NIRSPEC

UCLA Astrophysics Program U.C. Berkeley W.M. Keck Observatory

Samuel B. Larson/George Brims

Revised January 4, 1999

NIRSPEC Software Programming Note 20.00
Motion Control Transputer Code

1 Introduction

This document describes the transputer software that performs all motion control commands. It is organized in order of increasing complexity: a brief hardware/software overview, the software architecture, the individual modules that make up the software code, the motion control commands that the modules perform, and finally the generic procedures that the codes call when performing the commands. At the end of the document is a section describing how to make common software changes.

2 Overview

2.1 Hardware

NIRSPEC has three transputers dedicated to performing motion control commands. Each transputer is housed on a DAQ17 board with four 8-bit I/O ports and four 8-bit input ports. These ports can be addressed individually or together as a single 32-bit port. The I/O ports are used to send pulses to the motor drives to control motors, and the input port is used to read the status of microswitches.

The transputers are connected in a parallel architecture such that the first transputer (T1) receives all motion control commands, performing the commands it is responsible for and passing the rest to the second transputer (T2). Likewise, T2 performs its commands and passes the rest to the third transputer (T3).

All command acknowledgments and error messages are passed in the reverse manner: T3 sends its replies to T2. T2 sends its replies and those of T3 up to T1. T1 sends all replies up to the host transputer.

T1 is responsible for all commands to the filter wheels, the slit wheel, and the calibration unit cover slide. T2 handles the echelle grating, cross disperser grating, calibration unit flip mirror, and calibration unit pinhole mechanisms. T3 controls the image rotator mechanism.

2.2 Software

All transputer motion control software are written in the OCCAM parallel processing language and reside on the *nirspec* Sparcstation in the directory */kroot/kss/nirspec/transputer/motor*. There is an OCCAM program for each of the three transputers T1, T2, and T3: *motor1.occ*, *motor2.occ*, and *motor3.occ*, respectively. All previous versions of each program are also in this directory and are named by adding two digits to their parent name. For example, *motor109.occ* is the ninth version of the program *motor1.occ*.

In addition to the occam programs are three include files which are attached to the programs when they are compiled: *nirspec.inc*, *motor.inc*, *motor_clockdef.inc*. *Motor.inc* contains all mechanism characteristics (such as speeds and gear ratios), motor pulse output addresses, microswitch addresses, and acknowledgment identification numbers. *Motor_clockdef.inc* has the I/O port addresses on the transputer boards and values for the on-board timeout feature. *Nirspec.inc* is located in the directory

/kroot/kss/nirspec/transputer, and it contains the command identification numbers and transputer communication protocol definitions used by all the NIRSPEC OCCAM programs.

3 Software Architecture

Each transputer program has several modules running simultaneously: a watchdog, a command buffer, a report buffer, and one module for each mechanism under its jurisdiction.

3.1 Watchdog

The watchdog module controls a DAQ17 safety feature that sets all I/O bits to zero after a hardware timer expires. For the motors, this means that in case of a software glitch, motor power cannot be left on.

The watchdog is an ALT construction inside of a WHILE loop. The ALT waits for one of three things to occur:

1) a certain amount of time (shorter than the hardware timeout) to elapse, in which case the hardware timer is reset,
2) a message thru a hardware channel indicating a failure, which ends the WHILE loop, or
3) receipt of a motor quit command from the command buffer (section 3.2), which is used to properly terminate all parallel processes of each motor code.

In normal operation, the watchdog cycles through the WHILE loop, continually resetting the timer before it can expire. If the software code hangs up or receives a failure message, it cannot reset the timer, and upon timeout the motor ports are set to zero and motor power is disabled.

3.2 Command Buffer

The command buffer handles all communication with the transputer upstream from itself. It consists of an ALT statement repeated with a WHILE loop. The ALT waits for an incoming message from either the upstream transputer or from its report buffer (section 3.3). Messages from the upstream transputer to command one of that transputer's mechanisms are passed to the module controlling that mechanism; those meant for other mechanisms are passed to the downstream transputer. Messages from the report buffer are passed to the upstream transputer.

If the command buffer receives a quit reply from the report buffer, indicating that all modules in its code except itself have properly terminated, it sends this reply upstream and terminates, completely ending the program.

