K1DM3
Control System
Software User's Guide

William Deich

University of California Observatories

07 Jan 2020
ver 3.7b

Contents

1 Overview **6**
- 1.1 Active Components of K1DM3 . . . 6
  - 1.1.1 Drum . . . 6
  - 1.1.2 Swingarm . . . 9
- 1.2 Position Summary . . . 13
- 1.3 Motion Controllers . . . 13
- 1.4 Software . . . 14
  - 1.4.1 Back-end Services . . . 14
  - 1.4.2 End-User Applications . . . 15
- 1.5 K1DM3 Private Network . . . 15

2 Starting, Stopping, Suspending Services **17**
- 2.1 Starting/Stopping . . . 17
- 2.2 Suspending Dispatchers . . . 19
- 2.3 Special No-Dock Engineering Version . . . 19
- 2.4 Periodic Restarts . . . 20

3 Components, Assemblies, and Sequencers **21**
- 3.1 Elementary Components . . . 21
- 3.2 Compound Keywords . . . 22
- 3.3 Assemblies . . . 22
- 3.4 Sequencers . . . 23
- 3.5 The ACTIVATE Sequencer . . . 24
- 3.6 The M3AGENT Sequencer . . . 27
- 3.7 Monitoring Sequencer Execution . . . 27

4 Dispatcher Interfaces for Routine Operations **29**
- 4.1 TCS Operations . . . 31
- 4.2 M3AGENT . . . 32
- 4.3 Hand Paddle . . . 33
  - 4.3.1 LED's . . . 34
  - 4.3.2 Buttons . . . 35

5 GUI's **36**
- 5.1 *Tcsgui* . . . 36
- 5.2 The Engineering GUI . . . 38
- 5.3 The *K1dm3mon* GUI . . . 40

6 Restrictions & Interlocks **41**
6.1 Introduction . . . 41
6.2 Bypassing the M3AGENT Sequencer . . . 42
6.3 Conditions Without Override . . . 43
6.3.1 Swingarm . . . 43
6.3.2 Drum . . . 44
6.4 Overridable Conditions . . . 44

7 Recovery From Failures **47**
7.1 General Troubleshooting Tips . . . 47
7.2 Low-level Actuator Operations . . . 48
7.3 Specific Issues . . . 49
7.3.1 Reconnect failure . . . 49
7.3.2 Amplifier fault . . . 50
7.3.3 Erroneous unclosed-clamps . . . 50
7.3.4 Actual unclosed clamps . . . 51
7.3.5 Air supply nozzle will not disconnect. . . . 52
7.3.6 Need to operate swingarm, mirror facing down . . . 52
7.3.7 Software lockout is active . . . 53
7.3.8 SASS: Swingarm past deploy point . . . 53
7.3.9 SASS: Swingarm in hard stop . . . 53
7.3.10 SASS: misaligned arms . . . 54
7.3.11 SASS: Arms misaligned *and* are past deploy point . . . 54
7.3.12 SASS: overshot V-blocks . . . 55
7.3.13 SASS: impossible encoder value . . . 55
7.3.14 Docking pin becomes stuck . . . 55
7.3.15 Docking pin position error . . . 56
7.3.16 In-tower signal failure . . . 56
7.4 Power Outage . . . 56
7.5 SASS Temporary Override Procedure . . . 58

8 Swingarm Safety System (SASS) **60**
8.1 Background . . . 60
8.2 The Safety Monitor and Outputs . . . 60
8.3 *k1dm3saf* . . . 62

9 The K1DM3 Monitor **64**

10 Galildisp Configuration **66**
10.1 Major Elements of Control . . . 66
10.1.1 Motor Axes and Digi-Axes . . . 66
10.1.2 Non-Axis Keywords . . . 67
10.1.3 Sequencers . . . 67
10.2 Dispatcher and Controller Information . . . 68
10.3 Configuration Files . . . 68
10.3.1 Template and Definition files . . . 69
10.3.2 The *.defs* and XML files . . . 69
10.3.3 The *.conf* files . . . 69
10.4 Custom Dispatcher Code . . . 70

11 Utility Programs **71**
11.1 *k1dm3.summarize* . . . 71
11.2 *k1dm3.status.email* . . . 74
11.3 *k1dm3.io* . . . 74

12 *k1dm3* KTL Service Keywords **75**
12.1 Dispatcher Keywords . . . 75
12.2 Controller Keywords . . . 77
12.3 Digi-Axis Keywords . . . 78
12.4 Motor-Axis Keywords . . . 79
12.5 Assembly (Combo-Axis) Keywords . . . 87
12.5.1 Motor-like Assemblies . . . 87
12.5.2 Digi-like Assemblies . . . 89
12.6 Status from Safety System . . . 91
12.7 TCS Interface Keywords . . . 91
12.8 Major Sequencer Keywords . . . 92
12.9 Elevation Keywords . . . 93
12.10Environment Keywords . . . 94
12.11Other Keywords . . . 95

13 *k1dm3saf* KTL Service Keywords **98**

14 *k1dm3mon* KTL Service Keywords **101**

A Useful Numbers **104**
A.1 Network Addresses . . . 104
A.2 Swingarm Safety System . . . 105
A.3 Swingarm Actuators . . . 106
A.4 Drum . . . 107
A.5 Typical Move Times . . . 108
A.6 Other Numbers . . . 108

B Actuator Calibration and Position Adjustment **109**
B.1 Swingarm Absolute Encoders . . . 109
B.2 Drum Readhead . . . 110

C Modifying *k1dm3saf* Galil Code **111**

D Modifying *k1dm3* Galil Code **114**
D.1 Introduction . . . 114
D.2 Downloading Code for *Galildisp* . . . 115

E How to do a Fresh Install **118**
E.1 Backups . . . 118
E.2 System Configuration . . . 118
E.3 Software Installation . . . 119

Bibliography **119**

Subject Index **121**

KTL Keyword Index **124**

Chapter 1

Overview

1.1 Active Components of K1DM3

The K1DM3 module is conceptually simple. It has just two degrees of freedom: the inner drum rotates to move the tertiary mirror to any instrument position (Nasmyth, "bent-Cassegrain," or Cassegrain orientation), and the swingarm moves the mirror in or out of the Cassegrain light path. This section summarizes all the components that are visible to the control system.

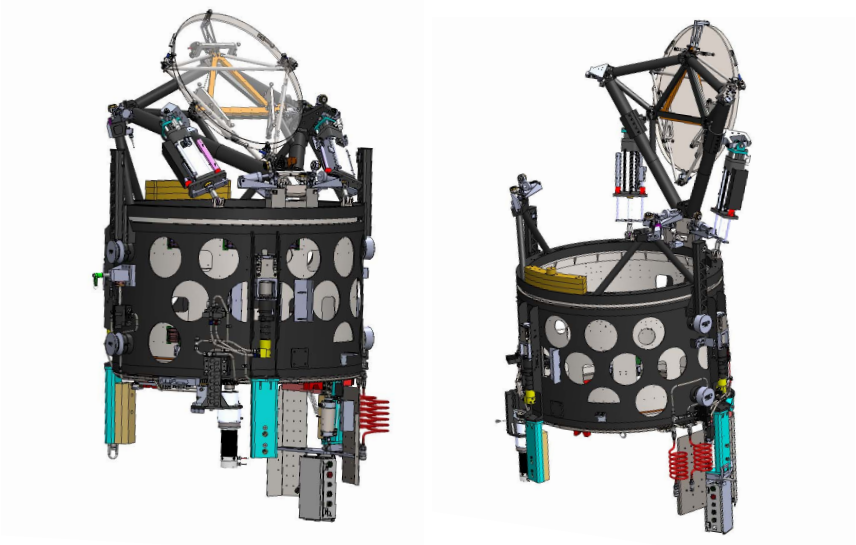


Figure 1.1: The K1DM3 Module, swingarm deployed (left) and retracted (right).

1.1.1 Drum

The drum's rotation is driven by a Magmotor C33-H-400 servomotor (fig 1.2). Accurate position feedback is provided by a Renishaw LM10 encoder system, using an encoder tape

with 5μ resolution mounted on the inner drum, and a readhead mounted on the outer drum. The tape has distance-coded reference marks, allowing the drum to home within $\sim$ 20 cm motion from any position. Additionally, there are limit switches at the Cass/Stow and MirrorUp (6- and 12-o'clock) positions. These switches provide the ability for the drum to be calibrated with no motion whatsoever at those two positions, which is important for bootstrapping the system from a cold start.

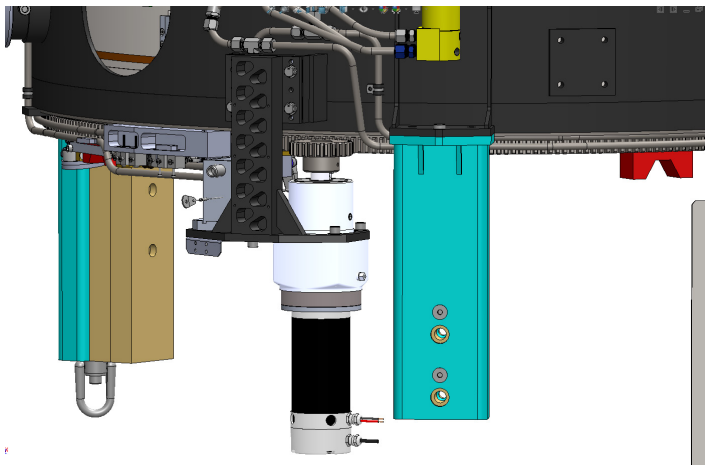


Figure 1.2: The drum rotation drive.

Each instrument position around the drum, plus the 6- and 12-o'clock positions, is defined by a detent. The detent is engaged by a detent mechanism after the motor drives the drum to the instrument position (fig 1.3, upper). The motor's drive gear is set up with a large amount of backlash — nearly 1900 motor encoder counts — between it and the drum's ring gear. This enables the option of finishing a move sequence with the motor backing off by $\sim$1/2 backlash, before engaging the detent mechanism to pull the drum into the final position. It turns out that the drive reliably positions the drum to 2 load encoder counts, or 10μ, and that the detent mechanism can easily pull the drum into final position regardless of whether the motor is engaging firmly with the gear ring, so the large backlash is not in fact required, and this optional end-of-move sequence is not implemented.

The detent mechanism uses a Bimba air cylinder that drives a steel wheel into or out of one of the position-defining V-blocks (fig 1.3, lower). If air pressure is lost, the detent will disengage. Two limit switches indicate when the detent mechanism fully engaged or disengaged.

The module is balanced when the swingarm is deployed and clamped in the V-blocks, but is imbalanced when the swingarm is retracted out of the light path. The imbalance is not a problem in the gravity-stable Cass/Stowed (mirror facing downwards) position, but can present a danger in the "MirrorUp" orientation. If the motor is unpowered, and the swingarm is not in a gravity-stable orientation, then the out-of-balance drum can rotate dangerously under gravity. Therefore, to ensure safe operation at all times, the swingarm may not be retracted except when (a) the drum is in the gravity-stable orientation, or (b) the drums are pinned together to prevent rotation.

In addition to rotating the inner drum, the drum control system is responsible for supplying compressed air, power, and networking to the inner drum's swingarm controls. The

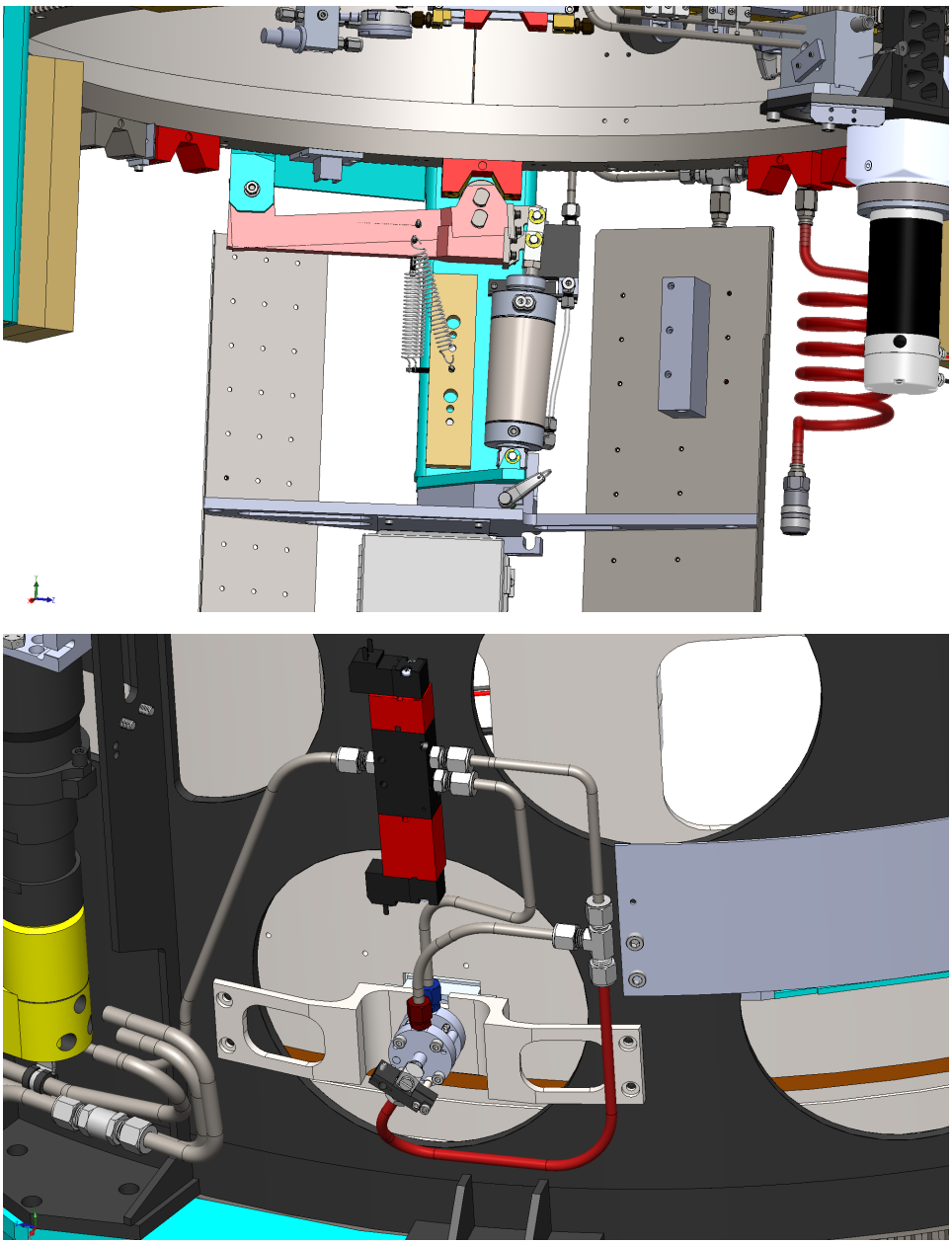


Figure 1.3: Upper:The detent mechanism (pink), showing its engaged and disengaged positions; and the position V-blocks (red). Lower: The air supply mechanism

compressed air is supplied through a nozzle that extends to engage with a receiver on the inner drum, or retracts when the drum must be rotated. Extension and retraction of the nozzle is controlled by an air cylinder. A limit switch indicates when the nozzle is retracted.

