VME Software for LRIS Motor Control

John Cromer

Issued: June 8, 1993.
Revised: November 8, 1994

Contents

0.1 Introduction

This document describes the software, written using the VxWorks development system, that runs on the VME, Sparc 1E crate used to control the LRIS mechanical stages. This software consists primarily of the "cserv" functions, spawned by the program `cserv` when it receives a message from the host workstation. The low level routines used to communicate with the motor controllers and encoders which are called by the `cserv` program are described as well as functions which can be executed from the VxWorks console. Note on the LRIS, `cserv` is actually called `lserv`, as opposed to the HIRES, where it is called `hserv`. For more information on the mechanics of how the VME software handles messages from the workstation see the `cserv` manual.

For detailed descriptions of individual functions see the Function Description document produced from the source code headers of motor control software functions with the program `wflman`.

0.2 Hardware Description

Figure 1. shows the hardware layout of the LRIS motor control system. All communications from the VME system to the instrument run through two serial cables, one to a daisy chain of four intelligent motor controllers and one to a daisy chain of absolute encoders. Each controller is multiplexed to a number of motors and the identity of the motor under control is encoded in the output bits of the controller. Each controller has 2 8-bit input words or 16 input bits. LRIS discrete mechanical stage positions are encoded in these input bits. Limit switches are connected to input bits 7 and 8 on input word `I2`. Home switches, when used, are connected to bit 6 on input word `I2`. Absolute encoders are used to index other stages.

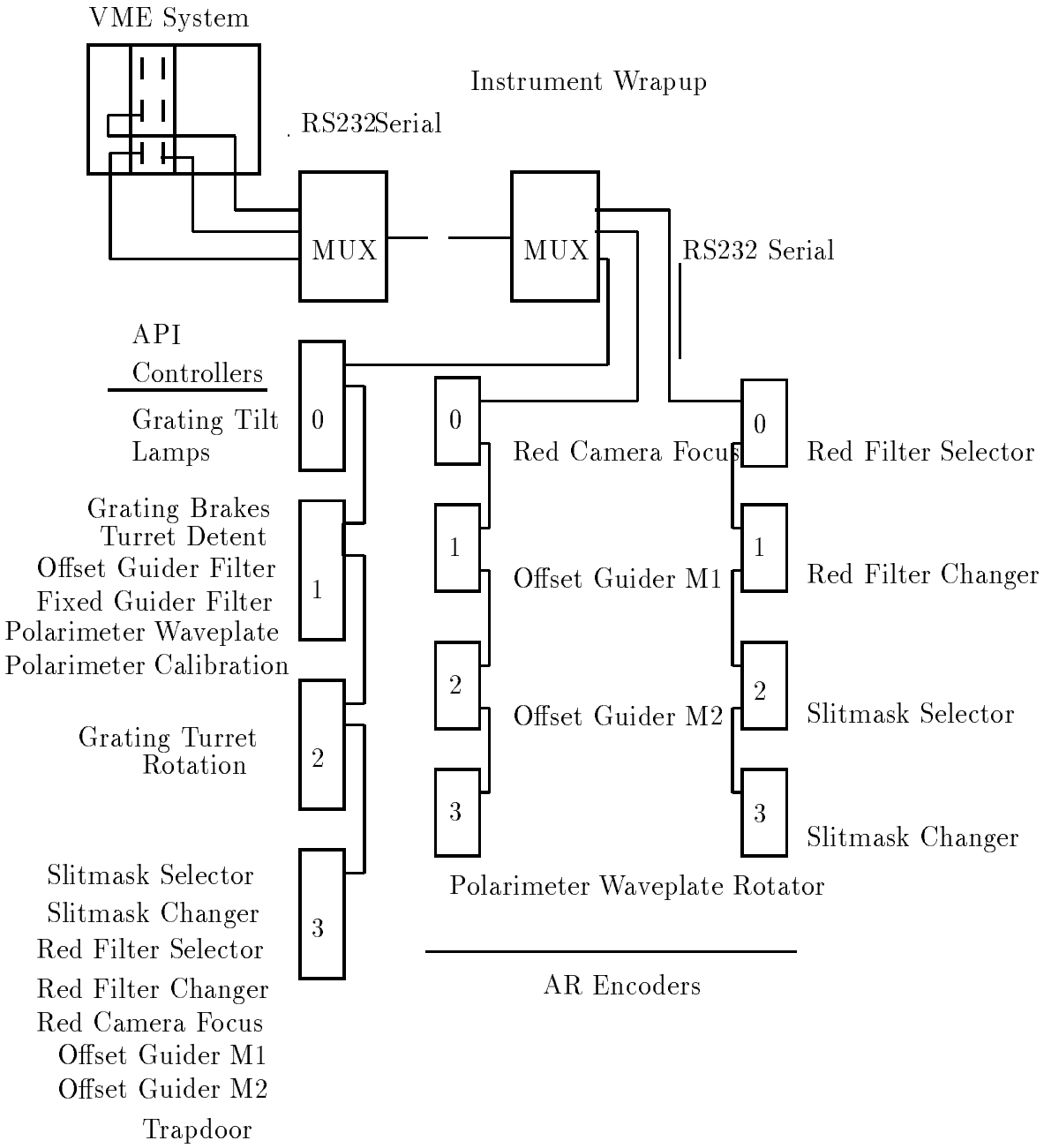

Figure 1. LRIS Motor Control Hardware Overview

Below the LRIS mechanical stages are listed.

