Adding a New Stage to the LRIS Motor Control Software

John Cromer

August 3, 1993

Contents

0.1 Introduction

In the not-too-distant future, more moveable stages such as the polarimeter and the blue camera with its focus and filter mechanisms, will be added to the LRIS. This paper describes the implications to the software of adding such additional stages, and the necessary modifications to make the stages work. New software for both the Unix workstation and the VxWorks VME system must be added. In most cases, the addition of data to a table and the cloning of already exsisting functions will be sufficient to integrate the new stage into the existing software system. A broad outline of what must be done is as follows:

1. Create the workstation software: Add keywords and MUSIC message numbers to the appropriate tables and create the FIORD functions.
2. Create the VME software: create the "set" and "show" functions and add the stage entries to the global tables.
3. Calibrate the stage: Determine stage parameters and update global tables. Update unit conversion tables on the Unix side.
4. Test the stage
5. Add sections to the relevant documentation.

These steps will be amplified in the following sections and examples will be given in the final section. This manual does not include CCD-specific software. See *Adding an additional CCD Camera to the LRIS Software* for adding another CCD camera to the LRIS.

Certain assumptions are made in this manual about the reader's understanding of the LRIS. As an LRIS enthusiast you should know how to distinguish among the different "types" of stages–discrete, limit, blind and encoder. You should also understand the basic organization of the software into "host" (Unix workstation) and "target" (VME system) partitions. For a discussion of these and other fascinating topics, see the *FIORD Software for LRIS Motor Control* and *VME Software for LRIS Motor Control.*

0.2 Adding the Stage to the FIORD Software

Create the FIORD software in the following steps:

1. Define a keyword to be used by `show` and `modify` to control the new stage. The new keyword should be added to the keyword definitions document. A long version (greater than eight characters) should be defined as well as a short version (eight characters or less) for FITS compatibility.
2. Update the file `/kroot/kss/lris/fiord/include/lris_msgs.h`. In the first section, increase the value of `MAX_STAGES` if you are adding a new stage, the value of `MAX_ENCODERS` if you are adding and encoder also, or the value of `MAX_LAMPS` if you

are adding a new reference lamp. In the next section of the file add an new `#define` for the stage ID. And in the final section add four new MUSIC message numbers, two to define a command to move the stage and two to show the current position of the stage. LRIS motor control messages are numbered from 3500-4000.

3. In the directory `/kroot/kss/lris/fiord`, update the files `fiord_proto.h` with the new function prototype `#defines`, and `fiord.c` with the new FIORD table entry. The easiest way to do this is to copy an exisiting entry and modify it for the new keyword. Note, for the LRIS, input and output functions are named `input_`*keyword* and `output_`*keyword*.

4. In the directory `/kroot/kss/lris/fiord/lris` create the file `f_`*keyword*`.c` containing the two new FIORD functions `input_`*keyword* and `output_`*keyword*. These routines can easily be created by copying existing functions and makeing a few simple changes. For discrete stages, you should use functions for a discrete stage such as the grating turret or red filter selector. For an encoder stage, use functions for an encoder stages such as the red camera focus mechanism. All you will need to change are the macros defining the message numbers and the stage ID.

   For encoder stages, you will have to define the units conversion in the FIORD function module. This may not be possible until the stage is finished and on the instrument so that parameters such as scale and offset can be determined experimentally. Update the values when the information is available.

5. Update `/kroot/kss/lris/fiord/lris/makefile` to add the new functions to the list therein. Typing `make` in this directory will create the FIORD library and can serve as a compilation test. Move up a directory level to the `/kroot/kss/lris/fiord` directory and type `make` to create the complete, shared, keyword library. After this, `show` and `modify` should work with the new keywords.

0.3 Adding the stage to the VME system

Use the following procedures to add the new stage to the VxWorks software that runs on the VME system:

1. Decide which "move" function (found in `lrs_move.c`) is appropriate to the new stage. The choices are:

   (a) `move_int_simple()` for simple discrete and limit stages

   (b) `move_float_simple()` for simple encoder stages

   (c) `move_compound()` for compound stages

   (d) `move_int_blind()` for discrete stages with no position feedback.