(The engaged position is affirmed by the presence of air pressure at the inner drum.) The nozzle can engage a receiver at either the 6- or 12-o'clock positions.

The inner drum receives power through a set of contacts and brushes that are akin to short segments of slip rings. Sets of contacts are installed at each detent position, but power to the inner drum is only available at the 6- and 12-o'clock positions.* At each position, there is a pair of contacts for each of the 48 V and 24 V supplies, four contacts for the 10 Mb/s Ethernet connection, and two contacts for an analog clamps-closed signal (described below).

1.1.2 Swingarm

The swingarm, mounted on the inner drum, is driven by two Exlar GSX40 linear actuators, called *Arm A* and *Arm B*. These actuators have incremental encoders with resolution approximately 0.31μ and backlash $\approx 60\mu$. They are supplemented by Renishaw LA-11 absolute encoders with resolution 0.24μ. The absolute encoders are mainly provided for an independent safety system (chapter 8) to continuously verify that the swingarm actuators are remaining in sync at all times, but are also used to initialize the incremental encoders.

The swingarm uses three canoe spheres that set into matching V-blocks for accurate positioning when the swingarm is deployed for use with Nasmyth and bent-Cass instruments. Normal canoe spheres are used where the swingarm rests in the bipod weldments. The third canoe sphere, located at the swingarm hinge, is split into two halves — one on each side of the pivot. The swingarm's hinge has a substantial amount of compliance, so that the hinge pivot places no force on the swingarm when it is resting in its neutral position in the V-blocks (figs 1.4 and 1.5).

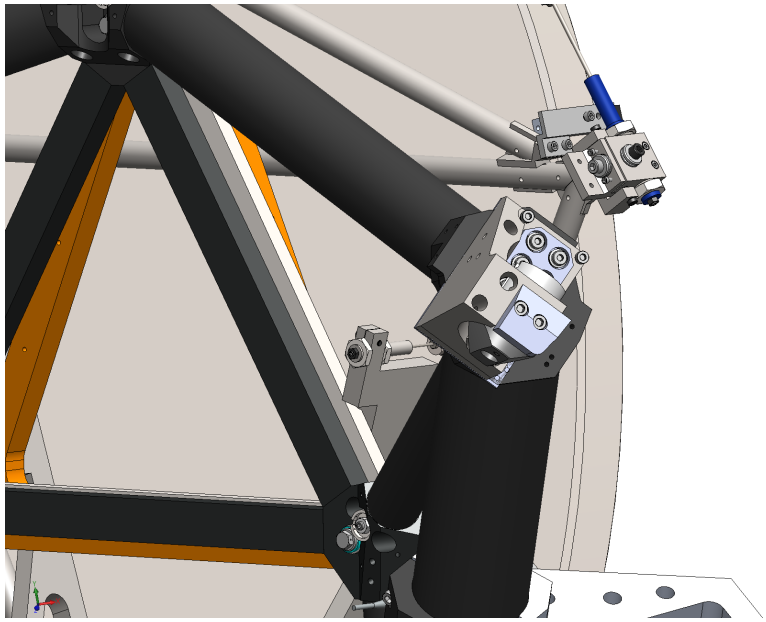


Figure 1.4: One of the swingarm's canoe spheres

*Full slip rings were evaluated, but deemed too costly.

When the swingarm is operated, the two actuators must stay in close alignment. This is achieved by configuring the control system to slave *Arm B* to track the instantaneously-commanded position of *Arm A*. To deploy the swingarm into the V-blocks, *Arm A* is commanded to move quickly until it is about 12 mm away from the V-block position, at which point it changes to a slow speed that will not cause damage as the spheres engage with the V-blocks. At the initial contact with the V-blocks, the compliant hinge is bearing a significant load, and the actuators continue to extend about 1 mm further to transfer the load entirely to the canoe spheres. The actuators have internal hard stops that have been adjusted so that they are only about 1 mm beyond the neutral point in the V-blocks. If the actuators were to somehow drive past the neutral point, they will hit the hard stops before damage occurs to the swingarm itself.

The swingarm is held in the V-blocks by four clamps (fig 1.6), each driven by a Bimba air cylinder. Clamps *A* and *B* clamp the corresponding actuators at the bipod V-blocks/canoe spheres; clamps *C1* and *C2* clamp the two halves of the hinge V-block/canoe sphere. Normally, the swingarm will not move unless the clamps indicate full-open, and the drum will not rotate unless it can confirm that the clamps are closed (the clamps are never closed unless the swingarm is in the V-blocks). Each clamp has a limit switch to indicate the fully-open position, which is monitored by the swingarm subsytem. A separate analog signal indicates which clamps are closed, and this is monitored by the drum subsystem, not the swingarm subsystem. This signal is only available when the drum is at one of the instrument positions, where there are contacts to electrically connect the inner and outer drums. When closed, each clamp produces a different analog voltage: clamps {*A,B,C1,C2*} generate signals of $\{1, 2, 4, 8\} \times 660\,mV$, respectively. The analog voltages are summed together to produce a single analog value that is decoded by the drum subsystem.

In the full-retracted position, the swingarm is secured by a dock mechanism that is mounted on the tertiary tower (fig 1.7). The purpose of the docking mechanism is to provide extra security in the event of a large earthquake. The swingarm has a tang that enters the docking mechanism, whereupon the dock controller drives a docking pin through a hole in the tang. The pin is driven by a Bimba air cylinder, and has limit switches to indicate the retracted and engaged positions. In normal operation, the swingarm will not move unless the docking mechanism indicates "Disengaged." A special engineering version of the control software (see section 2.3) allows the swingarm to be controlled without regard to the dock position.

The swingarm moves at a slow speed when approaching the fully-retracted position, just as it does when engaging the V-blocks. At both ends, the "slow" region is 40,000 motor encoder counts wide, and the slow speed is 3000 motor encoder counts/sec, so it takes about 13 seconds to move through a slow region. The "fast" region in between is 319,200 motor encoder counts wide, and the fast speed is 25,000 motor encoder counts/sec, so it also takes about 13 seconds to move through the fast region.* Typical move times are listed in the table in Appendix A.5.

In normal on-telescope operations, the swingarm is either in the `Deploy` position (for Nasmyth instruments), or the `Retract` position (Cassegrain instruments). When stored on the handling cart, the swingarm is normally left in the `Deploy` position. It is moved to

*The width of 40,000 counts is far larger than required for safe operation. It was initially chosen because it was considered just large enough for a staff member with a hand paddle to press the E-Stop button if the swingarm was not going to stop on its own. Now that the safety system has been thoroughly proven, it would be safe to narrow the regions to about 10,000 counts, which would save about 10 seconds per deploy or retract.

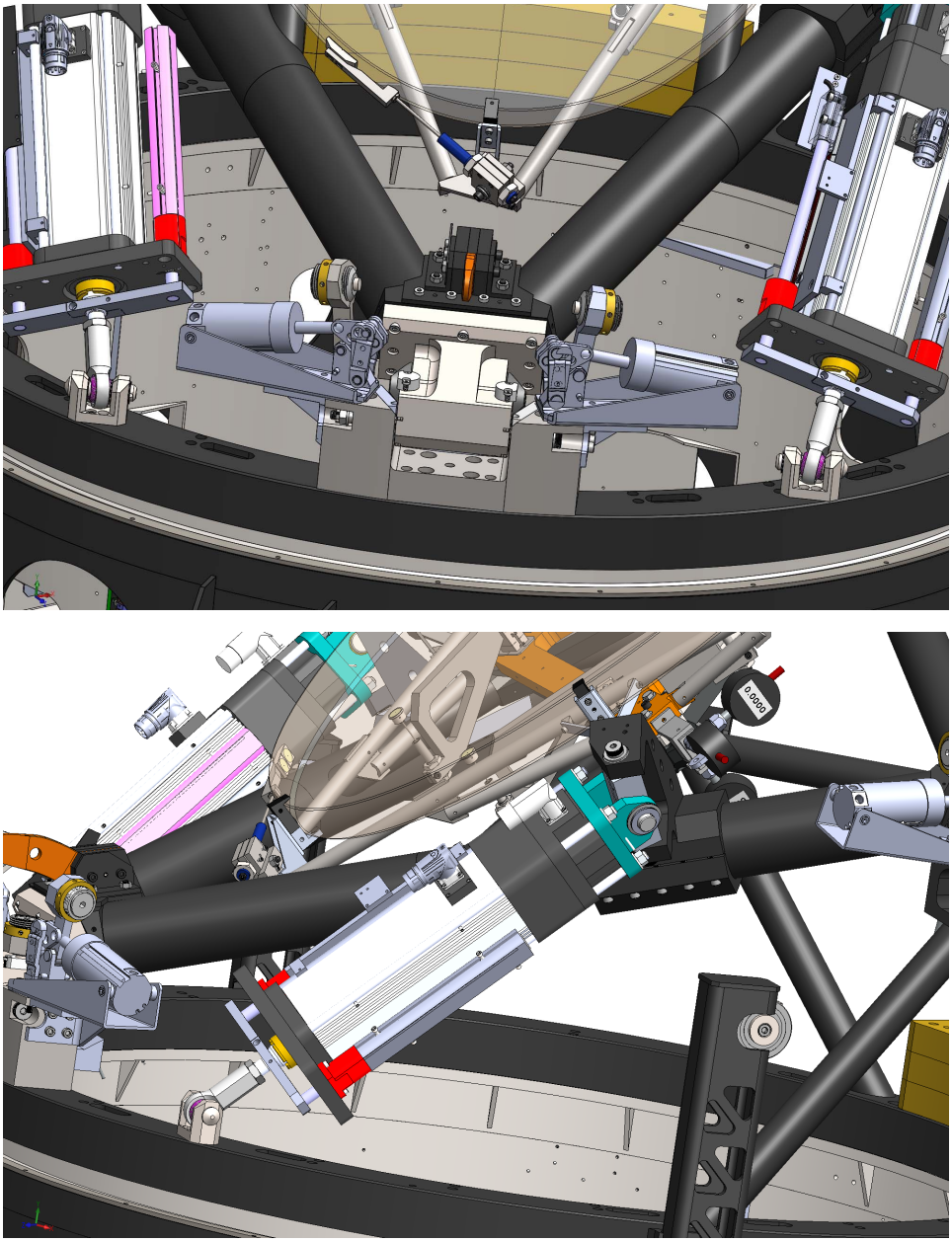


Figure 1.5: The "split" V-block at the swingarm hinge, with its two clamps open (upper), and a side view of an actuator.

`Mirror90`, meaning the mirror is at 90° from the X-Y plane, for mirror removal/mounting. There are also two additional named swingarm positions, to help applications recognize important deviations from the nominal positions: `Deployed,Unclamped` is used to indicate that the swingarm is correctly positioned in the V-blocks, but one or more clamps is not

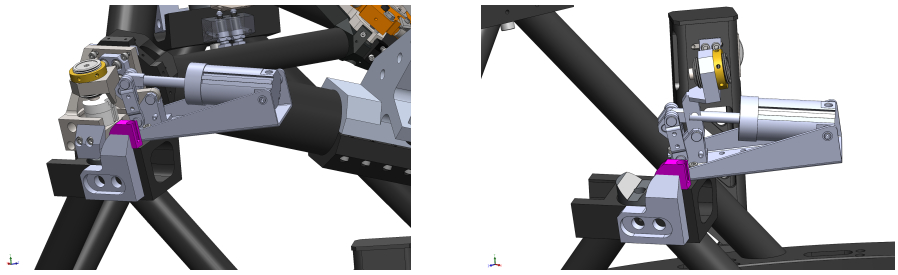


Figure 1.6: Clamp closed on swingarm (left), and clamp open (right).