Table 1. Movable Stage Numbers, Names and Types

Stage Number	Stage Name	Stage Type
0	Grating Turret	Discrete
1	Turret Detent	Limit
2	Grating Tilt Brake	Limit
3	Grating Tilt	Encoder
4	Red Filter Selector	Encoder
5	Red Filter Changer	Encoder
6	Blue Filter Selector	Encoder
7	Blue Filter Changer	Encoder
8	Red Camera Focus	Encoder
9	Slitmask Selector	Encoder
10	Slitmask Changer	Encoder
11	Offset Guider M2	Encoder
12	Offset Guider M1	Encoder
13	Offset Guider Filter	Blind
14	Slitview Guider Filter	Blind
15	Trapdoor	Limit
16	Polarimeter Calibration Wheel	Discrete
17	Polarimeter Waveplate Rotator	Encoder

Discrete stages are those which have a few discrete positions numbered 1 through M, M being the maximum number of positions available. The number -2 is used to designate a position that is unknown to the software.

Limit stages are those that move between limit switches. They have only two valid positions. In the software, those two positions are designated -1 and +1, with -2 being "unknown." When a limit stage is positioned such that neither limit switch is engaged, the position in the software will be reported as -2.

Encoder stages are those that have encoders attached and whose positions are more or less continuous to the limit of the encoder resolution. There are two different kinds of encoders used on the LRIS. One is an incremental encoder used on the grating tilt mechanism. These encoders are attached directly to the API controller unit and are read through the controller. An incremental encoder loses its reading when power is removed from it and must be homed to be calibrated. The other kind of encoder is an absolute encoder. Absolute encoders are connected to their own controllers, have individual addresses on a separate serial link, retain their readings when power is removed and must be queried separately apart from the API controller that actually moves the motor. Table 2. shows a list of all LRIS encoders, the names of the stages they serve, and the type of encoder.

Table 2. Encoder Numbers, Names and Types

Encoder Number	Stage Name	Encoder Type
0	Grating Tilt	Incremental
1	Red Camera Focus	Absolute
2	Offset Guider M2	Absolute
3	Offset Guider M1	Absolute
4	Polarimeter WavePlate	Absolute
5	Red Filter Selector	Absolute
6	Red Filter Changer	Absolute
7	Slitmask Selector	Absolute
8	Slitmask Changer	Absolute

In the source code and comments, encoder stages are sometime referred to as continuous or float stages.

Blind stages are those that have only a single limit switch to provide position feedback. They must be moved to the limit and then indexed a given number of motor steps to the destination position. There is no way for the software to know if the move was successful or not unless the first limit move failed. Blind stages are discrete.

0.3 Source Code Locations

In general the motor control software is partitioned as follows.

Table 3. LSERV Modules and functions		
Module	Functions	Description
lrs_encoder.c	read_abs_encoder() move_abs_encoder()	Functions that deal with absolute encoders
lrs_global_init.c	lrs_global_init()	Initialization of all global tables and data
lrs_init.c	lrs_init() abs_init() mcs_init()	Initialization of LRIS controller and encoder hardware
lrs_mcs.c	mcs_com() mcs_limit() mcs_output() mcs_switch() mcs_verify()	Functions that communicate with motor controllers directly
lrs_move.c	move_int_simple() move_float_simple() move_compound() move_api_encoder move_gturret() move_int_blind() home_simple() home_compound()	Functions which move stages
lrs_changers.c	move_float_changer() move_sc()	Functions which move jukeboxes
lrs_verify.c	verify_pos_num() get_stage_position_code() get_stage_position() bins_to_int() set_all_positions() read_api_encoder() zero_api_encoder()	Functions which verify the states of the LRIS hardware
lrs_save.c	write_stage_position() read_stage_position()	Functions to save and restore stage positions to disk
s_*keyword*.c	s_set_*keyword*() s_show_*keyword*()	cserv and associated specific functions
api.c	api()	A program to communicate to the controllers directly.
ax.c	ax()	A program to communicate to the encoders directly.

All motor control software resides in the directories under `/kroot/kss/lris/vme/lserv`. In the directory `/kroot/kss/lris/vme/lserv/shell` there is a subdirectory for each mechanical stage on the LRIS. These directories contain functions for manipulating each stage at a low level from the VME console.

Table 4. Low Level Modules and Functions

Module	Functions
current.c	set_current() show_current() show_lamps()
gr_optical.c	gr_optical()
gr_service.c	gr_service()
move_detent.c	turret_detent_in() turret_detent_out()
move_red_filter_changer.c	red_filter_changer_in() red_filter_changer_out()
move_red_filter_selector.c	move_red_filter_selector()
move_slitmask_changer.c	slitmask_changer_in() slitmask_changer_out()
move_slitmask_selector.c	move_slitmask_selector()
move_turret.c	move_turret()
move_tv1filt.c	move_tv1filt()
move_tv2filt.c	move_tv2filt()
show.c	show()
trapdoor.c	close_trapdoor() open_trapdoor()
tv1_init.c	tv1_init()

Detailed descriptions of the use of these functions from the VxWorks prompt on the VME console are given in the last chapter of the *LRIS Motor Control Software Trouble Shooting Guide*.