3.3 Report Buffer

The report buffer receives all reply messages from its mechanism modules (section 3.4) and from the downstream transputer. It is a WHILE looped ALT statement which waits for a message from any of these sources (which can be generated simultaneously from parallel processes) and passes them sequentially to the command buffer to be sent to the upstream transputer. This buffered architecture prevents parallel processes from trying to use the same communication channel simultaneously.

If the report buffer receives quit replies from each of its mechanism modules as well as its downstream transputer, it sends this information to the command buffer and then terminates itself.

3.4 Mechanism Modules

Each mechanism module performs all commands for one mechanism. The mechanism module simply consists of a WHILE loop (with the exception of the filter wheel modules; section 3.4.1) that receives commands for its mechanism from the command buffer and enters a CASE construction to perform them. These commands are described in section 4. Because each mechanism's control module operates independently in parallel, every mechanism can be run simultaneously. However, commands and replies must be passed sequentially through the transputer communication channels.

If a mechanism module receives a quit command, it promptly terminates the WHILE loop and returns a quit reply to the report buffer (section 3.3), thus ending the mechanism module part of the program.

3.4.1 Filter Wheel Modules

Because the filter wheel modules must communicate with each other, their construction is a bit different from the others. An ALT statement is placed within the main WHILE loop, and it waits for a command from either the command buffer or the other filter wheel module. These must be handled through separate communication channels to prevent the command buffer and the other filter wheel module from trying to use the same channel simultaneously. Each half of the ALT statement is constructed the same way as the other mechanism modules: a CASE statement divides the individual commands that are performed.

4 Description of Commands

Below is a description of each motor command. In parentheses are the command names used in the code.

4.1 Step Move (step)

This command is used to move a mechanism a desired number of motor half-steps. For clockwise moves, the command parameter contains the number of steps to be moved. For counterclockwise moves, the parameter contains the number of steps to be moved plus 999999. This is because negative numbers cannot be passed to the transputers (although they can be passed from them).

When the mechanism module receives this command, it decodes the parameter into either a positive (clockwise) or negative (counterclockwise) number of steps and calls either the *move* (section 5.1) or *move.limit* (5.2) procedure to send the proper pulses to the motor drive, depending on whether the mechanism has a limited range of motion. If the move was not aborted and no limit switches closed, the position counter is updated and a successful move reply is returned. In the case of an abort or a limit switch, the position counter is set to !1 and a reply is sent indicating which one occurred.

4.2 Position Move (pos)

A position move command tells the mechanism to go to a specific location (e.g. a certain filter or a specific grating angle). Each mechanism has its own position move command identifier, and the command parameter contains the desired position. Table 1 shows how the positions are defined for each mechanism as well as their valid ranges.

Note that positional moves cannot be performed unless the code knows the mechanism's location (i.e., the location counter must be greater than !1).

TABLE 1. Mechanism Position Definitions

Mechanism	**Positional Unit**	**Minimum Position**	**Maximum Position**
image rotator	0.01°	9000 (90°: TMA side)	27000 (270°)
filter wheel 1	filter number	3 (L')	11 (blank)
filter wheel 2	filter number	0 (NIRSPEC-1)	10 (K')
slit wheel	slit number	0 (1 pix HIRES slit)	11 (4 pix LORES slit)
echelle	0.01°	5000 (50.00°)	18200 (182.00°)
cross disperser	0.01°	0 (0.00°)	5800 (58.00°)
c.u. flip mirror	mirror position	0 (out of beam)	1 (in beam)
c.u. pinhole	N/A	N/A	N/A
c.u. cover	N/A	N/A	N/A

Most mechanisms have microswitches which close at specific positions (Table 2). The code looks at the status of these position switches where appropriate to ensure that the mechanism is moving correctly. The filter and slit wheels have a position switch that activates at every user position, so the code counts the number of times this switch closes during a move and compares this to the number expected. It also checks to see if the move ended with the switch closed. The echelle grating mechanism has a position switch located at the low resolution flat mirror position. The calibration unit mechanisms have position switches in both their "in" and "out" positions. The image rotator and cross disperser mechanism have no position switches.

Upon receipt of a position command, the code decodes it into a number of steps and a direction, and then calls the procedure *move* (section 5.1) or *move.limit* (5.2), depending on whether the mechanism has a limited range of travel. If the move was not aborted and no limit switches closed, the position counter is updated. Then, if the position switch operated as expected, a successful move reply is returned; if not, a position switch error is returned. and a successful move reply is returned. In the case of an abort or a limit switch, the position counter is set to !1 and a reply is sent indicating which one occurred.