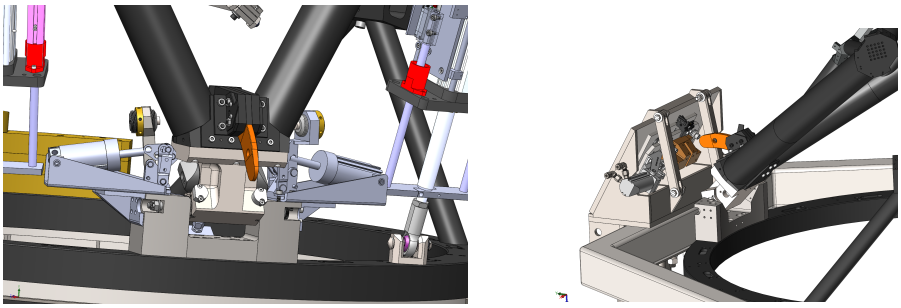


Figure 1.7: The docking tang (left), and the tang about to enter the dock slot (right).

closed. `NotFullyDeployedButClamped` is used to indicate that the swingarm is not within tolerance of the full in-V-block `Deploy` position, but the clamps are closed.

As noted above, swingarm retraction is only done with the drum at the mirror-facing-down orientation. Furthermore, the swingarm is not permitted to be released from the clamps unless the telescope elevation is in the range 45° – 75°. If the clamps were to open when the module is too close to horizontal, the compliant hinge would allow the swingarm to rotate (fall) several degrees under gravity before the compliance is used up and the swingarm would come to an abrupt stop. When the module is pointed at zenith, sag in the system prevents the docking mechanism from engaging.*

Swingarm deployment is also affected by the compliance in the hinge. When the telescope is at a very low elevation, moving the swingarm actuators to the normal deploy position doesn't actually move the swingarm all the way into the V-blocks, and therefore the clamps can't engage properly. In normal observing operations, the swingarm is only deployed when the telescope is at 67°. The same elevation is used for all mirror deployments to minimize variation of the swingarm position as it sets in the V-blocks. The value of 67° was chosen because it minimizes the average telescope elevation travel time for typical observing targets.†

*When retracting the swingarm, the sequencer will wait up to `EL_MAXWAIT` seconds for the telescope elevation to be in the range 45°–75, else it aborts the requested move. The upper angle is user-selectable, and stored in the writable keyword `EL_RETRACT`.

†The actual deployment elevation is user-selectable, and stored in the writable keyword `EL_DEPLOY`. However, the swingarm's positioning repeatability has not been characterized any elevation other than 67°, which should be used for all science observations. As with retraction, the sequencer will wait up to `EL_MAXWAIT` seconds for the elevation to reach this position, else it aborts the requested move.

The compliance in the hinge restricts safe rotation conditions as well as safe retraction conditions. The basic rule is that rotation is forbidden whenever the swingarm is not clamped in the V-blocks.

1.2 Position Summary

K1DM3 can be commanded to the following named positions:

- *Instrument Positions:* `LNas`, `RNas`, `LBC1`, `RBC1`, `LBC2`, `RBC2`. In each of these positions, the swingarm is deployed and clamped in the V-blocks, and rotated to the corresponding instrument position.

  Drum rotation among the above positions and/or the `Stow` position can be done at any telescope elevation. If the module is starting at the Cassegrain (swingarm retracted) position, the telescope elevation should be 67° for deployment.

- *Instrument Position:* `Cass`. In the Cassegrain position, the swingarm is retracted out of the light path.

  When retracting the swingarm, the telescope elevation should be in the range 45°–75°.

- *Supplementary position:* `Stowed`. The `Stowed` position is with the mirror deployed and clamped in the V-blocks, facing downwards. This is the same drum orientation that is used for retracting and deploying the swingarm.

  When moving to `Stowed` from the Cassegrain position, the telescope elevation should be 67°. Otherwise, moving to `Stowed` can be done at any telescope elevation.

- *Supplementary positions:* `MirrorUp`, `Mirror90`. These positions are intended for use on the handling cart. The `MirrorUp` position also has the mirror deployed and clamped in the V-blocks, but the drum is rotated 180 degrees from the mirror-down position. The `Mirror90` position is for use on the handling cart: the mirror is facing up and is horizontal, suitable for removing the mirror for re-coating.

  The `MirrorUp` and `Mirror90` positions are unbalanced against gravity, and the software will reject requests to move to these positions unless the outer and inner drums are pinned together.

In normal use, the above named positions are the only positions to which the module is sent. For engineering purposes, however, the drum and swingarm can be commanded to any position in their range of motion.

1.3 Motion Controllers

As noted in the previous section K1DM3 is conceptually simple. The module has just three motors and seven solenoid-controlled devices, which in principle could have been operated with a single multi-axis Galil controller. The actual system uses four Galil controllers, which is primarily due to unique issues of this system.

First, there is very limited room for routing cables between the fixed outer drum and the rotating inner drum that carries the swingarm, so that it is impractical to route dozens of motor power, encoder signals, solenoid controls and limit signals. Instead, a Galil DMC-4040

is mounted on the inner drum, and only power and a network cable are needed to connect them. In addition to the Galil control, one analog signal is provided for the outer drum to directly monitor whether the swingarm is clamped in the V-blocks. The tradeoff is that the Galil must be powered down at the end of every move, so that it doesn't continuously dump ∼5W of heat into the light path.

Second, system safety considerations (see chapter 8) led us to implement an independent, standalone, swingarm safety system (SASS) which runs on its own Galil RIO-47142, also mounted on the inner drum. The SASS reads the swingarm's Renishaw absolute encoders, and triggers the swingarm DMC-4040's Abort and Electronic LockOut (ELO) signals if the arms get out of sync, or if they are moving too fast as they reach end of travel. By using a standalone system running a small program (< 100 lines of Galil code), we gain high confidence that the safety system will execute correctly.

The complete set of controllers is as follows:

- Drum rotation is controlled by a Galil DMC-4040 that is mounted in the electronics box, beneath the primary mirror. This Galil is also responsible for controlling the air supply and power supply to the swingarm.
- The swingarm assembly, with two linear actuators and four clamps, is controlled by a Galil DMC-4040 that is mounted inside the inner drum.
- The swingarm safety system runs on a Galil RIO-47142, mounted on the inner drum.
- The swingarm's docking mechanism, mounted on the tertiary tower, is controlled by another Galil RIO-47142, installed in the electronics box below the primary mirror. (The dock was added late in the K1DM3 development cycle, long after the other electronics had been designed and built. It would have been costly and caused a significant schedule delay to have modified the existing electronics and cabling to accommodate the dock signals, whereas adding an additional Galil RIO was straightforward.)

The network connections between the motion controllers and the wider WMKO network are discussed in Section 1.5.

1.4 Software

1.4.1 Back-end Services

Each of the drum, swingarm, and dock controllers is managed by a separate instance of *galildisp*, an application for operating Galil controllers; their individual behaviors are set by configuration files, plus some custom code as needed. These *dispatchers* collectively make up the *k1dm3* KTL service, which provides all control for the K1DM3 mechanisms.

The safety system Galil carries out its function in a fully standalone manner — no external software daemon is needed for it to operate correctly. It sends safety status information plus the swingarm actuators' absolute encoder values to a UDP port on the instrument host, so that this information can be available to other clients.

The K1DM3 software includes these KTL services:

- *k1dm3*. This is the primary service for K1DM3, and provides all control for the K1DM3 mechanisms. There is a separate dispatcher daemon for each of the drum, swingarm, and dock Galil's, with each dispatcher serving part of the *k1dm3* keywords.

- *k1dm3saf*. This service monitors the the information sent by the safety system Galil RIO to the instrument host, and provides two functions: First, it makes the safety system status visible to KTL clients. Second, because the safety system reads the absolute encoders mounted on the swingarm actuators, it is used to initialize the swingarm actuators' incremental encoders each time the swingarm is activated.

- *k1dm3mon*. This service uses the *emir* application to provide a supplementary K1DM3 system monitor. It raises alerts under a variety of circumstances (e.g. a motor staying on for over 3 minutes is probably generating unwanted heat), and will also try to take safety actions that are beyond the scope of the *k1dm3* dispatchers (e.g. command the air supply to disengage if the detent is detected prematurely disengaging).

1.4.2 End-User Applications

Users primarily interact with K1DM3 through a control row in *Tcsgui*. There are also two K1DM3-specific applications: the engineering gui, *k1dm3_gui*, and a gui that displays the status of the supplementary *k1dm3mon* monitor service.

1.5 K1DM3 Private Network

As shown in figure 1.8, the instrument host, *k1dm3server*, connects to all the controlled components via a Ubiquiti EdgeRouter X in the electronics box. The 5 ports on the EdgeRouter X are configured as a simple switch.

The K1DM3 networked components use the private network 192.168.23.0/24.

The drum control Galil DMC-4040 and the dock control RIO-47142 are both inside the E-box, and are directly connnected to the EdgeRouter X. The swingarm control Galil DMC-4040 is mounted on the inner drum, and is connected to the Ubiquiti via a network cable that goes across the brush contacts. These brush contacts are not defined by any network standard, of course. They were verified through extensive testing to give trouble-free communications at 10 Mb/s, and communications with the inner drum are therefore locked down at 10Mb/s to ensure robust communications. The swingarm DMC-4040's two network ports act as a mini-switch, and the safety RIO-47142 is connected to the network by daisy-chaining through the second port on the DMC-4040.

The instrument host is dual-homed, with one interface (`enp3s0f0`) on the Keck operations network and the other interface (`enp3s0f1`) on the K1DM3 private network.

The full set of K1DM3 network addresses are listed in Appendix A.

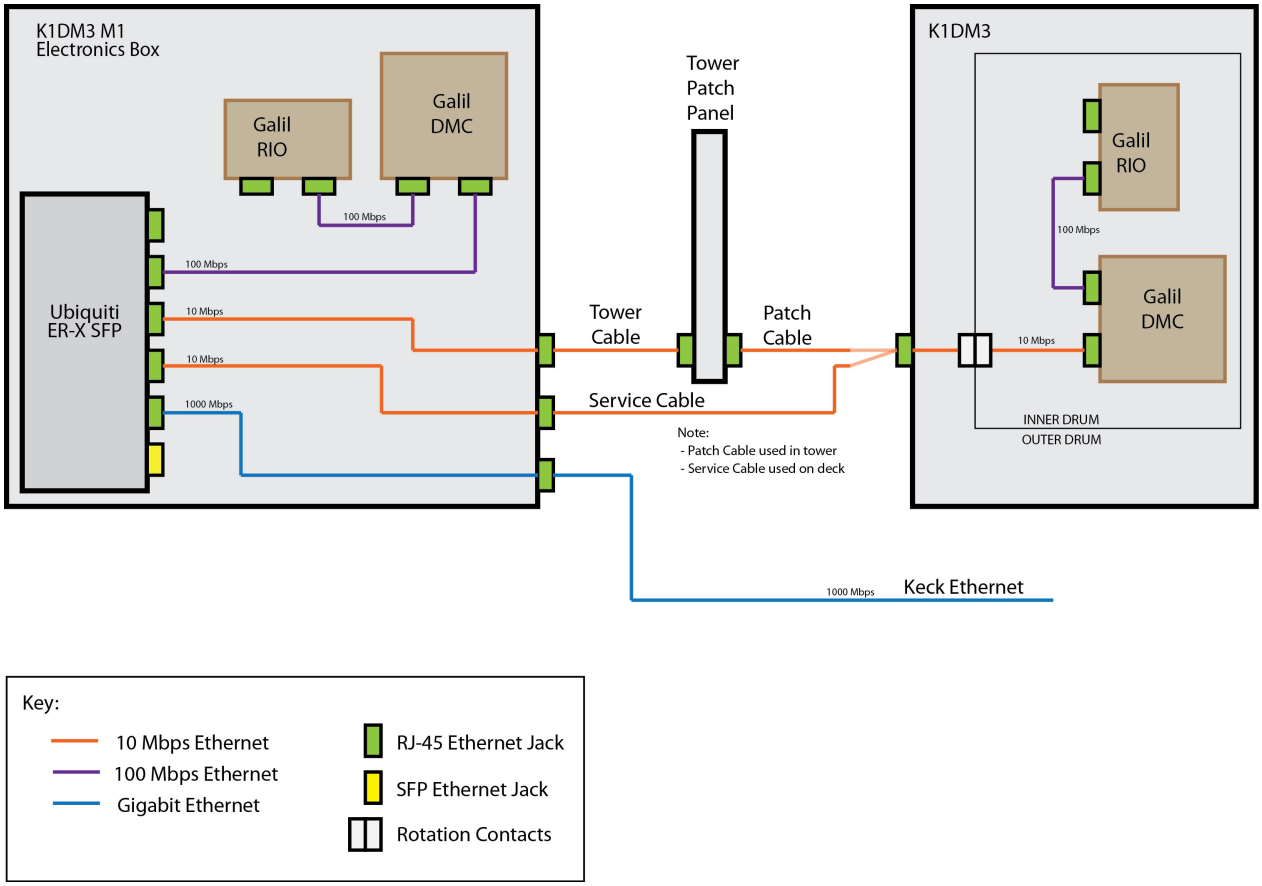


Figure 1.8: Network connections for K1DM3 Components. The Ubiquiti ER-X SFP is configured as a switch. The instrument host is connected to the blue line labelled "Keck Ethernet," which is on the VLAN 192.168.23.0/24.