0.4 Controller Communication and Control

0.4.1 The Serial Port Driver

The serial port driver consists of a number of functions in the modules `sio2Drv.c` and `sio2mpcc.h` that form a seamless VxWorks subsystem for controlling the Force SIO2 Serial Interface Card. Below is a list of functions in the driver and a very brief description of each.

Table 5. Serial Driver Functions	
sio2DevCreate()	Create and install VxWorks I/O device
sio2Drv()	Initialize and install VxWorks driver functions
sio2HrdInit()	Initialize serial board hardware
sio2Int()	Service an interrupt for port 0
sio2Int1()	Service an interrupt for port 1
sio2Int2()	Service an interrupt for port 2
sio2Ioctl()	Handle a task level I/O control request
sio2Open()	Return a pointer to the device software structure
sio2Read()	Perform a task level read of the serial port buffer
sio2Write()	Perform a task level write to the serial port buffer
sio2RxInt()	Service a receiver interrupt
sio2TxInt()	Service a transmitter interrupt
sio2SerialInt()	Service a serial control interrupt
sio2SetBaudRate()	Set the serial board baud rate
sio2Startup()	Start up the serial board transmitter

For more information about these functions, see the function description document generated from the function headers by `wflman`. Since the driver is a part of the VxWorks system, its source in in directory `/usr/vw502b/config/sun1e`.

0.4.2 Controller Communication

Only one function is required to handle all communications between the VME software and the serial port driver. This function is `mcs_com()` in module `lrs_mcs.c`. The function descriptions document provides a detailed description of how this functions works, but a brief description will be given here. The syntax for the function call is:

```
mcs_com(mode, command, reply)
```

where

`mode` is MCS_RETURN_VALUE, MCS_CONSOLE_OUT, or MCS_WAIT_FOR_MOVE as defined in `mcs.h`,

MCS_RETURN_VALUE indicates the controller will return a parameter value and this value should be passed back in the `reply` parameter,

MCS_CONSOLE_OUT indicates that all output from the controller will be displayed to the console only. No `reply` is returned,

MCS_WAIT_FOR_MOVE indicates `command` contains a controller move request and `mcs_com()` should wait for the move to complete,

`command` is the controller ASCII command sequence,

`reply` is an ASCII string returning any reply from the controller.

All functions which send and receive information to and from the motor controllers do so by calling this function.

0.4.3 Controller Support Functions

There are several other functions which deal more or less directly with the LRIS motor controllers. They are:

Table 6. API Support Functions	
Function	Description
mcs_limit()	Read the limit switch values from the current controller and return a value based on the switch values. No parameters required.
mcs_output()	Set output bits on the current controller. An 8-character ASCII parameter encodes the bits to set. `mcs_output("00000111")` will set bits 6, 7 and 8.
mcs_switch()	Switch the active controller to the one given by the unit number in the parameter. `mcs_switch(2)` makes the active controller unit number 2.
mcs_verify()	Execute the API "verify" command and return the parameter value to the calling program. `mcs_verify("I2",TYPE_CHAR,sval,&val)` will return in `sval` the value for the `I2` parameter. `mcs_verify("B",TYPE_INT,sval,&val)` will return in `val` the value of the `B` parameter.

More detail regarding these functions can be found in the Function Descriptions manual generated from the function headers by `wflman`.

0.4.4 Status Report Functions

Status information from the LRIS stages is available in the form of limit switch readings, input bit readings, incremental encoder readings and absolute encoder readings. Limit switch readings are handled directly by the function `mcs_limit()` discussed in table 6. Input bit readings are used to establish the positions of discrete stages. See Table 1 for a list of stages and stage types. The following programs check input bits and return stage positions.

Table 7. Functions for Reading Discrete Stage Positions	
Function Name	Brief Description
get_stage_position_code()	Read a position code from a controller.
get_stage_position()	Read a position code and convert it to a position.
verify_pos_num()	Read a position and compare it with a destination position.

Note `get_stage_position()` will return the position of any stage, no matter what type. It has calls to all the functions which return the status of individual stages.

More detail regarding these functions can be found in the Function Descriptions manual generated from the function headers by `wflman`.

As previously discussed, incremental encoders are those which lose their memory when power is removed and do not have their own controller. They must be connected to an API motor controller. Nominally the call: `lrs_verify("E",TYPE_INT,sval,&val)` would return the encoder reading in `val`, however since the correct controller must be selected along with a number of other things, a separate routine that does it all is provided. The following functions deal with incremental encoders.

Table 8. Incremental Encoder Functions

Function Name	Brief Description
read_api_encoder()	Read an incremental encoder through an API controller.
zero_api_encoder()	Zero an incremental encoder through an API controller.

See the Function Descriptions document generated with `wflman` for more details regarding these functions.