TABLE 2. Mechanism Position Switch Locations

Mechanism	Where the Position Switch Closes
image rotator	N/A
filter wheel 1	every filter position
filter wheel 2	every filter position
slit wheel	every slit position
echelle grating	180° (LR flat position)
cross disperser	N/A
c.u. flip mirror	in the beam; out of the beam
c.u. pinhole	in the beam; out of the beam
c.u. cover	open; closed

4.2.1 Dual Filter Wheel

When a filter is selected, both wheels must move appropriately so that 1) the desired filter is put in the beam, 2) a Lyot stop and blocking filter are put in the beam with the other wheel, and 3) the wheels avoid putting two unfiltered positions into the beam at the same time. Thus, the occam code was written to control the filter wheels as a unit.

The filter position command is sent to the wheel that contains the desired filter. That wheel is moved to the new filter in a direction that avoids putting an unfiltered position through the beam. Then the other wheel is moved to its unfiltered position. The code does not allow the user to select an unfiltered position, or else it would be backed with an unfiltered position in the other wheel. The point of all of this is to keep at least one filter in the beam at all times.

Filter wheel #2 only has one unfiltered position, so command parameters for filter wheel #1 simply contain a filter position number between 3 and 11 (0 thru 2 are unfiltered). Filter wheel #1 has three unfiltered positions: a thick (3.5 mm) PK50 blocker, a thin (2.5mm) PK50 blocker, and no blocker. Command parameters to filter wheel #2 accommodate these options in the following way. Position numbers 0-10 send filter wheel #2 to that position and filter wheel #1 to the no-blocker position (position 11 is not allowed because it is unfiltered). Position numbers 12-22 send filter wheel #2 to positions 0-10, respectively, and filter wheel #1 to the thin blocker position. Likewise, numbers 24-34 select filters 0-10 and the thick blocker.

4.3 Initialization (init)

An initialization command is used to determine where the mechanisms are located. This is accomplished by moving the mechanism in search of its initialization switch, of which the code knows the exact location. The command parameter in this case is irrelevant. The reply parameter indicates the result of the initialization attempt (Appendix B).

When the code receives an initialization command, it immediately calls the procedure *initialize* (section 5.3) or *initialize.limit* (5.4), depending on whether the mechanism has a limited range of travel. After a successful initialization, the mechanism is jogged to the “home” position, which is usually the user position nearest to the initialization switch, the location counter is set to the home step number, and a reply is sent indicating the result of the initialization.

4.4 Position Set (initloc)

Each mechanism module contains a counter that keeps track of which step number (out of the total number of steps in the complete range of travel) the mechanism is currently on. A position set command is used to tell the mechanism where it is without having it seek its home switch. The command parameter contains the step number. Upon receipt of this command, the code sets the location counter to the value found in the command parameter, and replies with the same command and parameter.

4.5 Position Read (location)

A position read command is used to retrieve the current location of the mechanism. The parameter sent with the command is irrelevant. When the code receives this command, it simply replies with the same command i.d., with the step number in the command parameter. A value of !1 indicates that the current position is unknown.

4.6 Switch Read (switch)

This command is used to read the status of the microswitches in the mechanism. The status of each switch is contained in one bit of the 32-bit input port of the DAQ17 board. A bit value of 1 means the switch is open, and 0 means the switch is closed. Upon receipt of the switch read command, the code reads the four bits of the read latch corresponding to the four switches for that mechanism and converts the information to a four bit integer, which is sent back in the command parameter of the reply.

4.7 Abort (abort)

This command halts the sending of pulses to the motor driver to immediately stop the mechanism. Each mechanism has its own abort command identifier so that isolated aborts can be performed. The code is written such that no matter what it is currently doing, it can receive this command and stop performing its task. Upon aborting, it replies with the same abort command identifier. If a move or an initialization is taking place, the code will send replies of those commands indicating that they were aborted.

4.8 Track (track)

This command is only implemented for the image rotator mechanism. This is more of a mode than a command: in tracking mode, the mechanism is velocity controlled rather than position controlled. The command parameter contains the desired motor velocity in half-steps per second. Upon receipt of this command, the code enters tracking mode by running the procedure irot.track. A parameter of zero stops the motor and turns tracking mode off.