Chapter 2

Starting, Stopping, Suspending Services

2.1 Starting/Stopping

K1DM3 services are normally started and stopped using the master command `$RELDIR/bin/k1dm3` on the instrument host, `k1dm3server`. The `k1dm3` command can be run by any account belonging to the `instr`, `k1dm3`, or `tcs` Unix groups, and it will always start services as the `k1dm3run` user. Invoked with no arguments (or with `-h`) you will see a list of choices like so:

```
    Use:
    k1dm3 [opts]  [help|start|restart|stop|status|pid|config]  subsystem...
where
     Subsystem               Description                 Invokes
   -----------               -----------               ---------------

   motioncontrollers             Motion control daemons    drum, dock, swingarm
   dispatchers or daemons        All K1DM3 dispatchers     rotate, dock,
                                                           swingarm, safety,
                                                           monitors
   swingarm, disp1 or dmc1       DMC-40x0 (swingarm)       k1dm3_dmc1
   rotate, drum, disp2 or dmc2   DMC-40x0 (rotation)       k1dm3_dmc2
   dock or disp3                 RIO (dock)                k1dm3_dock
   safety or k1dm3saf            K1DM3 Safety System RIO   k1dm3saf_rio
   monitors                      Monitor daemons           k1dm3mon1, k1dm3mon2
   k1dm3mon1 or monitor1         K1DM3 Main Monitor        k1dm3mon1.sh
   k1dm3mon2 or monitor2         K1DM3 Heartbeat Monitor   k1dm3mon2.sh
   k1dm3history or keygrabber    K1DM3 keyword history     k1dm3history(*)

* = runs on: vm-history-1.keck.hawaii.edu
```

Example 1. You can check the run status by using:

```
k1dm3 status
```

Example 2. You can start all dispatchers using:

```
k1dm3 start dispatchers
```

Example 3. You can restart just the swingarm dispatcher using:

```
k1dm3 restart swingarm or
k1dm3 restart dmc1
```

The dispatchers will not start correctly if the corresponding Galil is not online.* The start/stop script rejects attempts to start or stop the dispatcher if the Galil is not pingable (stopping the dispatcher is rejected on the assumption that your next action will be to restart it). You can override the rejection by appending "@" to the action to get an alternate version, e.g.

```
k1dm3 restart@ swingarm or
k1dm3 stop@ swingarm
```

The alternate versions for all drum and dock actions simply skip the ping check. The alternate version for starting the swingarm dispatcher is much more extensive. If the drum Galil is pingable, the startup sequence will unsuspend the drum dispatcher if it's suspended, turn on 24V power to the swingarm Galil if it's off, start the swingarm dispatcher, and finally restore the 24V power and drum dispatcher status to their prior state.

A second alternate version, run by appending "@@" to the action, adds another layer of smarts. It automatically selects between the normal and alt versions, depending on whether the target Galil is pingable. It is provided for use by applications such as the engineering gui (see its `Restart` menu, in section 5.2) that need a simple way to automatically invoke the "best" way to restart a dispatcher.

The rotation and dock dispatchers can be started about 60 seconds after the E-box power has been turned on, which allows time for the switch inside the E-box to boot up.

The swingarm dispatcher requires more care to start. The Galil will be pingable about 2-3 seconds after the inner drum's 24V supply is connected and on. Power to the inner drum is only available when the drum is at the 6- or 12-o'clock position, and is controlled by the rotation dispatcher's `INNER24V` keyword. Therefore the rotation dispatcher must be running and the drum homed so that the dispatcher knows the drum position. These steps are handled by the alternate swingarm startup version.

Each of the names listed under "`Invokes`," above, is a SysV-style start/stop script, which are found in the directory `$RELDIR/etc/init.d/`. The start/stop scripts are:

- `k1dm3_dmc1`. Dispatcher #1 for service *k1dm3*. Controls the Galil DMC-4040 that operates the inner drum (swingarm).
- `k1dm3_dmc2`. Dispatcher #2 for service *k1dm3*. Controls the Galil DMC-4040 that operates the rotation mechanism and hand paddle.
- `k1dm3_dock`. Dispatcher #3 for service *k1dm3*. Controls the Galil RIO-47142 that operates the swingarm dock mechanism.
- `k1dm3saf_rio`. Dispatcher for service *k1dm3saf*. Monitors the standalone Swingarm Safety System Galil RIO-47142.
- `k1dm3mon1.sh`. Primary dispatcher for service *k1dm3mon*. Monitors *k1dm3* and raises alarms.

*Exception: the swingarm safety service can be started regardless of the state of the safety Galil.

`k1dm3mon2.sh`. Dispatcher to monitor heartbeak keywords for service *k1dm3mon*.

`k1dm3history`. This controls a *keygrabber* instance (Lanclos and Deich, 2012) that keeps a complete history of keywords from the above K1DM3 services. At this writing, this service runs on the host `vm-history-1`.

The master start/stop command is a small shell script that invokes the general-purpose master script `$RELDIR/etc/init.d/lickStartStop` with the K1DM3-specific configuration file `$RELDIR/data/K1DM3/init/lickStartStop.conf`. Each of the individual dispatcher start/stop commands is a small shell script that invokes the general-purpose SysV script `$RELDIR/etc/init.d/lickSysV`, driven by the configuration file `$RELDIR/data/K1DM3/init/std_script.conf`.

2.2 Suspending Dispatchers

Historically, the dispatchers have regarded a non-responsive Galil as a severe error. However, K1DM3 operations require the swingarm Galil DMC-4040 to be powered off at all times except when actively using the swingarm. To support this, the dispatchers can be commanded to enter a special "suspended" state, using the keyword `DISP`x`SUSP` (here, x is the dispatcher number). In the suspended state, the dispatcher will do little more than wait to be un-suspended, and most *show* and *modify* commands will fail with the error "Dispatcher operations are suspended."

During K1DM3 development, it was common for the technical staff to casually shut off power to the control electronics without first suspending the dispatchers. To manage this situation gracefully, the *galildisp* dispatcher has been modified to "auto-suspend" itself if the Galil becomes non-pingable. While auto-suspended, the dispatcher will ping the Galil occasionally, and when the Galil responds to the ping, the dispatcher will reconnect and auto-unsuspend.

If you are going to shut off power to the entire electronics box — say, to remove it from the tower — it is recommended that you first suspend the dispatchers. The recommended method is to use the `ACTIVATE` keyword, which configures the system for different modes of use:

```
modify -s k1dm3 ACTIVATE=Transport
```

In the `transport` setting, the system is ready for transport in or out of the tertiary tower. In addition to suspending the dispatchers, the `ACTIVATE` sequencer will verify that the clamps are closed, the detent is engaged, the drum is oriented to Cass/Stow, the drums are pinned together, and all power supplies are off (except for the 120V supply to the electronics box itself).

2.3 Special No-Dock Engineering Version

In all normal operations, the swingarm assembly (section 3.3) — which is composed of the two swingarm actuators, the clamps subassembly, and the dock mechanism — commands the docking mechanism to open before commanding any actuator motion, and the actuator components refuse to move whenever the dock signal isn't "Disengaged"; this cannot be overridden with an ordinary safety bypass.

We found that during installation of the dock, it was necessary to be able to move the swingarm regardless of the dock state, and therefore we provided a special engineering version of the control software that allows the swingarm to be operated without regard to the dock mechanism: the dock mechanism is not included in the Swingarm assembly (so that problems with the dock will not be inherited by the Swingarm); the Swingarm will not command the dock mechanism; and the swingarm actuators will pay no attention to the state of the dock.

The dock dispatcher itself is entirely unaffected by this change, and you can monitor and command the mechanism using the normal `DOCK`*`xxx`* keywords.

> ⇒ In the swingarm-without-dock engineering version, all safe movement of the swingarm at the dock depends on personnel taking care to command the swingarm only when the dock mechanism is not in the way. There is *no* software safety.
>
> ⇒ The keyword `WITHDOCK` is true (false) if the swingarm assembly is (is not) configured to monitor and control the dock mechanism.

The following steps are required to remove the dock mechanism from Swingarm monitor and control:

1. Go to `svn/kroot/kss/K1DM3/k1dm3/configs`
2. Edit `Mk.dockoption`, and set `WITHDOCK=`
   (That is, set `WITHDOCK` to the empty string.)
3. Do: `make install`, `kdeploy`
4. Restart the rotation dispatcher: `k1dm3 restart rotate`

To restore the dock mechanism to normal Swingarm monitor and control, use similar steps as above, except set `WITHDOCK=1`:

1. Go to `svn/kroot/kss/K1DM3/k1dm3/configs`
2. Edit `Mk.dockoption`, and set `WITHDOCK=1`.
3. Do: `make install`, `kdeploy`
4. Restart the rotation dispatcher: `k1dm3 restart rotate`

2.4 Periodic Restarts

The K1DM3 *galildisp* instances have a memory leak, whose cause remains unknown. We recommend restarting the dispatchers monthly. Take care to restart the swingarm dispatcher only when the module is at the Cass/Stow or Retract positions:

1. `k1dm3 restart drum` *(*Wait 5 seconds)
2. `k1dm3 restart dock` *(*Wait 5 seconds)
3. `k1dm3 restart@@ swingarm`

Chapter 3

Components, Assemblies, and Sequencers

This section discusses how the *k1dm3* KTL service's basic components are combined in software into higher-level *assemblies* and controlled via *sequencers*.

Galildisp provides several kinds of elementary components for device control: (a) basic digital I/O bits; (b) simple digital stages that are controlled and monitored by a small group of I/O bits; and (c) motor stages.

Galildisp also provides *compound keywords*, which are keywords whose values are composed of other keywords and/or I/O values, and *assemblies*, which are logical stages composed of other stages and/or sub-assemblies.

3.1 Elementary Components

The three elementary components for device control are:

Simple I/O A simple digital I/O component generally has just one keyword associated with it, showing its current value. A simple analog input component will often have two keywords: one keyword will show the "raw" voltage read by the Galil, and a second keyword to show the value after conversion to a physically meaningful value: perhaps a temperature, or a power supply voltage, *etc.*

Digi-stage A "digital" stage, or just digi-stage, is controlled and monitored by a collection of I/O bits. For example, a solenoid-driven actuator might have two inputs from limit switches that indicate the actuator is in the open or closed position, and might have one output bit to select the direction of motion, and another to send a pulse to the solenoid.

A digi-stage has a standard set of 15-20 keywords, depending on configuration options. Each standard keyword has a name of the form ***stage***`XXX`, where ***stage*** is the name of the stage, and `XXX` is a 3-letter standard suffix, such as `POS` for position, or `XMV` for "can't-move" reasons. See section 12.3 for a complete listing of *k1dm3*'s digi-stage keywords.

Motor stage A motor stage is a Galil controller's bread-and-butter. Each motor stage has a set of 50–65 keywords, with the exact number depending on the stage configuration. As with digi-stages, each motor keyword has a name of the form *stage**XXX***. See section 12.4 for a complete listing of *k1dm3*'s motor keywords.

3.2 Compound Keywords

Compound keywords are defined in terms of other keywords, plus they can directly use digital and analog I/O levels. For writable compound keywords, the "modify" rules can either be one or more modify commands embedded in the KTL service configuration files, or they can be implemented as Tcl procedures, following the standard *galildisp* modify-API. Of particular note, *galildisp* includes a modify procedure that provides a robust interface to external scripts acting as sequencers (section 3.4), enabling complex sequencers to be readily triggered from the configuration data.

An example of a simple compound keyword is `PINNED`. This is a mask keyword with fields `CART` and `DRUMS`, which show the combined state of the input bits `CART_PINNED` and `DRUMS_PINNED`.

Compound keywords can be arbitrarily complex. For example, the `ACTIVATE` compound keyword implements the sequencing necessary to alternate between swingarm motion and drum rotation. These long, involved sequencer scripts are invoked as modify procedures from the keyword configuration.

3.3 Assemblies

Galildisp groups individually-controlled components into *assemblies*. An assembly is a software grouping of elementary components and/or other sub-assemblies. The assembly layer adds a set of keywords that treat the components as a single logical device. As with compound keywords, the "modify" rules for assemblies can either be one or more modify commands embedded in the KTL service configuration files, or they can be implemented as external scripts (sequencers), enabling arbitrarily complex rules.

Galildisp provides support for several kinds of assemblies. Broadly speaking, an assembly either acts like a digi-stage or a motor stage.

K1DM3 has these core assemblies:

- `CLAMPS`. The four swingarm clamps are almost always operated as a unit, using the `CLAMPSPOS` keyword to open or close the clamps. The `CLAMPS` assembly is made up of the four actual clamps, each of which is implemented as a digi-stage. The four digi-stages are `CLAMP_A` and `CLAMP_B`, which clamp the swingarm at the bipod weldment; and `CLAMP_C1` and `CLAMP_C2`, which clamp the swingarm at the hinge. When commanding the clamps, we usually want to command A and B simultaneously, and $C1$ and $C2$ simultaneously, which can't quite be achieved using the separate digi-stage keywords. The *k1dm3* service provides custom code, accessed by the compound keywords `CLAMP_AB` and `CLAMP_C1C2`, each of which issues Galil commands to operate a pair of clamps simultaneously. The `CLAMPS` assembly uses those custom keywords instead of the individual digi-stage keywords.

- `SWINGARM`. The `SWINGARM` is composed of the two swingarm actuators (`ARM_A_` and `ARM_B_`) the clamps (the `CLAMPS` assembly), and the docking mechanism (the `DOCK`

digi-stage). Although both actuators' keywords are part of the Swingarm assembly, in practice Actuator B is normally slaved to Actuator A, and is not directly commanded, except when tweaking the alignment between the two actuators.

