moved;
    hw->remaining -= (epicsInt32)moved;
  }

  if(hw->pos >= hw->lim_h_val) {
    hw->pos = hw->lim_h_val;
    hw->lim_h=1;
  }else
    hw->lim_h=0;

  if(hw->pos <= hw->lim_l_val) {
    hw->pos = hw->lim_l_val;
    hw->lim_l=1;
  }else
    hw->lim_l=0;
}
```

Update the internal position and limit state. Clip the position if it would haved exceed the limits.

8.3 Convenience

A helper function for searching for device support instances by global id number.

```
static
ELLLIST devices = {{NULL,NULL},0}; /* list of struct devsim */

static
struct devsim *getDev(int id)
{
  ELLNODE *node;
  struct devsim *cur;

  for(node=ellFirst(&devices); node; node=ellNext(node))
  {
    cur=(struct devsim*)node; /* Use CONTAINER() in 3.14.11 */
    if(cur->id==id)
      return cur;
  }
  return NULL;
}
```

8.4 Initialization

Device support instances will be created from the IOCsh. Here we specify the real positions of the hard limit switches.

In a real world example we would also give the hardware address of the controller.

```
static
void timercb(CALLBACK* cb);

static
```

```
void addmsim(int id, int llim, int hlim, double rate)
{
        struct devsim *priv=getDev(id);

        if(!!priv){
                printf("Id already in use\n");
                return;
        }

        priv=calloc(1,sizeof(struct devsim));

        if(!priv){
                printf("Allocation failed\n");
                return;
        }

        priv->id=id;

        callbackSetCallback(timercb, &priv->updatecb);
        callbackSetPriority(priorityHigh, &priv->updatecb);

        priv->rate=rate;

        priv->hw.lim_h_val=hlim;
        priv->hw.lim_l_val=llim;

        ellAdd(&devices, &priv->node);
}
```

8.4.1 init_record

Here we match the record and device support instances by matching the number given to the IOCsh function with the record OUT link.

```
static
long init_record(motorRecord *pmr)
{
  struct motor_dset *dset=(struct motor_dset*)(pmr->dset);
  msta_field stat;
  double val;
  long ret=0;
  struct devsim *priv=getDev(pmr->out.value.vmeio.card);

  if(!priv){
    printf("Invalid id %d\n",pmr->out.value.vmeio.card);
    ret=S_dev_noDevice;
    goto error;
  }
  pmr->dpvt=priv;

  callbackSetUser(pmr, &priv->updatecb);
```

Here we come upon the first of several quirks of the motorRecord.

If the RA_DONE flag is not initialized to 1 then the motor will immediately go into a moving state which can only be exited by a stop command.

```
  stat.All=0;

  stat.Bits.RA_DONE=1;
  pmr->msta=stat.All;
```

The motor position stored by the autosave module is the DVAL field. At start we must explicitly convert this to raw counts and set the controller.

```
  if( pmr->mres !=0.0 ){
    /* Restore position counter.  Needed for autosave */
    val = pmr->dval / pmr->mres;
    (*dset->start_trans)(pmr);
    (*dset->build_trans)(LOAD_POS, &val, pmr);
    (*dset->end_trans)(pmr);
  }

  return 0;
error:
  pmr->pact=TRUE;
  return ret;
}
```

8.4.2 Start Polling

Notice that the periodic update timer was not started in *init_record*. This is because the callback will cause record processing, but this will fail unless *iocInit()* has completed.

To avoid this we use the IOC initialization hooks to start the timers as iocInit finishes. Note that the *initHookAtEnd* hook is called after autosave has completed as well.

```
static
void inithooks(initHookState state)
{
        ELLNODE *node;
        struct devsim *cur;
        /* as of 3.14.11 initHookAtEnd is deprecated and initHookAfterIocRunning is proper  ↩
            */
        if(state!=initHookAtEnd)
                return;

        for(node=ellFirst(&devices); node; node=ellNext(node))
        {
                cur=(struct devsim*)node;

                callbackRequestDelayed(&cur->updatecb, 1.0/cur->rate);
        }
}
```

8.5 Poll Motor State

Polling the motor's state happens in several parts. It may be caused by the periodic timer, or by a hardware interrupt. In either case the update flag will be set and then the record will process.

8.5.1 Update Callback

The period timer does three things. First it rearms itself, second it sets the *updateReady* flag, then it processes the record.

The *updateReady* flag exists to distinguish the two conditions which could cause the motorRecord to process. If it is not set then processing is the result of a user action. If it is set then processing is the result of a change in the hardware state.