5 Procedures

5.1 move

The *move* procedure constructs stepping pulses for the motor drives and times their output to provide a linear acceleration to the cruising speed, a flat cruise, and a linear deceleration to stop at the desired position.

The following parameters are passed in the call of *move*.

write.latch = the hardware address to send the motor pulses
read.latch = the hardware address that contains the switch status
bit = which byte in the write.latch corresponds to the motor (0, 8, 16, or 24)
hilow = 0 for low motor current, 1 for high
pos.switch = the bit in the read.latch containing the position switch status
initial.speed = the starting speed in half-steps per second
final.speed = the final cruising speed in half-steps per second
ramp.steps = the number of steps used in the acceleration
steps = the total number of steps to move
steps.per.position = the number of steps between two adjacent user positions
interrupt = the channel to monitor for an abort command
reply = the channel used to send reply messages
switch.count = the number of times the position switch cycled

The procedure starts by constructing the stepping pulse, calculating how many steps to cruise at the final speed, and checking the status of the position switch. Four WHILE loops follow: one to accelerate the motor, one to cruise the motor, one to decelerate the motor, and one to slowly back up the motor. This mechanism back-up is used to help eliminate gear backlash. Clockwise moves always overshoot their mark by a certain amount and back into position so that moves always end going in the same direction.

During each WHILE loop, the code times when the next pulse should be sent. It then waits that amount of time in an ALT construction in order to listen for an abort command through the interrupt channel. An abort will make the code exit the WHILE statement and prevent it from entering any others. Any other command through the interrupt channel will be ignored and a reply will be sent indicating that the motor is busy.

If no commands are sent, the ALT sends the next stepping pulse at the required time and reads the position switch to see if its status has changed. If it changes from open to closed, switch.count is increased by one.

At the end of the procedure, if the move was aborted, switch.count is set to the reply parameter indicating an abort (Appendix B). The mechanism module that called the procedure then uses the value in switch.count to learn the result of the move attempt: either an abort or a number of position switch cycles.

5.2 move.limit

The following parameters are passed in the call of *move.limit*.

write.latch = the hardware address to send the motor pulses
read.latch = the hardware address that contains the switch status
bit = which byte in the write.latch corresponds to the motor (0, 8, 16, or 24)
hilow = 0 for low motor current, 1 for high
pos.switch = the bit in the read.latch containing the position switch status
limit1.switch = the bit in the read.latch containing the status of the clockwise limit switch
limit2.switch = the bit in the read.latch containing the status of the counterclockwise limit switch
initial.speed = the starting speed in half-steps per second
final.speed = the final cruising speed in half-steps per second
ramp.steps = the number of steps used in the acceleration
steps = the total number of steps to move
steps.per.position = the number of steps between two adjacent user positions
interrupt = the channel to monitor for an abort command
reply = the channel used to send reply messages
switch.count = the number of times the position switch cycled

Move.limit is identical to *move* except that after each stepping pulse it reads not only the position switch, but the limit switch for the direction it is going. If this switch closes, the code acts just as if an abort command were received: the current WHILE loop is exited, the remaining WHILE loops are not performed, and switch.count is set to the reply parameter that indicates a limit was reached (Appendix B).

5.3 initialize

The purpose of the initialize procedure is to scan the mechanism in search of its initialization switches to learn where it is located.

The following parameters are passed in the call of *initialize*.

write.latch = the hardware address to send the motor pulses
read.latch = the hardware address that contains the switch status
bit = which byte in the write.latch corresponds to the motor (0, 8, 16, or 24)
hilow = 0 for low motor current, 1 for high
pri.switch = the bit in the read.latch containing the status of the primary initialization switch
sec.switch = the bit in the read.latch containing the status of the secondary initialization switch
initial.speed = the starting speed in half-steps per second
final.speed = the final cruising speed in half-steps per second
ramp.steps = the number of steps used in the acceleration
total.steps = the total number of steps to move in search of the initialization switch(es)
interrupt = the channel to monitor for an abort command
reply = the channel used to send reply messages
result = the reply parameter indicating the result of the initialization attempt (Appendix B)

The procedure starts by checking the primary and secondary initialization switches. If the secondary switch is closed, it remembers by turning sec.switch TRUE. If the primary switch is closed, it moves the motor 1000 steps counterclockwise and checks the switch again to make sure it is open and not stuck closed (and also the secondary switch if it was initially closed). If both switches are still closed, it moves the motor to its original position and aborts the initialization attempt. If the primary switch is stuck but not the secondary switch, it begins the initialization routine using the secondary switch. If the primary switch is not stuck, it begins the initialization routine using the primary switch, just as if the primary switch was not closed initially.