- `DRUM`. The `DRUM` assembly is made up of the air supply mechanism (the `AIRSUPP` digi-stage), the detent mechanism (the `DETENT` digi-stage), and the rotation drive (the `ROTAT` stage).

An additional assembly, `M3MAN`, is provided as a helper for implementing the manual controls. Its components are the `DRUM` and `SWINGARM` assemblies. The hand paddle inputs are processed by this assembly, which simply invokes the general-purpose `M3AGENT` sequencer keyword (see below) to execute the appropriate command.

3.4 Sequencers

K1DM3's components must always be operated in specific and strict sequences, lest serious damage occur to the system. This is quite different from typical science instruments, in which components can usually be operated independently of one another without conflicts.

K1DM3 uses *sequencers* to simplify the work for users, along with sequencer keywords to trigger the actions. A sequencer is piece of software that that implements all the necessary keyword commands, in the correct order, to carry out a desired action. Simple assemblies' modify commands are a kind of sequencer, although we don't usually refer to them that way. In *galildisp*, a sequencer is usually an external program and is executed as the sequencer keyword's "modify" action. Unlike ordinary, standalone scripts, however, a sequencer script acts as an integral part of the dispatcher.

Sequencers are particularly important to *k1dm3* because moving between positions involves many high-level decisions and sequencing, all of which has to be presented to DCS as a single simple move. For example, when the `dcs1 TERTMOVEC` keyword demands a move from the `LNas` position to the `Cass` position, it is handled by *k1dm3*'s `TERTMOVEC` sequencer keyword (see figure 4.1 on page 30), which invokes a script that does the following steps:

1. Move the drum: disengage the drum's detent mechanism, rotate to the Cass drum position, engage the detent, and shut off the 48V rotation power.

2. Activate the swingarm subsystem: connect the inner drum air supply, turn on the 24V and 48V power supplies to the swingarm, unsuspend the swingarm and dock dispatchers, initialize the swingarm actuators' motor encoders from the absolute encoders, open the swingarm clamps, and initialize the actuators' brushless motors.

3. Move the swingarm: open the swingarm clamps, ensure the dock pin is retracted, move quickly to nearly the retracted (Cass) position, move slowly into the dock position, and engage the dock pin.

4. Switch to "observing" configuration: Suspend the swingarm and dock dispatchers, shut off the 48V and 24V power supplies, and double-check that the rotation power is off.

The K1DM3 sequencer scripts are implemented as bash scripts, although, since they are external programs, they can be implemented in any language, whether script or compiled program. A sequencer program normally communicates with the dispatcher using simple text commands over its standard input and output channels, instead of using

the standard KTL client keyword library. All K1DM3 sequencer scripts are installed in `$RELDIR/sbin/k1dm3/`.

Sequencer programs have several advantages over standalone external scripts:

- The dispatcher knows that the executing program is part of a particular modify command, and can manage it as an integrated part of the dispatcher.
- If the modify command times out, the dispatcher will signal the sequencer and its child processes, if any. (The dispatcher starts the program as a new Unix process group, so it is trivial to signal the entire collection of processes.)
- If the in-progress modify command needs to be cancelled, the dispatcher will signal the sequencer's process group.
- A sequencer script can invoke modify commands, call dispatcher procedures, set any keyword value, and returns a success or failure code exactly as is done by an internal modify routine.
- When the dispatcher executes a command that is triggered from a manual control configuration, the command executes in a "manual context," meaning that it won't be locked out along with normal keyword modify's from client applications. The manual context is inherited by every subsequent command triggered by that initial manual command, including other sequencer scripts and the modify commands that they in turn execute,* and therefore arbitrarily complex commands can execute on behalf of a manual input.
- Lastly, the dispatcher is usually configured to copy the standard output from a sequencer script into an associated string keyword, which provides a simple yet valuable means of tracking the progress of a large sequencer script.

3.5 The ACTIVATE Sequencer

The K1DM3 module's design imposes constraints so that rotation and swingarm motion cannot occur simultaneously. K1DM3 provides the `ACTIVATE` sequencer keyword to put the module into the correct configuration for operating the swingarm, rotating the drum, observing, or transport the module between tertiary tower and handling cart.

The `ACTIVATE` sequencer takes actions, as needed, to put the module into the requested configuration: suspend or un-suspend dispatchers; adjust control modes; turn on or off power supplies; engage or disengage the detent and air supply mechanisms; open the clamps if needed to initialize the swingarm; initialize the swingarm. However, the `ACTIVATE` sequencer does not move the drum or swingarm, except for the $\sim 250\mu$m required to initialize the swingarm.

The `ACTIVATE` definitions implements a set of best practices: not every operation is *required* for the dispatchers to permit motion, but they are all *desirable* during normal operations — for example, all sources of heat should be turned off during observations. Thus, even if the `ACTIVATE` keyword's value isn't `Swingarm`, it is often possible to operate

*A sequencer application has to use *galildisp*'s special "internal modify" method in order for the manual context to be passed through the sequencer to its own modify commands. If the sequencer uses the ordinary *modify* shell command, it will not run in a manual context.

the Swingarm. (Section 6.2 gives the minimum conditions for operating the drum and swingarm.)

The full set of ACTIVATE values is:

- `ACTIVATE=Swingarm`. In this configuration, *k1dm3* is able to move the swingarm. In normal operations, the basic conditions for swingarm operation are that the drum is initialized, the detent is engaged, and the position is either `MirrorUp` or `Cass/Stow`. If the position is `MirrorUp`, the inner and outer drums must be pinned together; if `Cass/Stow`, the module must be in the tower and the elevation must be at least 45°.

  The `ACTIVATE` keyword will indicate that the module is in the `Swingarm` configuration only when a much more extensive set of conditions is true, which collectively indicate that the rotation controls are powered down (and therefore not generating waste heat), and the `SWINGARM` software assembly is ready to move without further ado:

  - all dispatchers are in the `Ready` state;
  - the inner drum 24V power supply is on;
  - the detent mechanism is engaged;
  - the rotation motor is off;
  - the rotation power supply is off;
  - the inner drum air supply is connected;
  - the docking pin is disengaged;
  - the swingarm control mode is `Pos`;
  - the swingarm actuators are slaved together;
  - the brushless motors' sine amplifiers are initialized;
  - the actuators are calibrated.

  Setting `ACTIVATE=Swingarm` executes all of the commands to do the above, if the drum is already in the `Stow` or `MirrorUp` positions. The sequencer script for this configuration is *activate.swingarm*.

- `ACTIVATE=Swingarm/noinit`. This configuration means that the `SWINGARM` software assembly has taken all steps to prepare for motion, except that the actuators aren't initialized. This is useful from time to time for some engineering purposes. For example, you may need to interact with the swingarm, but the module is in an orientation that doesn't allow the clamps to be opened. The sequencer script for this configuration is *activate.swingarm*.

- `ACTIVATE=Rotate`. This configuration means that the drum is ready for rotation. Power to the inner drum is shut off before motion, because the power goes through contacts that are only live at at the 6- and 12-o'clock drum positions. The `Rotate` mode means:

  - the air supply is disengaged from the inner drum;
  - the inner drum dispatcher (dispatcher 1) is `Suspended`;
  - the other dispatchers are `Ready`;
  - the inner drum 24V and 48V power supplies are off;

  - the swingarm is deployed and the clamps are closed;
  - the outer drum (rotation power) 48V supply is on;
  - the rotation control mode is `Pos`.

  Setting `ACTIVATE=Rotate` executes the commands to do the above, if the swingarm is in already deployed and clamped. The sequencer script for this configuration is *activate.rotate*.

  The sequencer does not deploy the swingarm into the V-blocks if it is not already there, nor will it close the clamps if the swingarm is in the V-blocks but the clamps are open. (The sequencer is allowed to *open* the clamps as a normal part of transitioning to `Swingarm` mode. However, the deployed-but-unclamped position is sufficiently unusual that we do not let the sequencer automatically close the clamps, but instead require the clamps to be closed from outside of the `ACTIVATE` sequencer.)

  If the swingarm dispatcher is not yet suspended when the `ACTIVATE` sequencer executes `Rotate` mode, the sequencer will turn off the swingarm motors and put the swingarm mode to `Halt` before suspending the swingarm dispatcher. If the swingarm is already suspended, the sequencer will disregard the swingarm motor settings.

- `ACTIVATE=Observe`. This configuration implements the recommended settings for science work:
  - the drum's ordinal position must be positive (this implies that
  - the drum is positioned at one of its named ordinal positions);
  - the detent must be engaged;
  - the air supply is disengaged;
  - the inner drum power supplies are off;
  - the rotation motor power is off;
  - the inner drum dispatcher is suspended;
  - the swingarm is either at the Retract position, or it is deployed and clamped.

  The sequencer will command all the above as needed (excluding any drum rotation, of course), including closing the clamps if the swingarm is in the `Deployed,Unclamped` state. (This is at odds with the rule for activating `Rotate` mode, which won't close the clamps. I don't have an explanation for the difference. (Sorry).) The sequencer script for this configuration is *activate.observe*.

- `ACTIVATE=Transport`. This configuration is the setting recommended for moving the K1DM3 module in and out of the tertiary tower. All dispatchers are suspended; all power is off; the air supply mechanism is disengaged; the detent mechanism is engaged; the drum rotation position is MirrorUp; and the drums are pinned together.

  The sequencer will command all the above as needed, excluding any drum rotation and, of course, pinning the drums together is a hand operation. The sequencer script for this configuration is *activate.transport*.

3.6 The M3AGENT Sequencer

The `M3AGENT` keyword (section 4.2) unifies all three methods of controlling K1DM3 — TCS, command line, and hand paddle. Every standard operation needed to support these is implemented as one of the `M3AGENT` modify values. Its sequencer is the script *m3.agent*. This script simply checks its arguments and invokes one of the specific scripts *m3.halt*, *m3.init*, *m3.stby*, *m3.move*, *m3.jog*, *m3.stop* (Abort), *m3.nmir* (no-move-init-rotate), or *m3.nmie* (no-move-init-encoders).

Each manual button is implemented by simply invoking `M3AGENT` with the correct value.

TCS controls are implemented by directly invoking the appropriate *m3.**xxxx*** sub-sequencer, rather than invoking `M3AGENT` itself. By doing so, the sub-sequencer can see that it has been invoked by one of the `TERT`***xxx*** keywords, rather than `M3AGENT`, so it will set the appropriate complementary TCS keyword to `yes` or `true` for the duration of the command, and then to `no` or `false` upon completion of the command.

The `M3AGENT` sequencer primarily controls the module using the ACTIVATE sequencer and the `SWINGARM` and `DRUM` assemblies. It also will directly command the power supplies as needed (`INNER24V`, `INNER48V`, `OUTER48V`), recover from non-normal situations by commanding the `AIRSUPP`, `CLAMPS`, and `DETENT` digi-stages, and do a no-move-initialization by directly commanding the `ROTAT` drum rotation drive.

3.7 Monitoring Sequencer Execution

It is useful to know the general sequence in which these keywords are set and cleared, so that you can find relevant information from the keyword history, or create new GUI's that work well with the sequencer output. The `ACTIVATE` and `M3AGENT` sequencers have similar sets of keywords, and the following description applies to both of them. The `ACTIVATE` sequencer has keywords `ACTIVRUN`, `ACTIVERM`, and `ACTIVMSG` (and more); the `M3AGENT` sequencer has keywords `M3RUN`, `M3ERM`, and `M3MSG` (and more). When these sequencers run, the following keywords are set in the order shown below (here, *xxx* = {`ACTIV`, `M3`}):

Initialization before each execution of the sequencer:

- ***xxx***`RUN` = `true`
- ***xxx***`ERM` = "" (empty string)
- ***xxx***`MSG` = "" (empty string)

Immediately before exec'ing the sequencer:

- ***xxx***`MSG` = *`basename-of-sequencer`*`:  begin`
- ***xxx***`MSG` = *full command line of exec'd sequencer*

When sequencer runs:

- ***xxx***`MSG` = *various messages from sequencer itself*
- If the sequencer finds a problem and emits an error code and error message before exiting: ***xxx***`ERM` = *sequencer-provided error info*

Note that the sequencer is free to generates messages and error codes in any order.

When a sequencer program exits:

- If it exits with a non-zero exit status:
  - *xxx*`MSG` = `Error In program:` *program error info*
  - If the sequencer emitted an error code and error message before exiting, we add a second *xxx*`MSG` entry:
  - *xxx*`MSG` = *sequencer-provided error info*
  - *xxx*`ERM` = *same sequencer-provided error info*
- If it exits with status = 0, but it emitted an error code and error message before exiting:
  - *xxx*`MSG` = *sequencer-provided error info*
  - *xxx*`ERM` = *same sequencer-provided error info*
- If it exited with exit status = 0, and no error code or message (but possibly a success code/message):
  - *xxx*`MSG` = *basename-of-sequencer*`:  done`

  (Note that the `done` line is only emitted on success.)

xxx`RUN` = `false`

A key point is that the sequencer is running while *xxx*`RUN` keyword is true, and all sequencer output and status have been recorded in the *xxx*`MSG` and *xxx*`ERM` keywords before the *xxx*`RUN` keyword becomes false.

When an error occurs deep in a sequencer script, it will be logged in the *xxx*`MSG` keyword, but that will generally followed by a sequence of other messages that overwrite the initial problem message, making it hard for the non-expert user to see the original cause of the problem. However, the *xxx*`ERM` keyword will generally show the problem in its last two lines of output. A recommended way of notifying users of sequencer errors is to monitor the *xxx*`ERM` and *xxx*`RUN` keywords. When *xxx*`RUN` goes true, clear the *xxx*`ERM` display. Then display the most recent two lines of *xxx*`ERM` output. This will remain blank on success, but will show the error if there is a problem.