All encoder stages on the LRIS, except the grating tilt mechanisms, use absolute encoders. Each absolute encoder has its own serial interface. These interfaces are daisy chained and connected to the LRIS serial multiplexor as shown in Figure 1. Compumotor AR encoder interfaces are used. For a description of the command language used by these boxes see the *Compumotor AR User Manual*. Two functions are used on the VME crate to read absolute encoders and to move their associated stages.

Table 9. Absolute Encoder Functions

Function Name	Brief Description
read_abs_encoder()	Read an absolute encoder.
move_abs_encoder()	Move an absolute encoder stage.

See the Function Descriptions document generated with `wflman` for more details regarding these functions.

0.4.5 Save Status Functions

Since incremental encoders lose their readings when power is removed, a mechanism in software is provided to save these readings in case of a power failure. Two functions implement this mechanism.

Table 10. Status Save Functions

Function Name	Brief Description
write_stage_pos()	Write stage ID and current position to disk.
read_stage_pos()	Read stage ID and position from disk.

The function `write_stage_pos()` is invoked after every move of a stage with volatile position feedback. Currently, the LRIS has only one such stage, the grating tilt mechanism. Although the VME system is "diskless," the VxWorks NFS mechanism is used to write the file over the network to the instrument workstation. Stage status files are located with other database information in the `/usr/local/music/info` directory. The function `read_stage_pos()` is called by `lrs_init()` when the VME system is started up or whenever `lrs_init()` is run manually from the VME console. For details of these functions, see the Function Descriptions Document created from the headers by `wflman`.

0.4.6 Initialization Functions

A number of functions are executed upon startup and should be executed whenever power is cycled on the LRIS. (It is not necessary to reboot the VME crate because of a power loss to the instrument.) These functions send various initialization sequences to the LRIS controllers, read the stage positions and reference lamp states, and update the global sections of the VME software. The functions are:

Table 11. Hardware Initialization Functions

Function Name	Brief Description
lrs_init()	Initialize the instrument controllers and VME software.
mcs_init()	Initialize only the motor controllers.
abs_init()	Initialize only the absolute encoders.

They are all found in the module `lrs_init.c`. For details of these functions, see the Function Descriptions Document created from the headers by `wflman`.

One other function is used to initialize the VME software.

Table 12. Global Data Initialization Functions

Function Name	Brief Description
lrs_global_init()	Initialize global data and semaphores.

This module is where the stage information, encoder information and current position tables are initialized with default data. It is in a separate module, `lrs_global_init.c`. This function also calls the VxWorks routines to create the semaphores used to ensure exclusive access to the serial ports, and is only executed during the VxWorks motor control startup script.

0.4.7 Motion Control

Several general and specific functions exist to move the LRIS stages. These functions are:

Table 13. Motion Control Functions	
Function Name	Brief Description
move_int_simple()	Move a simple discrete stage.
move_float_simple()	Move a simple encoder stage.
move_float_changer()	Move a encoder-based "changer" stage
move_compound()	Move a compound stage.
move_sc()	Move a compound selector/changer stage
move_api_encoder()	Move an incremental encoder stage.
move_abs_encoder()	Move an absolute encoder stage.
move_int_blind()	Move a discrete blind stage.
move_gturret()	Move the grating turret.

The above routines are found in `lrs_move.c`, `lrs_encoder.c` and `lrs_changers.c`. For more details refer to the Function Descriptions document and to the notes for individual stages in this manual. For convenience, the algorithm outlines for `move_int_simple()` and `move_compound()` are given below.

`move_int_simple()`:

1. Check for legal stage number.
2. Check for legal position request.
3. Call `mcs_switch()` to switch to correct controller unit.
4. Call `mcs_output()` to select the correct motor.
5. Call `mcs_com()` to send the controller configuration sequence.
6. Call `get_stage_position()` if current position is not known.
7. Calculate move displacement.
8. Call `mcs_com()` to send the move sequence.
9. Verify destination position was reached. Call `mcs_limit()` is this is a limit stage, `verify_pos_num()` if a discrete stage.
10. Call `mcs_output()` to de-select the motor.
11. Update `current_positions[]` with the destination positions.

`move_float_simple`:

1. Check for legal stage number.
2. Check for legal positions request.
3. Call `mcs_switch()` to switch to correct controller unit.

4. Call `mcs_output()` to select the correct motor.
5. Call `mcs_com()` to send the controller configuration sequence.
6. Call `move_api_encoder()` if this is an incremental encoder stage, or `move_abs_encoder()` if this is an absolute encoder stage, to make the move.

`move_compound()`:

1. Check for legal stage number.
2. Check for legal position request.
3. Call `move_int_simple()` for the first auxillary stage move.
4. Make the main move by calling `move_int_simple()` for discrete and limit stages, `move_float_simple()` for encoder stages.
5. Call `move_int_simple()` for the last auxillary stage move.

In Table 14. each LRIS movable stage is listed with which of the above functions is called by the VME `lserv` function to move it.