```
static
void timercb(CALLBACK* cb)
{
        motorRecord *pmr=NULL;
        struct rset* rset=NULL;
        struct devsim *priv=NULL;

        callbackGetUser(pmr,cb);
        priv=pmr->dpvt;
        rset=(struct rset*)pmr->rset;

        dbScanLock((dbCommon*)pmr);

        callbackRequestDelayed(&priv->updatecb, 1.0/priv->rate);

        update_motor(&priv->hw);
        priv->updateReady=1;

        (*rset->process)(pmr);

        dbScanUnlock((dbCommon*)pmr);
}
```

Note that *update_motor* is called here so that it can be guarded by the record lock. This allows us to avoid threading issues.

8.5.2 update_values

When the motorRecord is processed the first thing which occurs is a call to the *update_values* device support function. This is a test by the record to determine why it was processed. It also gives the record the opportunity to save a copy of the fields which can be changed so that monitors can be posted if a change does in fact occur.

The only fields which can be updated are *RMP*, *REP*, and *MSTA*.

This function exists to transfer data from the hardware into the record.

```
static
CALLBACK_VALUE update_values(motorRecord *pmr)
{
        struct devsim *priv=pmr->dpvt;
        msta_field modsts;

        if(!priv->updateReady)
                return NOTHING_DONE;
        priv->updateReady=0;

        modsts.All=pmr->msta;

        modsts.Bits.RA_PLUS_LS = priv->hw.lim_h;
        modsts.Bits.RA_MINUS_LS = priv->hw.lim_l;
        modsts.Bits.RA_DONE = !priv->hw.moving;

        pmr->msta=modsts.All;

        pmr->rmp = (double)priv->hw.pos;

        return CALLBACK_DATA;
}
```

8.6 Command Transactions

Commands are built and sent to the device through three functions: start, build, and end transaction.

8.6.1 start_trans

This function begins a new transaction by aborting any previous uncommitted transaction.

```
static
long start_trans(struct motorRecord *pmr)
{
        struct devsim *priv=pmr->dpvt;
        ELLNODE *node, *next;
        struct trans *cur;

        for(node=ellFirst(&priv->transaction), next = node ? ellNext(node) : NULL;
                node;
                node=next, next=next ? ellNext(next) : NULL )
        {
                cur=(struct trans*)node;

                ellDelete(&priv->transaction,node);
                free(cur);
        }

        return 0;
}
```

8.6.2 Actions

Now we will look at some of the primitive actions which can go into a transaction.

The best (only) way to gain an understanding of what groups of primitives are actually used is to review the source for the motorRecord record support.

8.6.2.1 Position

A relative move is commanded if the axis has an encoder present. Otherwise absolute moves are used. It is always necessary to implement both move types.

```
static
void move_rel(motorRecord *pmr, double rel)
{
        struct devsim *priv=pmr->dpvt;

        priv->hw.remaining = (epicsInt32)rel;
}
```

Implement an absolute move in terms of a relative move.

```
static
void move_abs(motorRecord *pmr, double newpos)
{
        struct devsim *priv=pmr->dpvt;

        newpos -= (double)priv->hw.pos;

        move_rel(pmr, newpos);
}
```

Restore the internal position counter from an external source.

```
static
void load_pos(motorRecord *pmr, double newpos)
{
        struct devsim *priv=pmr->dpvt;

        priv->hw.pos = (epicsInt32)newpos;
}
```

8.6.2.2 Velocity

Set the move velocity. The motion profile used by the motorRecord has two velocities, we need only one and chose to ignore the base velocity.

```
static
void set_velocity(motorRecord *pmr, double vel)
{
        struct devsim *priv=pmr->dpvt;

        priv->hw.vel = vel;
}
```

8.6.2.3 Immediate

This both commands motion, and notes the time for use by the simulation.

```
static
void go(motorRecord *pmr)
{
        struct devsim *priv=pmr->dpvt;

        /* Simulation start */

        if(!priv->hw.remaining)
                return;

        if(!priv->hw.vel)
                return;

        /* make the sign of the velocity match remaining */
        priv->hw.vel = copysign(priv->hw.vel, priv->hw.remaining);

        priv->hw.moving=1;
        epicsTimeGetCurrent(&priv->hw.last);

        /* update record */
        callbackRequest(&priv->updatecb);
}
```

Cause an immediate halt.