   See *VME Software for LRIS Motor Control* for an explanation of the different types of stages. Create a custom "move" function if none of the above functions can be used.

2. Create a file named `s_keyword.c` containing the functions `s_set_keyword()` and `s_show_keyword()`. Use a similar existing file as a template.

3. Update the table in file `server_table.c`. This table links the "set" and "show" functions with the appropriate MUSIC message IDs. One entry per function will be required.

4. Edit the file `load_lserv` to ensure that the new VxWorks file will get loaded on the VME system at startup.

5. Edit the makefile to add the new VxWorks object file, its file dependencies and the appropriate lines for compiling the source.

6. Edit `lrs_global_init.c` to add a stage table entry for each new stage, a encoder entry for each new encoder, and an entry in `current_positions[]` for each new stage. The most important table is the stage information table. The information for this table comes basically from two sources: the hardware engineer, and your own experimentation.

   The hardware engineer should be able to give you the following information:

   (a) The API unit number of the controller that moves the stage.

   (b) The API unit number of the controller whose input bits encode each position number (discrete stages only).

   (c) The API S parameter, the power-off delay time

   (d) The API ID1 parameter, the default state of the input bits.

   (e) The input bits that encode the stage's positions (discrete stages only).

   (f) The API output bit sequence that selects the stage's motor

   (g) The total number of positions the stage has (non-encoder stages only).

   (h) The Stage ID number of an auxiliary stage (compound stages only).

   (i) The Encoder ID number of an attached encoder (encoder stages only).

   (j) The position table, the position switch reading for each position 1 through the maximum number of positions (discrete stages only).

   It will probably be necessary for you to manually experiment with the stage to determine the following information:

   (a) The API velocity profile: A, the acceleration, D the deceleration, B the base speed, and M and H, the maximum speeds.

   (b) The API command sequence to home the stage.

   (c) The API command sequence to move the stage.

   (d) The "standard displacement", the number of motor steps between two adjacent positions (non-encoder stages only).

(e) The minimum motor position (encoder stages only).

(f) The maximum motor position (encoder stages only).

Adding the new stage information to the global table in `lrs_global_init.c` is critical to the correct operation of the stage. Therefore, the structure is shown here and each member is described below.

```
typedef struct {
    char   *name;               /* stage name */
    int    move_mc;             /* controller for move */
    int    stat_mc;             /* controller for status read */
    char   *ibits;              /* substring of iword for status */
    char   *select_seq;         /* output seq that selects this motor */
    char   *home_1_seq;         /* 1st home move */
    char   *home_2_seq;         /* 2nd home move */
    char   *config_seq;         /* api configuration sequence */
    char   *move_seq;           /* api move sequence */
    int    std_disp;            /* standard displacement between pos.*/
    int    max_positions;       /* max.  number of items in mechanism */
    int    min_motor_position;  /* min.  position in motor steps */
    int    max_motor_position;  /* max.  position in motor steps */
    int    aux_stage;           /* another associated stage id */
    int    encoder;             /* abs encoder id (if one exists) */
    int    pos_no_codes[MAX_PCODES];
                                /* position table, relating real */
                                /* stage positions to those */
                                /* reported by switch readings */
    } STAGE;
```

`name` The name of the stage. Examples are `grating_turret`, `grating_turret_detent`, `blue_filter_selector`. In general, spaces inside the name are avoided. Using spaces may not be fatal, however the stage name is used as a file name when the stage's status is stored on disk.

`move_mc`. The unit number of the API motor controller which is connected to the stage's motor. This unit number must be selected with the API `:` command before the stage can be moved.

`stat_mc`. The unit number of the API controller to which the stages status lines are connected. This unit number must be switched selected with the `:` command before the stage position can be read.