Chapter 4

Dispatcher Interfaces for Routine Operations

This section introduces the three high-level dispatcher interfaces that are used for all routine operations of K1DM3 (the graphical user interfaces for working with K1DM3 are introduced in chapter 5):

- TCS. Whenever K1DM3 is installed in the tertiary tower, its dispatcher monitors and responds to TCS demands. This is the most common mode of use. As illustrated in figure 4.1, when one of the TCS `TERT`*`xxxx`* keywords is set, the value is immediately mirrored in the *k1dm3* keyword of the same name. In turn, the change of the *k1dm3* keyword triggers the execution of the corresponding sequencer script.

- Hand Paddle. When the hand paddle is connected, most remote control by keyword is locked out.[†] The hand paddle buttons trigger motion by invoking the M3MAN keyword,[‡] which has a set of rules that invoke M3AGENT with an appropriate value for each button combination. In this case, the M3AGENT sequencers will execute in a "manual context" so that the dispatcher recognizes that each lower-level keyword modify is executing on behalf of a manual command.

- `M3AGENT`. The sequencer keyword `M3AGENT` supplies an all-in-one interface for commanding the module (see figure 4.1). Depending on the value that is written to it, it can invoke one of the "M3" sequencers that execute moves to any named position, TCS demands, hand paddle commands, or a special "no-move-init" action that can usually initialize the drum with no motion. This keyword is recommended as the standard interface for nearly all actions carried out by scripts and GUI's.

  It is not much of an exaggeration to say that the `M3AGENT` keyword contains everything one needs to know about the *k1dm3* service. Both TCS and hand paddle operations

[†]The keywords that may be commanded remotely are those that don't cause motion, and any *`xxx`*`STP` (STOP) command.

[‡]An implementation detail of the *galildisp* dispatcher prevents associating the manual control rules directly with the "compound" keyword M3AGENT. The workaround is to create a little assembly, M3MAN, which has the exact same set of valid positions as M3AGENT. The manual control rules are tied to the M3MAN assembly, and whenever a manual button is pressed to direct M3MAN to some action, the M3MAN "modify" rules simply invoke M3AGENT to carry out the action.

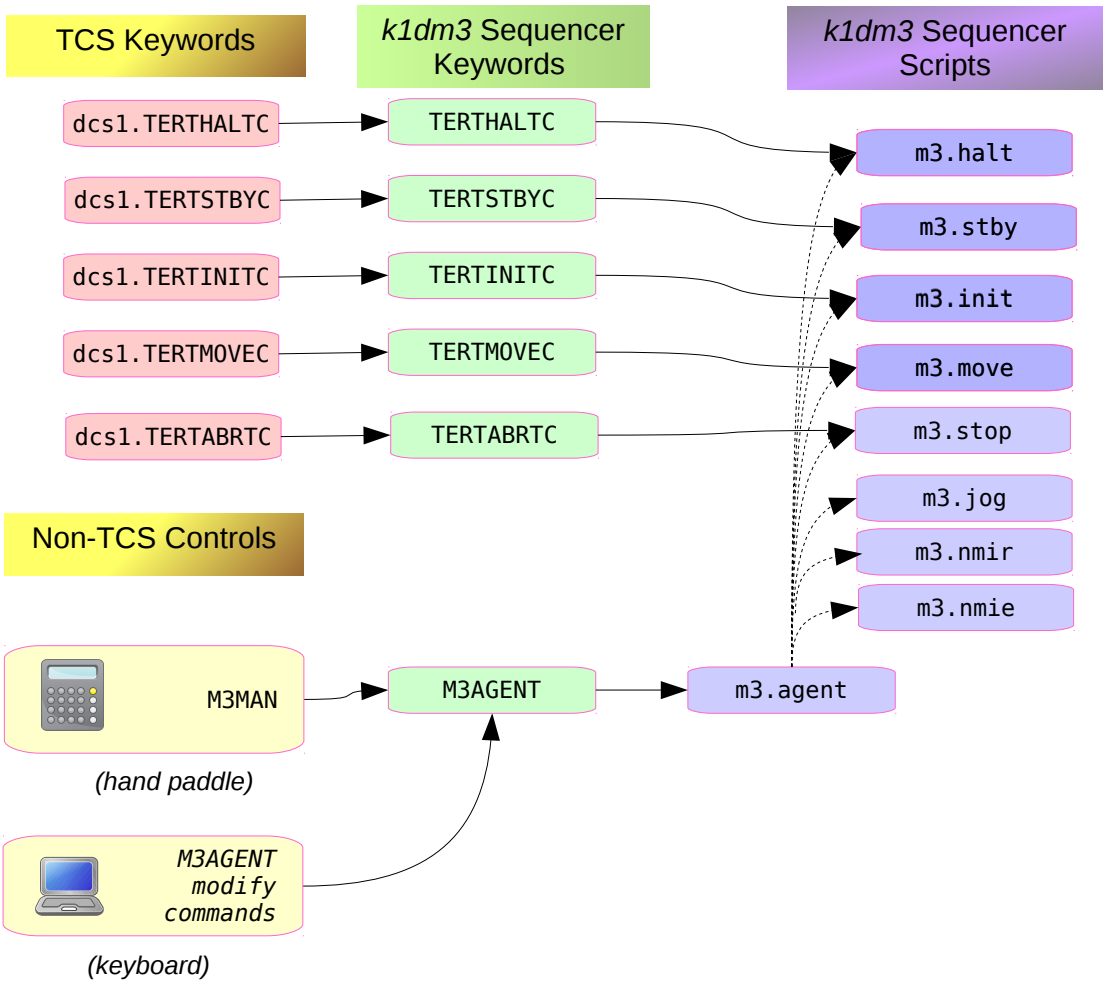


Figure 4.1: High Level Control Flow

require the module to function correctly using just a handful of simple commands. This implies that K1DM3 has to be able to recover from a wide range of foreseeable situations, without any assistance from technical staff. Therefore, the `M3AGENT` sequencer embeds rules to recover from a wide range of problems, especially abnormal start conditions.

4.1 TCS Operations

K1DM3 implements the standard DCS state model, as described in KSD-46a (Lupton, 1997), interacting with TCS through the `dcs1` KTL keyword service. When in the tertiary tower, or when the USEDCS flag is set in the engineering keyword `M3ENG`, K1DM3 will monitor the TCS tertiary keywords and execute the usual Halt/Init/Stby/Move/Abort requests. (The Abort action is not part of KSD-46a, but was retrofitted to support the new TCS execution model. For K1DM3, Abort is the same as a simple Stop action, and is implemented by invoking the Stop sequencer. Abort differs from Halt in that the latter explicitly leaves the module in a ready-to-move state, whereas Abort doesn't worry about ensuring a ready-to-move state.)

For convenience, we reproduce the familiar state diagram, Figure 3 of KSD-46a:

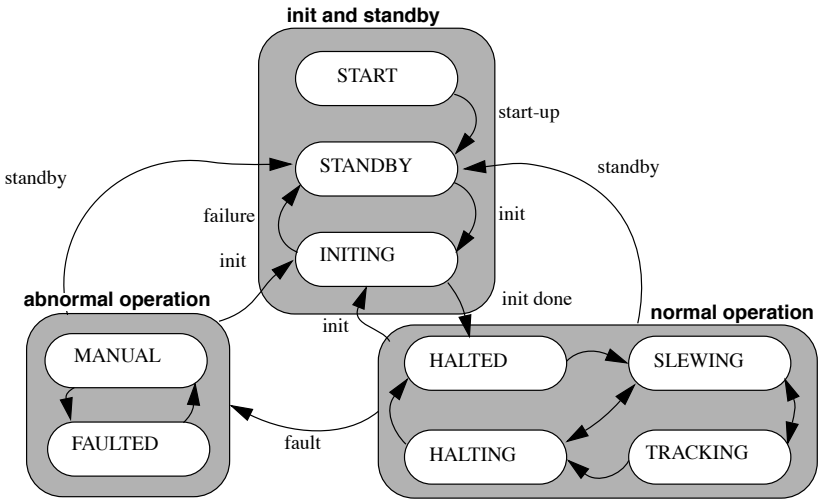


Figure 4.2: DCS State Diagram (from KSD-46a, fig 3)

K1DM3 provides sequencer keywords `TERTHALTC`, `TERTINITC`, `TERTSTBYC`, `TERTABRTC`, and `TERTMOVEC`, which are used to implement the corresponding TCS actions. The K1DM3 keywords mirror the same-name TCS tertiary keywords — that is, whenever the TCS keyword changes, its same-name K1DM3 keyword is changed to the same value. In turn, the "modify" procedure for the K1DM3 keyword carries out the actual action. The motivation for this approach is to provide a clean layer of separation between TCS and K1DM3 — one can test most of the command chain by directly modifying the K1DM3 keyword, without needing TCS to be available.

K1DM3 invokes the corresponding sequencer keyword's modify procedure, which command K1DM3 to carry out the desired sequence of actions, and sets the appropriate TCS result values. These modify procedures invoke sequencer scripts that are named, respectively, `m3.halt`, `m3.init`, `m3.stby`, `m3.stop`, and `m3.move`, and are installed in `$RELDIR/sbin/k1dm3/`.

4.2 M3AGENT

`M3AGENT` is an enum keyword with a set of *modify* values that allows it to be used as a master do-everything sequencer. Actions are started by invoking one of its modify choices:

```
modify -s k1dm3 M3AGENT=value
```

where the valid values are:

- TCS-like actions: `Halt`, `Init`, `Stby`, `Stop`
- Instrument positions: `LNas`, `RNas`, `LBC1`, `RBC1`, `LBC2`, `RBC2`, `Cass`
- Supplementary positions: `Stowed`, `Mirror90`, `MirrorUp`
- Other motions: `StepPos`, `StepNeg`, `JogPos`, `JogNeg`
- Special actions: `NoMoveInitRotate`, `NoMoveInitEncoders`, `EngrTest`

The `M3AGENT` values `Halt`, `Init`, `Stby`, and `Stop` trigger the same sequencer scripts as used to implement TCS commands.* The only difference between a TCS request and using the keyword `M3AGENT` is that TCS keywords `TERT`*xxx* are not set when using `M3AGENT`.

The position values (`LNas`, `RNas`, etc) likewise trigger the same `m3.move` sequencer script that is used for the TCS `TERTMOVE` command, with the only difference being that TCS keywords are not set when using `M3AGENT`.

The values `StepPos` and `StepNeg` implement the hand-paddle actions for commanding the drum to step to the next instrument position, either increasing or decreasing angle.

The values `JogPos` and `JogNeg` are mainly provided for engineering purposes. They cause the drum to rotate constantly in the direction of increasing or decreasing angle. The rotation speed is set by the `ROTATMSP` (MSP = Maximum SPeed) keyword, which defaults to 7818 Renishaw encoder counts/s (40,000 motor encoder counts/s).
Note: the `ROTATMSP` keyword limits the speed of any rotation, so if you set it to a low value for some engineering task, it's important to reset it to its maximum value for normal operations.† The value `Stop` stops any motion of both drum and swingarm. It differs from `Halt` because it only stops motion, but doesn't change any modes, turn off power supplies, *etc.*

The supplementary positions provide the daytime park position, and two engineering positions: `Stowed` means that the swingarm is deployed and clamped in the V-blocks, facing down (drum at the Cass/Stow orientation). `MirrorUp` also means that the swingarm is deployed and clamped in the V-blocks, but it is facing up, i.e. 180 degrees from the Stowed position. `Mirror90` is the position for removing the mirror for recoating. The drum is at the same orientation as MirrorUp, but the mirror is raised to be parallel to the ground.

*For *k1dm3*, the TCS `Abort` command is no different from `Stop`, and thus TCS `Abort` is implemented by simply invoking the `M3AGENT Stop` command.

†As a convenience, the k1dm3 KTL library allows you to use the literal string `maxv` for the maximum allowed value of any `int`, `int64`, `float`, or `double` keyword, e.g. `ROTATMSP=maxv`.

The special value `NoMoveInitRotate` is provided to initialize the rotator without any motion. This is an essential part of bootstrapping the system whenever power has been removed and the swingarm is not clamped in the V-blocks, such as after recoating the mirror, or after a power outage when the module is in the tertiary tower in the Cass position. If either of the limit switches at the Cass/Stow and MirrorUp (6- and 12-o'clock) positions are active, `M3AGENT` will set the current encoder value to the corresponding position, and mark the subsystem as initialized.* Without these switches, initialization can deadlock because the dispatcher will refuse to move the swingarm unless the drum is in the Cass/Stow or MirrorUp position, and the dispatcher will refuse to rotate the drum for initialization unless the swingarm is clamped in the V-blocks.

The special value `NoMoveInitEncoders` extends `NoMoveInitRotate` to also set the swingarm actuators by reading the Renishaw absolute encoders and setting the actuator motor encoders to the corresponding values. It is not essential for bootstrapping the system, but is a useful shortcut to a setting that is needed for any engineering work with the clamps.

Finally, the engineering value `EngrTest` can trigger the invocation of one of three "engineering" sequencer scripts. If the engineering keyword `M3ENG`'s value contains one of the flags `Engr1`, `Engr2`, or `Engr3`, then the matching script *m3.engrTest1*, *m3.engrTest2*, *m3.engrTest3*, is executed. Anything whatsoever can be placed in these test scripts, which is helpful for development of additional or alternative `M3AGENT` sequences.