The initialization routine consists of six WHILE loops nested inside of one giant WHILE loop. The six WHILE loops are used as follows:
1) accelerate the mechanism
2) cruise the mechanism
3) decelerate the mechanism if the switch has not been found yet
4) ensure the mechanism clears the switch if the switch was found during the previous deceleration
5) decelerate the mechanism if the switch has been found
6) back up the mechanism, stopping at the switch's center

The giant WHILE loop will repeat the routine in search of the secondary switch if the primary switch failed and the secondary switch was seen during the first search.

During each WHILE loop, the code times when the next pulse should be sent. It then waits that amount of time in an ALT construction in order to listen for an abort command through the interrupt channel. An abort will make the code exit the WHILE statement and prevent it from entering any others. Any other command through the interrupt channel will be ignored and a reply will be sent indicating that the motor is busy.

If no commands are sent, the ALT sends the next stepping pulse at the required time and reads the initialization switches. If the secondary switch is closed, it remembers by turning sec.search TRUE.

If the switch it is looking for closes, the search terminates and the motor is decelerated to a stop, making sure that the mechanism has completely passed the switch. During this switch-found deceleration, the code still monitors the initialization switch and counts how many half-steps the motor turns before the switch opens again. Then the mechanism reverses direction (counterclockwise) slow enough not to need an acceleration, and when it sees the switch close again it moves half of the total number of half-steps it measured for the switch closure. Thus, the mechanism comes to rest in the center of the switch's range of activation.

During the switch centering routine, the code is monitoring the switch to make sure it opens and closes as it should. If the switch does anything unexpected (e.g., the switch does not close again during the back-up WHILE loop), boolean variables are set to TRUE or FALSE. At the end of the procedure the code uses these variables to determine exactly what happened and report the proper result parameter.

The code sends the mechanism one complete revolution during the initialization routine, including the ramp up and ramp down, so that if no switches are found, it stops exactly where it began. If the search was for the primary switch and the secondary switch was seen during the search, the global WHILE loop repeats the procedure in search of the secondary switch.

At the end of the procedure, the report parameter of the procedure is set to the appropriate value (Appendix B). The mechanism module that called the procedure uses this value to learn the result of the initialization attempt, park the mechanism if it was successful, and pass the result back to the host computer (section 4.3).

5.4 initialize.limit

The following parameters are passed in the call of *initialize.limit*.

write.latch = the hardware address to send the motor pulses
read.latch = the hardware address that contains the switch status
bit = which byte in the write.latch corresponds to the motor (0, 8, 16, or 24)
hilow = 0 for low motor current, 1 for high
init.switch = the bit in the read.latch containing the status of the initialization switch
limit1.switch = the bit in the read.latch containing the status of the clockwise limit switch
limit2.switch = the bit in the read.latch containing the status of the counterclockwise limit switch
initial.speed = the starting speed in half-steps per second
final.speed = the final cruising speed in half-steps per second
ramp.steps = the number of steps used in the acceleration
total.steps = the total number of steps to move in search of the initialization switch(es)
interrupt = the channel to monitor for an abort command
reply = the channel used to send reply messages
result = the reply parameter indicating the result of the initialization attempt (Appendix B)

Initialize.limit is almost identical to *initialize* but has the following differences.
1) Secondary initialization switches are not implemented. Instead of going around again in search

of the secondary switch, the global WHILE loop is used to search in the reverse direction after a limit is reached.
2) The sixth WHILE loop (the one controlling the back-up switch centering) is outside of the global WHILE loop. This is because the global WHILE is only continuing the search for the same switch instead of repeating a whole search like in the initialize procedure.
3) After each stepping pulse, the code reads not only the initialization switch, but also the limit switch for the direction it is going. If the limit switch closes, the current WHILE loop is exited, the remaining WHILE loops are not performed, and the global WHILE loop repeats the search in the opposite direction.
4) The total number of motor half-steps moved in search of the initialization switch is large enough to move from one limit to the other. Thus, in the event that the initialization switch was not found, the mechanism will end up at the counterclockwise limit instead of at its original location.
5) The mechanism must always finish initializing in the same direction to reduce the effect of gear backlash. Thus, if the mechanism finds the switch and passes it while going in the final direction of approach (counterclockwise), the code pretends that it has not started looking for the switch yet and repeats the WHILE loop a third time. This search doesn't take long since it is already very near the switch.