`ibits`. An ASCII string containing the bits used to encode a stage's position. The positions of discrete stages are given by switches connected to the input bits of API controllers. Each

controller has two 8-bit input words designated `I1` and `I2`. Bits 1–8 make up `I1`. Bits 9–16 make up `I2`. (In the API documentation, bits in an input word are number from 1 from the left as the word is displayed with the `verify I1` command or the `status` command.) The `ibits` structure member tells the software which bits encode the stage's position. For example, the sequence `0000011110000000` indicates bits 6, 7, 8 and 9 give the position of the stage, in this case the red filter selector. Note this is bits 6, 7 and 8 on `I1` and bit 1 on `I2`. The software always reads both input words when a stage position is requested.

`select_seq`. An ASCII string containing the output bits used to select the stage's motor. Each controller controls 1 or more motors, selected by a motor multiplexor. The controller's output bits are used to switch the multiplexor and selected the desired motor. Each controller has one 8-bit output word designated `O1`. In API terminology, bits are numbered from the left beginning with 1. Different bits are set to selected different motors, and these "addresses" are encoded in the `select_seq`. For example, the select sequence `00001101` indicates bits 5, 6 and 8 should be set to select this stage, in this case, the red camera focus motor. The API command `SET 5 6 8` would be used to select this motor.

`home_1_seq`. The controller command sequence used to home the stage. The sequence, when sent to the controller would move the stage to its home, or reference, position.

`home_2_seq`. The second controller command sequence used to home the stage. At one point in the development of this project, it was said that the grating tilt mechanism would require two separate command sequences to home it. This turned out not to be the case. But here it is anyway. And it's in the code, too. Go figure.

`config_seq`. The API command sequence that configures a controller for this stage. The command sequence can be any legal combination of API commands. It is generally used to set the motor motion profile (accelerations, base and maximum velocities, decelerations, etc), the default input bit states, the "power-down" time. A typical sequence is `"A=2000,D=6000,B=5000,H=34000,M=34000,ID2=11111111,S=100\r \n"`.

`std_disp`. The stage's standard displacement. A better name for this might be unit displacement. It is the distance in motor steps required to move a discrete stage one position. The software moves discrete stages in multiples of this number to get from one position to another. An example is, 206475, the number of motor steps required to move the red filter selector one position. In general the number is equal to the full range of travel divided by the maximum number of positions minus one, ie $std\_disp = full\_travel/(max\_positions - 1)$.

`max_positions`. The stage's total number of discrete positions. This applies to non-encoder stages only. The value of this parameter for the slitmask selector is 10, for example. This number is 0 for encoder stages, 2 for limit stages.

`min_motor_position`. The minimum motor (or encoder) position valid for a stage. The software will reject any motor move whose destination position is less than this number. This check is performed on encoder stages only.

`max_motor_position`. The maximum motor (or encoder) position valid for a stage. The software will reject any motor move whose destination position is greater than this number. This check is performed on encode stages only.

`aux_stage`. A compound stage's auxillary stage. Used only by compound stages. The value should be -1 for any simple stage. This is the number (index position in this array of stage structures) of the stage that is moved before and after a move of the present stage. For example, the auxillary stage of the grating turret is the turret detent stage, number 1. Likewise stage 2, the grating brake, is the auxillary stage for the grating tilt mechanism. A non-negative number in this field indicates the stage is a compound one. An entry of -1 indicates no auxillary stage and thus the stage is a simple one.

`encoder`. The encoder number. This is the number (index position in the encoder table) of an encoder that provides position feedback for this stage. Example: encoder number 2 is the encoder attached to the offset guider M2 motor, stage 11. See the `lrs_global_init.c` file where these tables are initialized.

`pos_no_codes`. An integer array of numbers which are the switch codes corresponding to the discrete positions of a stage. This is basically a lookup table for converting switch readings to stage position numbers. Switch readings are read from the input bits of the stage's status controller. For example, the red filter selector stage's entries are 0, 15, 11, 9, 13, 12 and 8. When the stage is at position 2, the switch reading should be 11. At position 5, the API input bits when decoded should give a 12. Note these arrays are indexed from 0, and user positions are designated beginning with 1, so the zeroth value of each of these arrays is never used. The maximum number of array entries is defined in `stagestruct.h` as `MAX_PCODES`. This array is not used for blind, limit or encoder stages.