Table 14. Stage Names and Move Functions

Stage Name	Move Function
Grating Turret	move_gturret()
Turret Detent	move_int_simple()
Grating Tilt Brake	move_int_simple()
Grating Tilt	move_compound
Red Filter Selector	move_sc()
Red Filter Changer	move_float_changer()
Blue Filter Selector	move_compound()
Blue Filter Changer	move_int_simple()
Red Camera Focus	move_float_simple()
Slitmask Selector	move_sc()
Slitmask Changer	move_float_changer()
Offset Guider M2	move_float_simple()
Offset Guider M1	move_float_simple()
Offset Guider Filter	move_int_blind()
Slitview Guider Filter	move_int_blind()
Trapdoor	move_int_simple()
Polarimeter Calibration Wheel	move_int_simple()
Polarimeter Waveplate Rotator	move_float_simple()

In the fall of 1994, the slitmask and red filter selector/changer mechanisms were converted completely to encoder-based position feedback. Originally, microswitches were used to supply position data. Since these proved to be unreliable they were replaced with encoders. The

software was changed at this point and two new functions for moving the selector changers were introduced: `move_float_changer()` and `move_sc()`. `move_sc()` is simply the same algorithm as `move_compound()` but it calls `move_float_changer()` to actually move the stages. It also switches to the correct API controller and the software henceforth assumes that both selector and changer are controlled from the same API unit. Below is the algorithm used in `move_float_changer()` to move the individual motors of the selector/changer mechanisms:

1. Check for legal stage number.
2. Check for legal positions request.
3. Call `mcs_output()` to select the correct motor.
4. Call `mcs_com()` to apply power to the stage if necessary.
5. Call `mcs_com()` to release the brake if necessary.
6. Call `mcs_com()` to configure the stage.
7. Call `move_api_encoder()` if this is an incremental encoder stage, or `move_abs_encoder()` if this is an absolute encoder stage, to make the move.
8. Call `mcs_com()` to apply the brakes if necessary.

Note the algorithm is similar to that of the `move_float_simple()` function except for the addition of code to control power and brakes explictly.

Note also the configuration sequence for both the slitmask changer and the red filter changer contains a `S=200` API command which sets the current hold time after the move completes. This means current will remain applied to the motor for 2 seconds after the move finishes which gives the software time to apply the brakes and maintains the position of the changer during this time. A longer time interval can interfer with subsequent commands to the controller and lead to the controller rebooting itself or losing serial communications.

0.5 Global Data for the VME Motor Control Software

The VME software for motor control maintains tables in global memory, some statically initialized which provide information and parameters about LRIS controllers and encoders, others which are dynamic and store the states of LRIS stages. Information about movable stages, stage controllers and controller parameters is kept in an array of structures called `stages`. Each movable stage on the LRIS has an entry in this array of structures. The stage structure is defined in `stagestruct.h` and is of the following form:

```
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;
```

Each of the structure members is described below.

`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.

`max_positions`. The stage's total number of discrete positions. This applies to discrete 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.

`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.

`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. These are the numbers that 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, 8 and 1. 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`.

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 */
    int    min_pos;         /* minimum encoder position limit*/
    int    max_pos;         /* maximum encoder position limit */
    int    offset;          /* encoder offset */
    int    scale;           /* encoder scale */
    char   *port            /* encoder serial port */
    } 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. This is the kind of information found in the "idiosyncrasies" section of the manual. Suffice it to say for now that where ever you see an AX in the software, read it as AR.

`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. In general, 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 student will recognize as the API verify command requesting the value of the `E` parameter.

`min_pos`. The encoder minimum position. This is the encoder position at its absolute minimum of travel – usually against a negative limit switch. This value is used as a software limit in the VME software and is defined in the file `encoder_params.h` in the `fiord` section of the `/kroot` directory tree.

`max_pos`. The encoder maximum position. This is the encoder position at its absolute maximum of travel – usually against a positive limit switch. This value is used as a software limit in the VME software and is defined in the file `encoder_params.h` in the `fiord` section of the `/kroot` directory tree.

`offset`. The encoder offset. This is the encoder value at a known reference location. It is sometime called the "bias" or "intercept." It's value is defined in the file `encoder_params.h` in the `fiord` section of the `/kroot` directory tree.

`scale`. The encoder scale. This is the value used to convert absolute encoder units to user units. It is sometime called the "slope." It's value is defined in the file `encoder_params.h` in the `fiord` section of the `/kroot` directory tree.

`port`. The encoder serial port. This is a character string that designates the VxWorks I/O system name of the serial port to which the encoder is attached. The two possible values are "`/sio2/1`" and "`/sio2/2`."

A global array of integers is used to store the current positions of all stages. The contents of this array, `current_positions[]`, is initialized with default values of -2 (unknown) in the function `lrs_global_init.c`. The startup software, specifically, `lrs_init()`, requests the positions of all stages from the hardware and updates this array. The position of the grating tilt mechanism is read from a file. Currently, this is the only stage on the LRIS whose position is volatile. (It is also the only stage which normally must be homed.)

The reference lamp states are also stored in a global integer array, `lamp_status[]`.

Other global data are defined in `lrs_global_init.c` but are currently not used.

0.6 Stages

In this section, individual stages will be discussed. Some notes on the stage's motion will be given to provide context to the software discussion. The `lserv()` functions will be described, the move functions used and unusual features and necessary actions will be discussed.