5.5 irot.track

This procedure puts the mechanism into velocity control mode. In this mode, the code sits at an ALT statement, either sending pulses to the motor at the selected velocity or receiving incoming commands. Upon receipt of a position read or switch read, the code will perform these commands just as in the normal command mode. Upon the receipt of a track command, the code alters its current motor pulse rate to the rate in the command parameter. If the parameter is zero, it stops the motor and ends the procedure, sending the code back into the normal command mode. If the parameter is too large, the code continues sending pulses at its current rate and returns an error message back to the host. An abort command does exactly the same thing as a track command with a parameter of 0.

The following parameters are passed in the call of *irot.track.*

write.latch = the hardware address to send the motor pulses
read.latch = the hardware address that contains the switch status
bit = which byte in the write.latch corresponds to the motor (0, 8, 16, or 24)
hilow = 0 for low motor current, 1 for high
min.step = the lowest allowable step number for the mechanism
max.step = the highest allowable step for the mechanism
limit1.switch = the bit in the read.latch containing the status of the clockwise limit switch
limit2.switch = the bit in the read.latch containing the status of the counterclockwise limit switch
location = the location step counter of the mechanism
param = the motor speed in half-steps per second
message = the channel to monitor for new commands
reply = the channel to use to send reply messages

6 Modifying the Code

6.1 Motor Speeds

Motor moving characteristics are stored in the file */kroot/kss/nirspec/transputer/motor/motor.inc*. Their variable names are self-explanatory. Change the values as desired and recompile the code by changing directories to */kroot/kss/nirspec/transputer* and typing

```
compmot
```

6.2 Backlash Corrector

Clockwise motor moves overshoot their mark by a certain amount and arrive at the final position in the counterclockwise direction in order to remove the effect of gear backlash. This overshoot value is stored in the variable *backlash.steps* at the beginning of the *move* and *move.limit* procedures in the three motor transputer codes *motor1.occ*, *motor2.occ*, and *motor3.occ*.

To change this overshoot value, enter the directory */kroot/kss/nirspec/transputer/motor* and open the latest version of the file to be changed. Change the value of backlash.steps to the desired number of steps of overshoot. Test-compile this new version by typing

occam *file1 file2*

where *file1* is the name of the file that was modified (e.g., motor109), and *file2* is the name of the code to be replaced (e.g., motor1). Note that the file extension *.occ* should be left off. Now change directories down to */kroot/kss/nirspec/transputer* and recompile the code by typing

```
compmot
```