Information and operating parameters for the various encoders are similarly defined in the ENCODER structure. An array of these structures is defined and initialized in `lrs_global_init.c`. Each encoder on the LRIS has an entry in this array of structures. Each structure entry has the following form as defined in `stagestruct.h`:

```
typedef struct {
    char   *type;           /* encoder controller type "AX" or "API" */
    char   *dca;            /* encoder daisy chain address */
    int    etol;            /* move error tolerance */
    int    max_tries;       /*max times to try to reach etol */
    int    mspr;            /* motor steps per shaft revolution */
    int    espr;            /* encoder steps per shaft revolution */
    char   *init_seq;       /* encoder initialization sequence */
    char   *read_seq;       /* encoder read sequence */
    } ENCODER;
```

Each of the structure members is described below.

`type`. An ASCII string encoding the type of encoder, "API" indicating an incremental encoder read through an API controller, or "AX", indicating a Compumotor absolute encoder. The astute student of the LRIS documentation will notice in the Compumotor manuals and hardware descriptions, the absolute encoders are referred to as "AR" encoders. Why then,

the "AX" designation in the software? Why, indeed? The original LRIS motor controllers were Compumotor AX controllers. The "AX" designation was used in the original version of the software and has remained even in the designation of encoders. Programmer braindeath.

`dca`. An ASCII string containing the numerical address of the encoder on the serial daisy chain.

`etol`. The error tolerance for a move using this encoder in encoder steps. All encoder moves are implemented as servo loops. The servo loop will iterate until the difference between the destination position and the current position is less than this number or until the loop times out. See `max_tries` also.

`max_tries`. The maximum number of times to execute a servo loop before it times out. All encoder moves are implemented as servo loops. The servo loop will continue to iterate as long as the loop counter is less than or equal to this number and the difference between the destination position and the current position is greater than `etol`. See `etol`.

`mspr`. Motor steps per revolution. This number is the total number of microsteps per shaft revolution of the motor to which this encoder is connected. In general, the number is 12800.

`espr`. Encoder steps per revolution. This number is the total number of steps per shaft revolution of the encoder. For Compumotor AR encoders in decimal mode, this number is 10000.

`init_seq`. The encoder initialization sequence. This is an ASCII character sequence which is used to initialize an encoder.

`read_seq`. The encoder read sequence. This is the ASCII character sequence that must be sent to an encoder (or motor controller) to read the encoder's current position. For the absolute encoders, the sequence is `nPR` where `n` is the daisy chain address of the encoder. For incremental encoders, the sequence is `VER E` which the dedicated LRIS enthusiast will recognize as the API verify command requesting the value of the `E` parameter.

0.4 Calibrating the New Stage

It is necessary to detemine the operating parameters of the stage so that the remaining software data areas can be updated with the correct information. This can be done by moving the stage directly using the API controller alone without either workstation or VME crate software. A terminal can be connected directly via a serial link to the LRIS communications multiplexor or to the API box itself. Alternatively, the VxWorks program `api` can be run on the VME console and will simulate a direct connection to the API controller daisy chain.

0.4.1 All Stages

Generally it is a good idea to first move the stage slowly and in small increments to make sure there are no interferences or obstructions.

For all stages the velocity profile and the home command must be determined. Determining the velocity profile is a matter of art and taste, as well as choosing parameters that minimize wear and tear on the motor. Try different values for the base and maximum speeds. Choose the combination that yields the quietest running motor and completes a normal move in a suitable amount of time.

The home command will either be `"-HOM,E=0\r\n"` or `"PAUSE,MOV -`n`\r\n"` where n is a number slighly larger than the stage's full travel. The first command is used to home stages that have incremental encoders which are connected to an API unit, such as the grating tilt mechanism. The second home command above is used for every other type of stage on the LRIS. It is simply a move to the negative limit of the stage.

0.4.2 Limit Stages

Limit Stages are essentially discrete stages with only two positions. The full travel distance is equal to the standard displacement. To determine this distance, move the stage in the negative direction a large number of steps until it his its negative limit. Set the API `P` parameter to zero. Then drive the stage to its positive limit. Read the value of the `P` parameter. This is the stages's full travel in motor steps. The standard displacement of a limit stage should be set to a number a few thousand steps larger than the full travel.