In general, the "set" routines all do the same thing:

1. Unpack the stage ID from the music message.
2. Unpack the destination position from the music message.
3. Send the first status message back to the workstation.
4. Take the serial port semaphore(s).
5. Pass the ID and position to the appropriate "move" function
6. Give the serial port semaphores(s).
7. Update the current position global table.
8. Send the final status message back to the workstation.

The "show" routines are similar in function. They each unpack the stage ID from the music message, use it to index into the `current_positions[]` array return the value found there, suitably translated into a stage position. In general, "show" functions do not send requests to the instrument for stage states. The reason for this can be found in the design of the system. It is a "serial" system. When a request is made to the instrument, the serial port is locked exclusively to the requesting function. Any other requests to the instrument are queued, waiting for the port semaphore to be released. If this happened during a request for some position to go into the image header, the image writing functions might time out and not be able to get the value for the header. While not as robust as a direct query, obtaining the stage positions from global VME memory provides a satisfactory way around the serial limitations of the system.

Note all conversions from encoder readings to user units are done on the host workstation in the FIORD system. See the *FIORD Software for LRIS Motor Control* manual for details. The `lserv()` functions only deal with switch and encoder readings, not mm, microns, etc.

Below, each stage and its associated `lserv` function will be discussed. Idiosyncrasies and deviations from the norm will be noted. Unusual requirements and how they are handled for each stage will be discussed.

0.6.1 The Grating Turret Subsystem.

Grating Turret.

The `lserv()` functions `s_set_grating()` and `s_show_grating` field music messages and handle requests relating to the grating turret.

The grating turret is the most complex of the movable mechanisms on the LRIS. The turret rotates, moving any of 5 different grating stations to either of two different positions, the optical port or the service port, making a total of 10 legal turret positions. The grating turret is a compound stage. The turret detent mechanism, which locks the grating turret into position, is the auxillary stage. The turret detent must be moved before and after each grating turret move. Additionally, the turret cannot be moved if a grating is tilted. The software to move the turret must detect this condition and home the grating tilt before attempting the turret move. In order to accomplish all this the `s_set_grating()` function calls a custom move function `move_gturret()`. Here are the general algorithms for each:

`s_set_grating()`:

1. Unpack stage ID and desired grating number from music message.
2. Send back first status to host workstation.
3. Take the controller serial port semaphore.
4. Convert desired grating number to turret position number.
5. Call `move_gturret()` to make the move.
6. Update `current_positions[stage_id]` with the new position.
7. Release the semaphore.
8. Call `zero_api_encoder()` to zero the current grating encoder.
9. Send final status back to host workstation.

`move_gturret()`:

1. Switch to grating tilt controller.

2. Send configuration sequence to grating tilt controller.
3. Set output bit 1 on controller to turn on opto+5.
4. Check for all grating tilt mechanisms at home.
5. Home the current grating tilt if necessary.
6. Reset output bit 1 on this controller to turn off opto+5.
7. Switch to grating turret controller.
8. Apply power to the turret.
9. Remove the grating detent.
10. Move the turret.
11. Check success of move.
12. Insert the grating detent.
13. Remove power from turret.

For more details, see the function description for `move_gturret()` in the `wflman` generated Function descriptions document.

Note step 8 in the `s_set_grating()` algorithm. After a turret move completes, a new grating station is connected to its motor controller via the turret position switch. Since the grating tilt encoder is a incremental encoder, the encoder reading after a grating turret move is invalid. Setting the encoder to zero is valid because the grating tilt is at its home location, else the turret would not have moved.

The `s_show_grating` function handles the music request for the grating station that is currently in the optical port. Note this is not the same number as the grating position number. The turret position number is read from `current_positions[]` as usual but this number must be further translated to the grating station in the optical port. The array `opt_positions` is used to convert a grating turret position number to a optical port grating station number. This number is sent back to the host workstation to satisfy the `show grating` request.

The Grating Tilt

The `lserv()` functions `s_set_grangle()` and `s_show_grangle` field requests for the grating tilt mechanism. This mechanism is a compound mechanism, with the grating brakes as the auxillary stage. The brakes must be released before each tilt move and applied after each move. The `s_set_grangle()` function implements the following algorithm:

1. Unpack stage ID and destination encoder position from music message.

2. Send first status message back to host workstation.
3. Check grating turret position. If the position is an odd number, this indicates the grating station is at the service port and cannot be tilted. Return an error. If the positions is not an odd number continue.
4. Take the controller serial port semaphore.
5. Call `move_compound` to make the tilt move.
6. Call `read_api_encoder` to determine the final tilt position.
7. Update `current_positions[stage_id]` with this value.
8. Call `write_stage_pos()` to save the value to disk.
9. Release the controller serial port semaphore.
10. Send final status to host workstation.

The `s_show_grangle()` function is completely standard:

1. Unpack stage ID from music message.
2. Send back `current_positions[stage ID]` as the reply.

The grating tilt stage is the only stage on the LRIS that needs to be homed for normal operation. All of the other stages have nonvolatile mechanisms for position reporting.

Currently there are no keywords for moving either the turret detent or the grating brakes independently. It seems like a dangerous thing to provide. These stages can be moved "manually" from the VME console. See the *LRIS Motor Control Troubleshooting Guide* for details.