4.3 Hand Paddle

The K1DM3 hand paddle is a simple off-the-shelf component with 4 LED's and 8 buttons. Its controls operate at a high level: pressing a button generates a command to the `M3AGENT` keyword. This is distinctly unlike many systems in which manual controls directly control low level components. The exacting sequencing requirements for K1DM3 make it impractical to expect a hand paddle user to know and follow an extensive series of low-level commands, say to rotate the drum from the `Stowed` position to the `MirrorUp` position, and then raise the swingarm to the `Mirror90` position.

When the hand paddle is connected, external keyword commands are rejected if they might cause any motion or turn on power, except that keyword commands to stop axes are allowed. When a hand paddle button triggers any `M3AGENT` action, the executing code runs in a "manual context," which the dispatcher will allow to execute even though the same `M3AGENT` request would be rejected from a modify command. The lockout of external keywords can be overridden for up to 20 minutes by setting the `MAN_OVERRIDE` timer keyword to a number between 0 and 1200 seconds. (The override can be extended at any time by re-entering a new value for `MAN_OVERRIDE`.) This can be useful or even necessary for engineering purposes, but is not normally needed for normal work when the module is on its handling cart. All of the normal restrictions on module motion apply when the hand paddle is connected.

*The drum position may be inaccurate if the detent mechanism is not engaged during a `NoMoveInitRotate`, as the drum can be up to $\sim$ 6 mm from the nominal position while the switch remains active. However, the detent mechanism can only be engaged if the drum is initialized. In that case, best practice is to execute one `NoMoveInitRotate`, then engage the detent (which will drag the drum into accurate position), then repeat the `NoMoveInitRotate`. The sequencer for the TCS Init command does this check-engage-and-retry.

4.3.1 LED's

The four hand paddle LED's are, left-to-right:

- LED 4 (green). Solid: Paddle is connected, module is stopped.
  Blink: Paddle is connected, and an action is executing. Note that a request can be active and in progress for 5-10 seconds without any user-visible activity other than the blinking LED, because of the time required to power up a Galil and connect to it.
- LED 3 (green). Off: no error condition occurred (as recorded by `M3ERM` keyword).
  Blinking, 5 seconds: a command is rejected, or a runtime error occurred. All normal restrictions on module motion apply when the hand paddle is connected, so there can be many reasons for a rejected request; you should use the K1DM3 engineering GUI (section 5.2) to see the issue preventing motion.
- LED 2 (yellow). Off: Drums are not pinned together.
  On: Drums are pinned together, which is required for raising the swingarm to the `Mirror90` position. (This condition gets its own indicator light because it appears to be the most common cause of the blinking request-rejected LED.)
- LED 1 (red). Off: E-Stop not active.
  On: E-Stop is active.

4.3.2 Buttons

The buttons trigger actions when they are depressed, and do nothing when the button is released — motion does *not* auto-stop upon releasing the button. To stop any on-going motion, press the Stop or E-Stop buttons; also, if doing a Jog, you can press Jog again to stop it.

A peculiarity of the dispatcher is that you must hold the button for up to 50 ms to trigger an action, whereupon the leftmost LED will begin blinking. That is because the hand paddle inputs are processed entirely by the dispatcher, which receives Galil updates at 20 Hz, and are not "stretched" by the K1DM3 electronics, nor latched in the Galil.

Button-release is a no-op because each hand paddle action can take up to a minute to execute, which is a very long time to hold down a button that takes a fair amount of force to depress.

The eight buttons are shown in Table 4.1.

Table 4.1: Hand Paddle Functions

Row	Left Button	Right Button	Functions
1	**Shift**	**E-Stop**	• The **Shift** (or "Function") button does nothing on its own. However, holding it down while pressing another button selects a second function. • The **E-Stop** activates an E-Stop condition (at a hardware level, not merely a software E-stop).
2	**Out**	**In**	• Move the mirror out of or into the light path. ⋆ On a handling cart, **Out** means the `Mirror90` (horizontal) position. The drums must be pinned together for swingarm motion on the handling cart. ⋆ In the tertiary tower, **Out** means the full-retract position.
3	**MirrorUp/ Stow**	⇑**Step−** **Step+**	• Left button: rotates the drum between `MirrorUp` and `Stow` orientations, the two most common positions. • Unshifted right button: steps the drum to the next increasing-angle detent position. • Shifted right button: steps the drum to the next decreasing-angle detent position.
4	⇑**Jog−** **Jog+**	**Stop**	• Unshifted left button: invokes the `JogPos` action. • Shifted left button: invokes the `JogNeg` action. See section 4.2, above, for the description of these actions. • Right button: stops motion.

Chapter 5

GUI's

This section introduces the three graphical user interfaces for working with K1DM3 (the dispatcher's interfaces were introduced in chapter 4): the standard TCS interface, *Tcsgui*, an engineering utility, *k1dm3_gui*, and *k1dm3mon_gui*, a utility for showing the status from *k1dm3mon*.

5.1 *Tcsgui*

The reader is assumed to be familiar basic *Tcsgui* use (see figure 5.1); this section focuses on the *k1dm3* implementation of each of the *Tcsgui* functions.

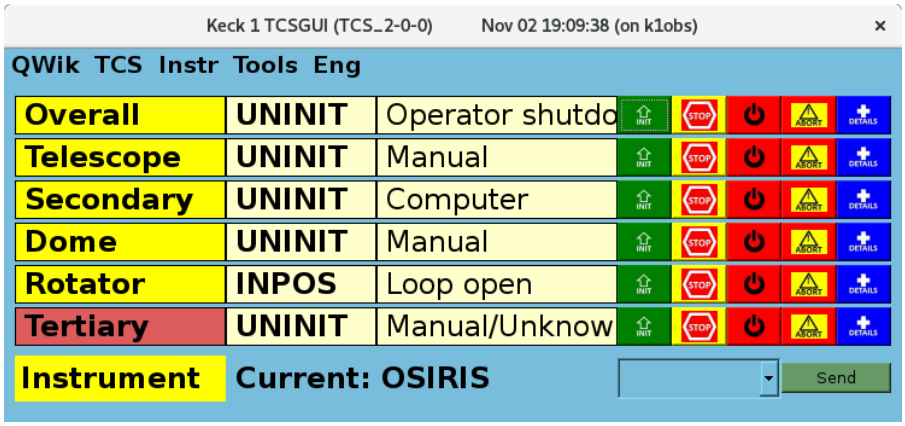


Figure 5.1: Tcsgui

Engineering GUI. The `Eng` drop-down menu has a `ROT DM3` entry (not shown here) that will ssh to k1dm3server and bring up the K1DM3 engineering GUI (section 5.2).

***Instrument*/Send**. Selecting an instrument from the drop-down menu and pressing `Send` causes *k1dm3* to move to that instrument position. It will auto-Init if necessary to fulfill the position demand. Exception: this will be rejected, with message "Init required," when K1DM3 is in standby (i.e., dispatchers are suspended).

- On completion of the move, all power to the inner drum is shut off, and 48V power for rotation is turned off.

INIT. The `INIT` button causes the *dcs1* keyword `TERTINITC` to be set to true, which triggers *k1dm3* to invoke the script to execute the script `$RELDIR/sbin/k1dm3/m3.init`. This is the only TCS command that can execute when the drum dispatcher is suspended. (The drum dispatcher acts as the "lead" dispatcher and handles all TCS interactions.)

- It un-suspends the dispatcher,
- initializes the drum if needed, and
- finishes with a move to `TERTDEST`.
- On completion of the move, all power to the inner drum is shut off, and 48V power for rotation is turned off.

The `INIT` will fail if the E-box is unpowered; otherwise, it will generally succeed.

STOP. The `STOP` button causes the *dcs1* keyword `TERTHALTC` to be set to true, which triggers *k1dm3* to invoke the script to execute the script `$RELDIR/sbin/k1dm3/m3.halt`.

- If either drum or swingarm is in motion, it is stopped.
- The control mode is set to `Pos`, leaving the module ready to receive normal motion commands.
- The `STOP` button does not turn off any power supplies.

Exception: the `STOP` button is ignored when K1DM3 is in standby (i.e., dispatchers are suspended).

0/1. The "on/off" button causes the *dcs1* keyword `TERTSTBYC` to be set to true, which triggers *k1dm3* to invoke the script to execute the script `$RELDIR/sbin/k1dm3/m3.stby`.

- If either drum or swingarm is in motion, it is stopped.
- All power to the inner drum is shut off, and 48V power for rotation is turned off.
- Lastly, it suspends all dispatchers, putting the system into "standby" mode.

ABORT. The `ABORT` button causes the *dcs1* keyword `TERTABRTC` to be set to true, which triggers *k1dm3* to invoke the script to execute the script `$RELDIR/sbin/k1dm3/m3.stop`.

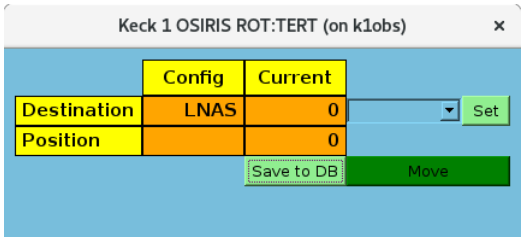
- If either drum or swingarm is in motion, it is stopped.
- The `ABORT` button does not turn off any power supplies. The `STOP` and `ABORT` are very similar, except that the `ABORT` command doesn't set any particular control mode after stopping ongoing motion.

Exception: the `ABORT` button is ignored when K1DM3 is in standby (i.e., dispatchers are suspended).

DETAILS.

The detail panel (see figure 5.1) can be used to send K1DM3 to a new position, without affecting the currently-selected instrument.

To do this, select a new instrument from the details panel drop-down menu, and *Tcsgui* will set the new tertiary position in the `rot:tert:userDp` EPICS channel (K1DM3 will receive this as `dcs1` keyword `TERTDEST`). Then press the 'Set' button, and *Tcsgui* will set the `rot:tert:slew` EPICS channel to True; when K1DM3 sees this transition (as `dcs1` keyword `TERTMOVEC`) it initiates a move command to the tertiary position.



In general, all unnecessary power is turned off after a successful move or after the system is put into the Standby state. After a Fault, `STOP`, or `ABORT`, however, the power supplies are not in any particular state.

5.2 The Engineering GUI

The engineering gui, *k1dm3_gui*, can be started from any host on which the *k1dm3* KTL library is installed. *Tcsgui*'s drop-down entry, "`Eng→ROT DM3`," will ssh to `k1dm3server` and bring up *k1dm3_gui*.

The interface is shown in figure 5.2. The drum status is shown on the left side, and the swing-arm state is shown on the right.

In the middle boxes, the writable keywords have their values shown in grey boxes; these are buttons that allow you to select a new value and command a motion. The value is greyed-out if the corresponding `XMV` keyword (see sections 7.1, 12.4, and 12.5.1) is non-blank, indicating that this element is currently forbidden from moving. The non-writable keywords (e.g. `Drum pin`) have their values shown on a black background.

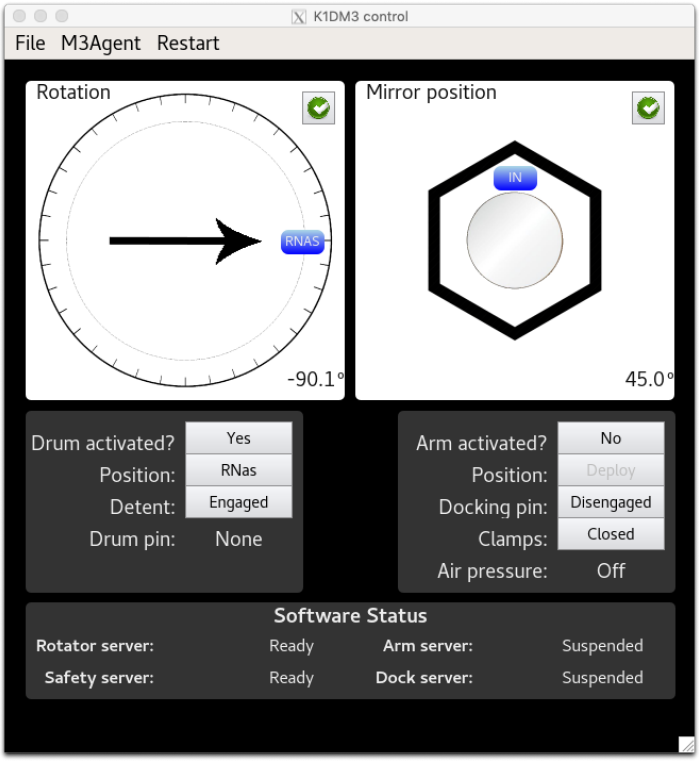


Figure 5.2: K1DM3 Engineering GUI

Near the the bottom, the status of the dispatchers is shown, and at the bottom, any critical dispatcher alerts are shown in yellow.

As an engineering gui, the "Rotation" and "Mirror position" boxes do not try to hide the various state changes that are occurring under the hood, so to speak, and you will sometimes see the mirror position jump around in a non-physical manner. This happens as the swingarm control bootstraps each time the swingarm is activated and the inner drum Galil is powered up. Another minor glitch is that the fraction-complete motion indicators may go to 100%, then back off and show more motion, as the sequencer commands different components through their steps. (The dispatcher doesn't have a graceful way of combining the various motions into a single motion-complete indicator keyword, so the gui is left showing what is available.)