0.4.3 Discrete and Blind Stages

For discrete and blind stages, do the following:

1. Use the procedure described above for limit stages to determine the full travel displacement in motor steps.
2. Divide the full travel displacement by one less than the stage's total number of discrete positions to get the standard displacement, the number of motor steps between each consecutive position.

0.4.4 Encoder Stages

Use the following procedure to calibrate an absolute encoder stage:

1. Drive the stage to its negative limit and read the encoder. This is the value for the `min_motor_position` in the stage structure previously described.
2. Drive the stage to its positive limit and read the encoder. This is the value for the `max_motor_position`.
3. Drive the stage a known number of encoder steps and use a dial indicator to measure the actual displacement in user units. This gives you the scale factor in, for example, encoder steps per micron or millimeter or furlong.

4. Define a zero point and note the encoder value at that point. This is the offset factor. This along with the scale factor is used in the fiord routines to convert from user units to encoder position.

A stage that uses an incremental encoder instead of an absolute encoder is calibrated using the above procedure also, however, the offset factor is usually zero. Since the stage and encoder must be "homed," the encoder is automatically set to zero at that point. This is usually the zero point for user units also, but does not have to be.

0.5 Testing the Stage and Its Software

Testing can be broken down into three general areas:

1. VME system only tests.
2. FIORD/VME communications and comversions tests.
3. Fully integrtaed and burn-in tests.

VME system-only tests tend to overlap with the experimental procedures described previously for determining the stage's operating parameters. Use the `api` and `ax` programs to communicate with the controllers and encoders directly, or connect a terminal (or a PC running a terminal emulation program) to the controllers. Move the stage small displacements checking for interferences and obstructions. Move the stage to each of its limits of travel, checking API parameter `I2` to make sure the limit switches work correctly. Test any lock-out mechanisms. For example, on the slitmask mechanism, an open door will set both limit switches on the controller, effectively disabling the motor. Removing the key which lines up the slitmasks has the same effect. Move the stage known distances and physically measure the stage's movement to check calibration factors. For discrete stages, move the mechanism to each individual position and check the switch reading for validity. Make sure these readings agree with the lookup table stored in the `pos_no_codes[]` array in the stage information table. Finally execute the "move" function that will be called to move this stage, directly from the VxWorks console, or write a small, one-line, C program to call it. Make sure the stage responds correctly to this function.

Compile and link the VxWorks functions in `s_`*keyword*`.c` with the `-DSIMULATE` flag active. This allows these functions to be spawned by `cserv` and to execute without actually moving the motor. Now execute the `modify` and `show` commands on the host system and check the VxWorks console to see if the appropriate functions actually were spawned. The `s_set_keyword()` function should display the destination position. Make sure it agrees with the position desired. If this is an encoder position do a hand calculation using the units-conversion formulae in the FIORD output function and see if the value displayed on the VxWorks console agrees with the hand-calculated value. Does the value seem reasonable given the desired move distance? Try the `modify` command with illegal destination values. Make sure they are rejected cleanly and clearly.

When you are satisfied the stage is in working condition, the VxWorks software is correct, the network link and MUSIC messages are correct and the FIORD functions are working properly, recompile the VxWorks stage functions without the `-DSIMULATE` flag. And attempt to move the stage with the `modify` command from the workstation. Use the `show` command to read the stage position. Do this for every individual position of discrete, blind and limit stage. Verify encoder stages over the legal length of travel. "Burn in" the software by creating a shell file which uses the `modify` and `show` commands to move the stage repeatedly over an extended period of time. Note any errors—especially limit switch failures, which seem to be the most common.

0.6 Updating the Documentation

Notes about the stage and especially about any idiosyncracies concerning the stage should be added to the appropriate manuals:

1. *The LRIS Motor Control Troubleshooting Guide.*
2. *VME Software for LRIS Motor Control.*
3. *FIORD Software for LRIS Motor Control.*

The manual generated from header files via `wflman` should be supplimented with the new function headers.