0.6.2 The Red Filter Box

.

The functions `s_set_redfnum()` and `s_show_redfnum()` handle requests for the red filter selector/changer.

The red filter selector is a compound mechanism with the filter changer (or grabber as it's sometimes called) as its auxillary stage. A complete filter move consists of removing the current filter from the optical path with the grabber, moving the selector to the desired filter, and inserting this filter into the optical path with the grabber. The LRIS has 6 different filters for the red camera, including a clear holder, and hence the filter selector has 6 positions, numbered 1–6. For the changer stage, the software must first apply power and then release the brake before a move can be made. After the move is completed the brake is then applied before the power is released. The `move_sc()` and `move_float_changer()` functions do this.

`s_set_redfnum()`:

1. Unpack stage ID and desired red filter number from music message.
2. Send back first status to host workstation.
3. Take the controller serial port semaphore.
4. Call `move_sc()` to make the move.
5. Update `current_positions[]` with the new position.
6. Release the semaphore.
7. Send final status back to host workstation.

`s_show_redfnum()`:

1. Unpack stage ID from music message.
2. Send back `current_positions[stage ID]` as the reply.

0.6.3 The Slitmask Box

.
The functions `s_set_slitmask()` and `s_show_slitmask()` handle requests for the slitmask selector/changer.

The slitmask selector is a compound mechanism with the slitmask changer (or grabber) as its auxillary stage. A complete slitmask move consists of removing the current slitmask from the optical path with the grabber, moving the selector to the desired slitmask, and inserting this slitmask into the optical path with the grabber. The LRIS has 10 different slitmask positions numbered 1–10. For the changer stage, the software must first apply power and then release the brake before a move can be made. After the move is completed the brake is then applied before the power is released. The `move_sc()` and `move_float_changer()` functions do this.

`s_set_slitmask()`:

1. Unpack stage ID and desired slitmask number from music message.
2. Send back first status to host workstation.
3. Take the controller serial port semaphore.
4. Call `move_sc()` to make the move.
5. Update `current_positions[]` with the new position.
6. Release the semaphore.
7. Send final status back to host workstation.

`s_show_slitmask()`:

1. Unpack stage ID from music message.
2. Send back `current_positions[stage ID]` as the reply.

0.6.4 The Guider Subsystem

.
The guider subsystem consists of the offset guider which as two motors, the slit viewing guider which has no motors, and a movable filter wheel for each.

The Offset Guider

.
The functions `s_set_tv1fpos()` and `s_show_tv1fpos()` handle requests for the offset guider.

The offset guider mechanism consists of two motors each with an encoder. One motor, designated M1, moves a pickoff mirror with respect to the guider camera lens, simultaneously adjusting its position and the guider focus. The other motor, designated M2, moves the entire offset guider assembly changing only the position of the pickoff mirror, not the guider focus. Both motors must be moved with each offset guider request in order to change the pickoff mirror position and keep the image in focus. Thus, the `s_set_tv1fpos()` function makes two calls to `move_float_simple()` instead of just one. The algorithm is:

1. Unpack stage ID and destination encoder position for M2 from music message.
2. Unpack stage ID and destination encoder position for M1 from music message.
3. Send back first status to host workstation.
4. Take the controller serial port semaphore.
5. Call `move_float_simple()` to move M2.
6. Update `current_positions[]` with M2's new position.
7. Call `move_float_simple()` to move M1.
8. Update `current_positions[]` with M1's new position.
9. Release the semaphore.
10. Send final status back to host workstation.

Similarly, `s_show_tv1fpos()` must send back the encoder positions of M1 *and* M2 in order for the FIORD function on the workstation to be able to calculate the actual position in mm, of the offset guider. The algorithm is:

1. Unpack stage ID for M2 from music message.
2. Unpack stage ID for M1 from music message.
3. Pack `current_positions[M2]` into music message.
4. Pack `current_positions[M1]` into music message.
5. Send back the music message.

The Guider filters

Requests to the offset guider filter wheel are handled by `s_set_tv1filt()` and `s_show_tv1filt()`. Requests to the slit viewing filter wheel are handled by `s_set_tv2filt()` and `s_show_tv2filt()`.

As noted previously, both the offset guider filter wheel and the slit viewing guider wheel have no positions feedback except for one limit switch each. In the software these are called blind stages. Both have 4 positions, numbered 1–4. Both are handled in the standard way with a call to `move_int_blind()` which moves the wheel to the limit switch first to get a reference point, then moves the wheel in the opposite direction the required number of motor steps to obtain the desired position. The `lserv()` function algorithms are:

`s_set_tv1filt()`:

1. Unpack stage ID and desired position number from music message.
2. Send back first status to host workstation.
3. Take the controller serial port semaphore.
4. Call `move_int_blind()` to make the move.
5. Update `current_positions[]` with the new position.
6. Release the semaphore.
7. Send final status back to host workstation.

`s_show_tv1filt()`:

1. Unpack stage ID from music message.
2. Send back `current_positions[stage ID]` as the reply.

The functions `s_set_tv2filt()` and `s_show_tv2filt()` operate in exactly the same manner.