APPENDIX A: Command Identification Numbers

```
-- image rotator
VAL cid.mot.irot.step IS 150 (BYTE) :
VAL cid.mot.irot.pos IS 151 (BYTE) :
VAL cid.mot.irot.init IS 152 (BYTE) :
VAL cid.mot.irot.initloc IS 153 (BYTE) :
VAL cid.mot.irot.location IS 154 (BYTE) :
VAL cid.mot.irot.switch IS 155 (BYTE) :
VAL cid.mot.irot.abort IS 156 (BYTE) :
VAL cid.mot.irot.track IS 157 (BYTE) :

-- filter wheel 1
VAL cid.mot.filt1.step IS 160 (BYTE) :
VAL cid.mot.filt1.pos IS 161 (BYTE) :
VAL cid.mot.filt1.init IS 162 (BYTE) :
VAL cid.mot.filt1.initloc IS 163 (BYTE) :
VAL cid.mot.filt1.location IS 164 (BYTE) :
VAL cid.mot.filt1.switch IS 165 (BYTE) :
VAL cid.mot.filt1.abort IS 166 (BYTE) :

-- filter wheel 2
VAL cid.mot.filt2.step IS 170 (BYTE) :
VAL cid.mot.filt2.pos IS 171 (BYTE) :
VAL cid.mot.filt2.init IS 172 (BYTE) :
VAL cid.mot.filt2.initloc IS 173 (BYTE) :
VAL cid.mot.filt2.location IS 174 (BYTE) :
VAL cid.mot.filt2.switch IS 175 (BYTE) :
VAL cid.mot.filt2.abort IS 176 (BYTE) :

-- slit wheel
VAL cid.mot.slit.step IS 180 (BYTE) :
VAL cid.mot.slit.pos IS 181 (BYTE) :
VAL cid.mot.slit.init IS 182 (BYTE) :
VAL cid.mot.slit.initloc IS 183 (BYTE) :
VAL cid.mot.slit.location IS 184 (BYTE) :
VAL cid.mot.slit.switch IS 185 (BYTE) :
VAL cid.mot.slit.abort IS 186 (BYTE) :

-- echelle mechanism
VAL cid.mot.echl.step IS 190 (BYTE) :
VAL cid.mot.echl.pos IS 191 (BYTE) :
VAL cid.mot.echl.init IS 192 (BYTE) :
VAL cid.mot.echl.initloc IS 193 (BYTE) :
VAL cid.mot.echl.location IS 194 (BYTE) :
```

```
VAL cid.mot.echl.switch IS 195 (BYTE) :
VAL cid.mot.echl.abort IS 196 (BYTE) :

-- cross-disperser mechanism
VAL cid.mot.disp.step IS 200 (BYTE) :
VAL cid.mot.disp.pos IS 201 (BYTE) :
VAL cid.mot.disp.init IS 202 (BYTE) :
VAL cid.mot.disp.initloc IS 203 (BYTE) :
VAL cid.mot.disp.location IS 204 (BYTE) :
VAL cid.mot.disp.switch IS 205 (BYTE) :
VAL cid.mot.disp.abort IS 206 (BYTE) :

-- calibration unit flip mirror
VAL cid.mot.calm.step IS 210 (BYTE) :
VAL cid.mot.calm.pos IS 211 (BYTE) :
VAL cid.mot.calm.init IS 212 (BYTE) :
VAL cid.mot.calm.initloc IS 213 (BYTE) :
VAL cid.mot.calm.location IS 214 (BYTE) :
VAL cid.mot.calm.switch IS 215 (BYTE) :
VAL cid.mot.calm.abort IS 216 (BYTE) :

-- calibration unit pinhole
VAL cid.mot.calp.step IS 220 (BYTE) :
VAL cid.mot.calp.pos IS 221 (BYTE) :
VAL cid.mot.calp.init IS 222 (BYTE) :
VAL cid.mot.calp.initloc IS 223 (BYTE) :
VAL cid.mot.calp.location IS 224 (BYTE) :
VAL cid.mot.calp.switch IS 225 (BYTE) :
VAL cid.mot.calp.abort IS 226 (BYTE) :

-- calibration unit cover
VAL cid.mot.calc.step IS 230 (BYTE) :
VAL cid.mot.calc.pos IS 231 (BYTE) :
VAL cid.mot.calc.init IS 232 (BYTE) :
VAL cid.mot.calc.initloc IS 233 (BYTE) :
VAL cid.mot.calc.location IS 234 (BYTE) :
VAL cid.mot.calc.switch IS 235 (BYTE) :
VAL cid.mot.calc.abort IS 236 (BYTE) :

-- general status values for motors
VAL cid.mot.abort IS 240 (BYTE) :
VAL cid.mot.quit IS 241 (BYTE) :
VAL cid.mot.busy IS 242 (BYTE) :
VAL cid.mot.invalid.command IS 243 (BYTE) :
VAL cid.mot.invalid.command.parameter IS 244 (BYTE) :
```

APPENDIX B: Reply Parameters

```
VAL init.failed IS -6 :
VAL init.offset IS -5 :
VAL init.fail.no.switch IS -4 :
VAL init.fail.stuck.switch IS -3 :
VAL init.fail.interm.switch IS -2 :
VAL init.abort IS -1 :
VAL init.success.pri IS 1 :
VAL init.success.sec IS 2 :
VAL move.velocity IS -5 :
VAL move.pos.switch IS -4 :
VAL move.abort IS -3 :
VAL move.limit.switch IS -2 :
VAL move.pos.unknown IS -1 :
VAL move.success IS 1 :
VAL motor.generic.reply IS 0 :
VAL motor.busy IS -10 :
VAL motor.invalid.parameter IS -11 :
VAL motor.invalid.command IS -12 :
```