0.7 Example: Adding the Blue Camera Focus Mechanism

0.7.1 FIORD Software

Start first with the workstation side of the software and update the FIORD system. Invent a keyword to be used with `show` and `modify`: BLUFOCUS, and proceed to update the FIORD software.

In `fiord.c` make a copy of the table entry for the red camera focus and simply change "red" to "blue." The new entry should look like this:

```
{ "blue-camera-focus",
        "BLUFOCUS",    _float,         input_blufocus, output_blufocus,
                                       { LMOT_SERV, LINF_SERV, -1},
                                       { " microns "},1,
                                       { &blufocus_kval_info,
                                       &blufocus_response_kval_info, NULL}}
```

Next add the FIORD function prototypes to the `fiord_proto.h` file:

```
extern int input_blufocus();
extern int output_blufocus():
```

Now update the `lris_msgs.h` file. Since the focus is an encoder stage you will need to increment `MAX_STAGES` and `MAX_ENCODERS`:

```
#define MAX_STAGES 19
#define MAX_ENCODERS 4
```

Define a stage name (similar to the red camera focus):

```
#define LRS_BLUE_CAMERA_FOCUS 18
```

Define the MUSIC message numbers:

```
#define LRS_SET_BLUFOCUS 3566
#define R_LRS_SET_BLUFOCUS 3567
#define LRS_SHOW_BLUFOCUS 3666
#define R_LRS_SHOW_BLUFOCUS 3666
```

And finally define a broadcast for this message if desired. This should be done since you included `kval` broadcast response functions in our table in `fiord.c`:

```
#define B_LRS_BLUFOCUS 652
```

This concludes the update of `lris_msgs.h`. Now the actual FIORD functions must be created. Make a copy of `f_redfocus.c` and call it `f_blufocus.c`. Edit this file to create the new functions by simply changing all occurrences of `redfocus` to `blufocus` and `REDFOCUS` to `BLUFOCUS`. Also update the header comments. The FIORD functions are ready to be compiled now but since the stage calibration information is explicitly defined here, they must be updated again after the stage has been calibrated.

Finally, update the `makefile` to add `f_blufocus` to the list of `NAMES` in the file.

0.7.2 VME Software

Now update the VxWorks software that runs on the VME crate. Use the file `s_redfocus.c` as a template for the `s_blufocus.c` file where the `s_set_blufocus` and `s_show_blufocus()` functions will be stored. As with the FIORD functions, change all instances of `redfocus` to `blufocus` and `REDFOCUS` to `BLUFOCUS`. Change `printf`s so they refer to the blue focus rather than the red focus, and, of course, update the header comments to describe the new functions.

Update `server_table.c`:

```
LRS_SET_BLUFOCUS , s_set_blufocus ,
LRS_SHOW_BLUFOCUS , s_show_blufocus ,
```

In the file `load_lserv` add the following line near the same line for `redfocus`:

```
ld<s_blufocus.o
```

Update the file `lrs_global_init.c` to add entries for the new stage and encoder. Make a copy of the stage table entry for the red camera focus and use it as a template for the blue camera focus. The final entry should look like this:

```
{ "blue_camera_focus",     /* name of stage 18 */
  3,                       /* move controller */
  3,                       /* status controller */
  "0000000000000000",      /* status input bits */
  "00001100",              /* output select sequence */
  0,                       /* abs encoder - no home */
  0,                       /* abs encoder - no home */
  "A=500,D=10000,B=200,H=50000,M=10000,ID2=11111111,S=100\r\n",
                           /* configuration sequence */
  "PAUSE,MOV ",            /* move sequence */
  0,                       /* standard displacement */
  0,                       /* number of positions */
  4084863,                 /* minimum encoder position */
  4202736,                 /* maximum encoder position */
  -1,                      /* no aux stages */
  4,                       /* absolute encoder */
  -1,0,0,0,0,0,0,0,0,0     /* abs position table */
  },
```

The minimum and maximum encoder positions listed above are of course wrong. These must be changed after the stage is calibrated.