0.6.5 The Red Camera Focus

.
The functions `s_set_redfocus()` and `s_show_redfocus()` handle requests for the red camera focusing mechanism.

The red camera focus mechanism is a simple, encoder stage that moves the camera barrel along the LRIS optical path. The full range of travel is only a few millimeters. No auxillary stages are required. The move is completely standard and uses only general mechanisms as they've been described above.

`s_set_redfocus()`:

1. Unpack stage ID and desired encoder position from music message.
2. Send back first status to host workstation.
3. Take the controller serial port semaphore.
4. Call `move_float_simple()` to make the move.
5. Update `current_positions[]` with the new position.
6. Release the semaphore.
7. Send final status back to host workstation.

`s_show_redfocus()`:

1. Unpack stage ID from music message.
2. Send back `current_positions[stage ID]` as the reply.

0.6.6 The Trapdoor

.
The functions `s_set_trapdoor()` and `s_show_trapdoor()` handle requests for the trapdoor stage.

The trapdoor mechanism is a simple, limit stage that opens and closes a toilet-lid-like door at the top of the spectrograph. It has only two valid positions. No auxillary stages are required. The move is completely standard and uses only general mechanisms as they've been described above.

`s_set_trapdoor()`:

1. Unpack stage ID and desired position from music message.
2. Send back first status to host workstation.

3. Take the controller serial port semaphore.
4. Call `move_int_simple()` to make the move.
5. Update `current_positions[]` with the new position.
6. Release the semaphore.
7. Send final status back to host workstation.

`s_show_trapdoor()`:

1. Unpack stage ID from music message.
2. Send back `current_positions[stage ID]` as the reply.

0.7 Software Control of Reference Lamps

Reference and calibration lamps are connected to bits 3–8 on API unit 0 (the grating tilt controller). Lamp 1 is assigned to bit 3, Lamp 2, to bit 4, etc. Although bits 7 and 8 are set up in the hardware and software, lamps are not connected to these switches yet. Switching a lamp on or off is a simple matter of setting or resetting the appropriate output bit. For example, the API command `set 5` would switch lamp 3 on. The command `reset 3 6` would switch lamps 1 and 4 off. In general the VME software simply sends these sequences to the controller and maintains a global status array called `lamp_status[]`. The following table gives the software functions used to control the lamps.

Table 15. Lamp Control Functions

Function Name	Brief Description
s_set_lamp()	Execute workstation request to switch a lamp on or off.
s_show_lamps()	Return status of all lamps to host workstation.
switch_lamps()	Send request to lamp controller to turn a lamp on or off.
get_lamp_statii()	Send request to lamp controller to read status of all lamps.

All lamp functions are found in the module `s_lamp.c`. Below are general algorithms executed by the above four functions.

`s_set_lamp()`.

1. Unpack lamp ID from music message.
2. Unpack integer value for music message.
3. Send first status to host workstation.

4. Take serial port semaphore.
5. Call `switch_lamps(lamp ID, bval)` to switch the lamp state.
6. Release the semaphore.
7. Send second status back to host workstation.

`s_show_lamps()`.

1. Pack `lamp_status[]` contents into music response.
2. Send response to host workstation.

`switch_lamps()`.

1. Check valid lamp ID.
2. Call mcs_switch() to select the correct controller.
3. Use `bval` to create controller request `set` or `reset`.
4. Switch on lamp ID to complete the request with the proper bit number to switch.
5. Send the request to the controller.
6. Send request to the controller to read back `O1`, the output bits.
7. Verify the appropriate bit was set (or reset).

`get_lamp_statii()`.

1. Call `mcs_switch()` to select the correct controller.
2. Call `mcs_com()` to read `O1` the output bits.
3. Update `lamp_status[]` with the state of the appropriate bits in `O1`.

Note there is no way for the software to know if the lamps are actually on. It can only read the states of the output bits.

0.8 Idiosyncrasies

Finally, we conclude the manual with a few notes about the major software idiosyncrasies and inconsistencies in the VME motor control software.

1. The Compumotor absolute encoders are often referred to, in naming symbols and in the comments as AX encoders. The correct designation, as a glance at the cover of the Compumotor Encoder User Manual will show, should be AR encoders. This is a result of a holdover from the days of Compumotor AX controllers, which were the controllers of choice when the LRIS was first designed.

2. There are a number of exceptions in the general stage handling code for the grating tilt mechanism. As noted before, the grating tilt position must be saved to disk after each move in order to recover from a power failure. The OPTO+5 input on the the grating tilt API controller (unit 0) must be activated before the stage is homed and deactivated afterwards. This is done by setting and reseting bit 1 (`set 1` and `reset 1`) on the controller before and after the home command is sent. See `move_gturret()` in module `lrs_move.c`. After every stage move, the output bits of the selected controller are reset to 0. This applies brakes on certain stages. On the grating tilt controller, this turns off all the lamps should any be on. An exception was added to the general move code to not reset the output bits for the grating tilt stage.

3. When checking the status of a limit stage, not only does the correct api unit need to be selected, but the correct motor must be selected by setting the appropriate output bits on the controller. This is not true of discrete or encoder stages.