```
static
void stop(motorRecord *pmr)
{
        struct devsim *priv=pmr->dpvt;

        /* Simulation stop */

        if(!priv->hw.moving)
```

```
                return;

        priv->hw.moving=0;
        priv->hw.remaining=0;

        /* update record */
        callbackRequest(&priv->updatecb);
}
```

8.6.3 build_trans

Stores a transaction in the device support transaction list.

```
static
RTN_STATUS build_trans(motor_cmnd cmd, double *val, struct motorRecord *pmr)
{
        struct devsim *priv=pmr->dpvt;
        struct trans *t=callocMustSucceed(1,sizeof(struct trans), "build_trans");

        switch(cmd){
        case MOVE_ABS:
                t->nargs=1;
                t->args[0]=*val;
                t->trans_proc=&move_abs;
                break;
        case MOVE_REL:
                t->nargs=1;
                t->args[0]=*val;
                t->trans_proc=&move_rel;
                break;
        case LOAD_POS:
                t->nargs=1;
                t->args[0]=*val;
                t->trans_proc=&load_pos;
                break;
        case SET_VELOCITY:
                t->nargs=1;
                t->args[0]=*val;
                t->trans_proc=&set_velocity;
                break;
        case GO:
                t->nargs=0;
                t->trans_proc=&go;
                break;
        case STOP_AXIS:
                t->nargs=0;
                t->trans_proc=&stop;
                break;
        case SET_HIGH_LIMIT:
        case SET_LOW_LIMIT:
                /* TODO */
        case SET_VEL_BASE:
        case GET_INFO:
                free(t);
                return OK;
        default:
                printf("Unknown command %d\n",cmd);
                free(t);
                return OK;
        }
        ellAdd(&priv->transaction, &t->node);

        return OK;
}
```

8.6.4 end_trans

Here we simply iterate through the list of transaction and execute the actions.

```
static
RTN_STATUS end_trans(struct motorRecord *pmr)
{
        struct devsim *priv=pmr->dpvt;
        ELLNODE *node, *next;
        struct trans *cur;

        for(node=ellFirst(&priv->transaction), next = node ? ellNext(node) : NULL;
                node;
                node=next, next= next ? ellNext(next) : NULL )
        {
                cur=(struct trans*)node;

                switch(cur->nargs){
                case 0:
                        (*cur->trans_proc)(pmr);
                        break;
                case 1:
                        (*cur->trans_proc)(pmr,
                                cur->args[0]);
                        break;
                case 2:
                        (*cur->trans_proc)(pmr,
                                cur->args[0],
                                cur->args[1]);
                default:
                        printf("Internal Logic error: too many arguments\n");
                        return ERROR;
                }

                ellDelete(&priv->transaction, node);
                free(cur);
        }

        return OK;
}
```

8.7 Boiler Plate

Standard code export the IOC shell function and device support.

```
struct motor_dset devMSIM = {
        {
         8,
         NULL, /* report */
         NULL, /* init */
         (DEVSUPFUN) init_record,
         NULL /* get_ioint_info */
        },
        update_values,
        start_trans,
        build_trans,
        end_trans
};
epicsExportAddress(dset, devMSIM);

static const iocshArg addmsimArg0 = { "id#", iocshArgInt };
static const iocshArg addmsimArg1 = { "Low limit", iocshArgInt };
static const iocshArg addmsimArg2 = { "High limit", iocshArgInt };
static const iocshArg addmsimArg3 = { "Update Rate", iocshArgDouble };
static const iocshArg * const addmsimArgs[4] =
{ &addmsimArg0, &addmsimArg1, &addmsimArg2, &addmsimArg3 };
static const iocshFuncDef addmsimFuncDef =
{ "addmsim", 4, addmsimArgs };
static void addmsimCallFunc(const iocshArgBuf *args)
{
  addmsim(args[0].ival,args[1].ival,args[2].ival,args[3].dval);
}

static
void msimreg(void)
{
        initHookRegister(&inithooks);
        iocshRegister(&addmsimFuncDef, addmsimCallFunc);
}
epicsExportRegistrar(msimreg);
```

Also the database definition.

```
variable(motorRecordDebug)
device(motor, VME_IO, devMSIM, "Moter Simple Sim")
registrar(msimreg)
```