Similarly, use the red camera focus entry in the encoder table to add an entry for the blue. It should look like this:

```
{ "AX",          /* camera focus encoder controller */
  "4",           /* daisy chain address */
  6,             /* error tolerence (encoder steps) */
  16,            /* number of times to try move */
  12800,         /* motor steps per shaft rev */
  10000,         /* encoder steps per shaft rev */
  "",            /* intialization sequence */
  "PR "          /* read sequence */
},
```

The only item changed in this entry is the daisy chain address.

Finally, add an entry to the `current_positions[]` array with a value of `-2`, indicating the power up default value for the stage's position is unknown. And add an entry to the initial values of the `home_flags[]` array with a value of 0.

Update `makefile` to add the new module. Add the object filename to the `all:` list and add the dependency and compile lines as in the following:

```
s_blufocus.o:  s_blufocus.c $(CDIR)/lris_msgs.h $(IDIR)/mcs.h
        cc $(CFLAGS) s_blufocus.c
```

Use:

```
cc $(CFLAGS) -DSIMULATE s_blufocus.c
```

to compile the source so that the simulation mode is activated. This can be used to test the new functions without actually moving the stage.

0.7.3 Initial Testing and Calibration

Once the stage is electrically connected to the LRIS multiplexor system, the operating characteristics can be determined. Use the `api` program to manipulate the motor directly and the `ax` program to read the encoder.

First attempt to read the encoder:

1. Type `ax` on the VxWorks console.
2. Type `4pr` to get the position report from the encoder at address 4.
3. Type `q` to quit the program.

If there is no response from the encoder, check cabling and power. Make sure the encoder controller is configured correclty and is set to the correct address.

Next attempt to move the new motor to check for obstructions or interferences and then determine the minimum and maximum positions:

1. Type `api` on the VxWorks console.
2. Type `:3` to switch to API unit number 3.
3. Type `set 5 6` to switch to the blue camera focus motor. Recall the multiplexor address is 0x0c or 00001100 in binary, and API numbers bits from left to right starting with 1.
4. Type `a=500,d=10000,b=200,h=10000,m=100000` to enter the velocity profile.

5. Type `mov 1000` to move a small distance.
6. Type `mov 10000` to move a larger distance.
7. Type `move -10000` to try negative moves.
8. Type `move -10000` to go negative again. This should be repeated throughout the full range of travel, with carefull visual inspection to make sure there are no interferences that could cause the motor to stall or damage other components. Stop when the motor reaches the negative limit.
9. Type `ver i2` to see if the limit switch is set. Bit 7 of I2 should be set if a negative limit is encountered. Check the positive limit also.
10. Type `q` to quit the `api` program.
11. Type `ax`
12. Type `4pr` to read the encoder. This reading is the number that goes into the minimum encoder position field in the table entry in `lrs_global_init.c` and is also the value of BLUFOCUS_ENCODER_OFFSET in module `f_blufocus.c`.
13. Type `q` to quit the `ax` program.
14. Type `api`. It should not be necessary to switch back to API 3 or select the new motor.
15. Type `mov 100000000` to move the stage to the positive limit.
16. Type `q` to quit the `api` program.
17. Type `ax`.
18. Type `4pr` to read the encoder again. This value is the number that goes into the maximum encoder position field in the blue camera stage table entry in `lrs_global_init.c`.
19. Type `q` to quit the `ax` program.

Note if the minimum encoder position is a larger number than the maximum encoder position, the encoder shaft must be uncoupled from the motor and rotated sufficently so that the maximum position will be a greater value than the minimum position.

Use a similar procedure to the above to move the stage around 100000 or so motor steps and measure the actual displacement in microns and in encoder steps. These measurements give you the scale factor, the value for `BLUFOCUS_ESPM` in `f_blufocus.c`, in encoder steps/micron to convert from one units system to the other. Now you can calculate the `BLUFOCUS_MAX` and `BLUFOCUS_MIN` values in microns also defined in module `f_blufocus.c`. If you trust your hardware engineer to tell you what the scale is, then you don't need to actually make the measurement.

When both modules `f_blufocus.c` in the FIORD software and `lrs_global_init.c` in the VxWorks software are updated with the correct values for the stage's parameters, the stage and software are ready to test.