The top-right corner of the "Rotation" and "Mirror position" boxes contain status icons. The icons are red if there is a motion-preventing problem (An X for a serious error; a dash for a less-serious condition); yellow if there is an alert condition; green if the assembly is "Ready", or grey gears if the assembly is moving.

The `Restart` pull-down menu at the top of the GUI allows the user to restart individual dispatchers, or all of the motion-control dispatchers, if and only if the user interface is running on the same computer, *k1dm3server*, as the dispatchers themselves. This may be amended in the future to work from any computer that has access to the `k1dm3` KTL service.

The `M3Agent` pull-down menu at the top of the GUI gives direct access to the `M3AGENT` master do-everything keyword (see section 4.2), allowing you to command motion to any position. It also lets you bring up a log window that shows the combined output of the `M3MSG` and `ACTIVMSG` keyword log streams.

Detail panels for the drum and swingarm can be brought up by clicking the status icons. The detail panel shows a variety of keywords; as with the main panel, any grey boxes are clickable to allow you to command the corresponding writable keyword.

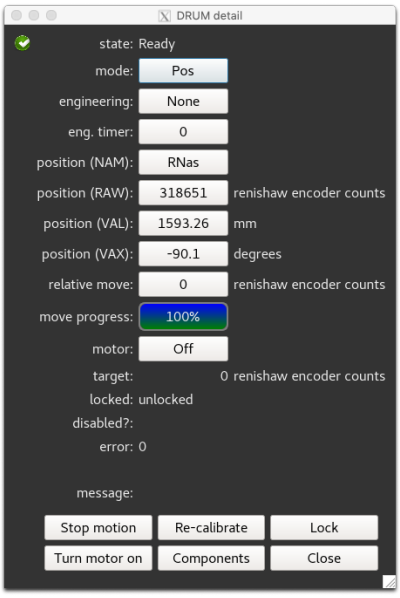


Figure 5.3: Drum Detail Popup

5.3 The *K1dm3mon* GUI

The *K1dm3mon* monitor daemon (section 9) raises an alert when any of around 20 conditions occurs. The conditions are grouped hierarchically, and divided into topics of "Drum," "Swingarm," and so on. *K1dm3mon_gui* provides a quick at-a-glance view of the conditions, by presenting them in a tree view, grouped into separate tabs.

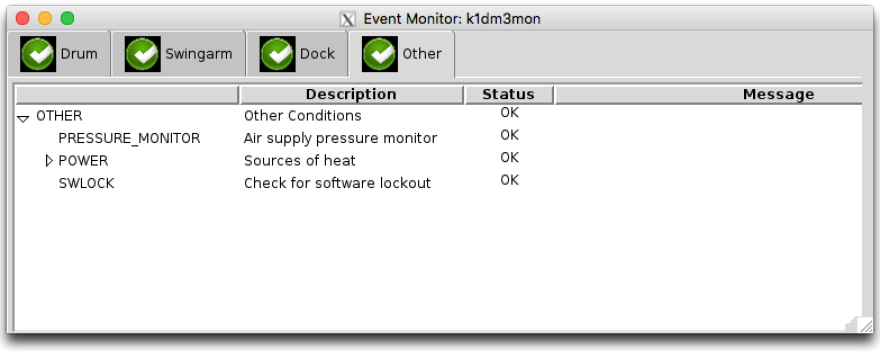


Figure 5.4: k1dm3mon_gui

Chapter 6

Restrictions & Interlocks

6.1 Introduction

The K1DM3 dispatchers implement many software interlocks. This section lists the interlocks, including the rationale behind any non-obvious constraints, and discusses when and how to override them.

Any motor stage or digi-stage — in *galildisp*, a "digi-stage" represents a mechanism that is controlled and monitored by a collection of I/O bits — can have zero or more constraints (interlocks) defined in its configuration. Each stage also has a *stage*`XMV` keyword (mnemonic: `XMV` = "can't move") that lists the active interlocks that prevent motion.

Generally, if a stage has rejected a move request, the reason will be quickly understood by inspecting the `XMV` keyword. If a constraint becomes true while the stage is in motion, the dispatcher will stop the stage. In some cases, the constraint will only be true for a brief moment. For example, if a stage triggers an overspeed condition, and the dispatcher promptly stops the stage, then the condition will only appear in the `XMV` keyword during the brief overspeed period. In that case, the stage's *stage*`ERR` and *stage*`ERM` keywords (error number and error message, respectively) will show the reason for stopping. Sometimes, you may need to inspect the logfile for additional information.

Assemblies (combo-stages) also have `XMV` keywords, but they do not have their own explicitly-defined constraints. Instead, an assembly inherits the constraints of its components, except for those that are expected to be handled by normal sequencing. For example, if K1DM3's swingarm clamps are closed, the two actuators `ARM_A_` and `ARM_B_` are forbidden from moving, and the keywords `ARM_A_XMV` and `ARM_B_XMV` will include the notice "`Clamp state is 'Closed'`." However, this is a normal operating condition — the `SWINGARM` assembly opens the clamps before commanding the actuators to move — so we do not want the `SWINGARMXMV` keyword to indicate that the clamp status is a problem. We handle this in the dispatcher configuration by flagging the clamps-closed constraint as "part of normal sequencing," which tells the dispatcher not to add the component-level `XMV` value to the parent assembly's `XMV` keyword.

Some constraints are for reporting only. For example, no motion is possible when an E-stop is active, because the E-stop condition shuts off power to all moving components. This is made visible to the end user by adding a software constraint to forbid motion when the E-stop is active, so that the condition shows up in the stage's can't-move reason.

Certain constraints can be bypassed by setting an "engineering" flag that turns off the

safety interlock. Each assembly, motor stage, and digi-stage has an engineering-flags keyword, *stage*`ENG`, and setting *stage*`ENG=XSAFETY` will bypass all the "bypassable" constraints for that stage. Each stage also has an engineering-timer keyword, *stage*`ENT`, which can be set between 0 and 1200 seconds. The timer is automatically set to 1200 seconds whenever the `ENG` keyword is set; when it expires, the engineering flags are inoperative. The timer keyword can be modified at any time to extend or disable the engineering flags.

When an assembly's engineering keyword is set, it recursively sets the engineering flags of its components to the same value. Thus, if you want to enable the `XSAFETY` flag for all the components of the swingarm, you can simply set `SWINGARMENG=XSAFETY`, or clear the engineering flags with `SWINGARMENG=none`. As a general rule, it is safer to set a stage-specific flag, rather than set an assembly-level flag which may unintentionally bypass additional restrictions.

6.2 Bypassing the M3AGENT Sequencer

In normal use, one should use the `M3AGENT` sequencer for all motions, and the `ACTIVATE` sequencer for some high-level engineering tasks. This will follow best practices for leaving the system in a ready-to-observe state (inner drum air disconnected, power off, *etc.*). Sometimes, for engineering tasks, one wants to issue lower-level commands to put the system into a movable state, but not put the system into the ready-to-observe state. We find that it's all too easy to forget some essential command and be tripped up repeatedly by interlocks, so this section provides a bare-bones outline of what's required for motion.

Drum

The minimum conditions for operating the drum are:

- Swingarm in V-blocks and clamps are closed. If not, and drum is at 6- or 12-o'clock, start with:

  ```
  modify -s k1dm3 M3AGENT=NoMoveInitEncoders
  ```

- Then, if swingarm isn't in the V-blocks:

  ```
  modify -s k1dm3 SWINGARMNAM=Deployed,unclamped
  ```

- If clamps aren't closed:

  ```
  modify -s k1dm3 CLAMPSPOS=Closed
  ```

- Air supply disengaged from inner drum:

  ```
  modify -s k1dm3 AIRSUPPPOS=Disengage
  ```

- Inner drum 24V and 48V supplies are off:

  ```
  modify -s k1dm3 INNER48V=Off
  modify -s k1dm3 INNER24V=Off
  ```

Swingarm

The minimum conditions for operating the swingarm are:

- If in the tower, and the drum is at the Cass/Stow position, then the telescope elevation must be $\geqslant 45$ degrees.
- If on the handling cart, the drum must be at the MirrorUp orientation.
- Whether in the tower or on the handling cart, if the drum is in the MirrorUp orientation, the inner and outer drums must be pinned together.
- The drum detent mechanism and air supply must be engaged:

```
modify -s k1dm3 DETENTPOS=Engage AIRSUPPPOS=Engage
```

6.3 Conditions Without Override

6.3.1 Swingarm

The following conditions forbid swingarm movement, and do not have an `XSAFETY` override:

- an E-stop is active;
- any cables except the hand paddle are disconnected;
- an actuator motor has overheated;
- there is no signal to indicate if the module is in the telescope tower or on the handling cart;
- the module is in the mirror-facing-up orientation but the inner and outer drums are not pinned together;
- the drum is not homed.

The distinction between in-tower and on handling-cart is important because the swingarm can't be safely unclamped when the module's orientation is near horizontal, i.e. at low elevations or on the handling cart. When operating the swingarm with the mirror facing up — normally only needed for mirror removal — the inner drum becomes grossly unbalanced when the swingarm retracts out of the V-blocks. In that case, gravity can cause the inner drum to whip around at startling and dangerous speeds, and therefore this motion is always disallowed unless the drums are pinned together. (When retracting the mirror in the normal operational mirror-down "Cass/Stow" orientation, the module is stable against gravity.) The drum must be homed for swingarm motion so that we can trust the reported swingarm orientation (mirror up vs down). (Note that in the positions with swingarm power, the drum can be homed without motion, so there is no reason to supply a safety-override, and good reasons not to do so.)

6.3.2 Drum

The following conditions forbid drum movement, and do not have an `XSAFETY` override:

- an E-stop is active;
- any cable is disconnected (except the hand paddle);
- no signal is active indicating the module is either in the telescope tower or on the handling cart;
- the inner and outer drums are pinned together;
- the air supply is not disconnected from the inner drum;
- the detent mechanism is not disengaged;
- power to the inner drum is on;
- one of the two position switches that trigger at the MirrorUp and Cass/Stow positions are active, but the drum encoder is not in agreement.

6.4 Overridable Conditions

The following five conditions can be overridden by setting the engineering `XSAFETY` flag for the appropriate stage or assembly. Using the override must be done with careful attention both to the specific condition you are intending to override, and also to the other conditions that will be overridden by setting that stage or assembly's `XSAFETY` flag. Reminder: If you set or clear `SWINGARM` engineering flags, the changes will apply to all of the `CLAMPS`, `ARM_A_`, and `ARM_B_` components.

1. The swingarm clamps and actuators will normally refuse to operate when the drum is at the mirror-down (`Cass/Stow`) orientation, and either (a) the module is in the tertiary tower and the elevation is below 45°, or (b) the module is on the handling cart.

   Consequences of misused `XSAFETY` override:

   - If the swingarm is clamped in the V-blocks, opening the clamps allows the mirror to fall $\sim$ 5cm before the compliance in the swingarm hinge is used up and the mirror is abruptly halted.
   - If the swingarm is retracted in the mirror-facing-up orientation the drum will become unbalanced against rotation; if the drum detent is not engaged (or becomes disengaged due to loss of air), an uncontrolled rotation can occur.

   Therefore, only override this condition if you've mechanically prevented the drum from rotating — such as by pinning the drums together — and ensured the mirror won't drop abruptly from the V-blocks.

   To override the restriction for all clamps and the swingarm actuators, use the assembly keyword:

   ```
   SWINGARMENG=XSAFETY
   ```

   Also affects: items *2, 3, 4*

2. The swingarm actuators will not operate if the drum detent is disengaged.

   Consequences of misused `XSAFETY` override: If you override this to retract the swingarm in the mirror-down orientation, the drum will become unbalanced, and can trigger uncontrolled rotation.

   To override the restriction for the swingarm actuators, use the actuator keywords:
   `ARM_A_ENG=XSAFETY ARM_B_ENG=XSAFETY`

   Also affects: items *1, 3, 4*

3. The swingarm clamps will not operate if the swingarm is not within 0.5 mm of its zero-point in the V-blocks. If you are sure that it is safe to operate the clamps — say, a technician is manually assisting the swingarm to be very close to the proper point, but it's still 1.0 mm away — you may use the `XSAFETY` override and command the clamps to close.

   Consequences of misused `XSAFETY` override: If the swingarm is not close to the correct in-V-block position, The hardened surfaces of the V-blocks could be damaged by being struck by the clamps, or the clamp faces could be damaged by striking part of the swingarm.

   To override the restriction, use the clamps assembly:
   `CLAMPSENG=XSAFETY`

   Also affects: items *1, 2, 4*

4. During K1DM3 installation, and perhaps during some servicing, it's necessary to move the swingarm in the tower without having the docking mechanism connected. At other times, engineering work may also make it necessary to move the swingarm with the dock disconnected. To do this kind of work, the swingarm dispatcher must be reconfigured and restarted, following the instructions in section 2.3, so that it knows it is operating without a connected dock.

   When configured to operate this way — indicated by keyword *`k1dm3`*`.WITHDOCK = false` — the dispatcher expects that the dock's wiring cables will be disconnected. The two swingarm actuators will refuse to move if the docking cables loopback signal is active, since that is contrary to the assertion that the system is operating without the dock. If the dock is in fact connected, but it is necessary to have the swingarm ignore the dock's presence for engineering purposes, then you must override this with an `XSAFETY` override.

   Consequences of misused `XSAFETY` override: If you override this but the docking mechanism is installed and its pin engaged, the module can be seriously damaged by running the swingarm's docking tang into the docking pin.

   To override the restriction, use the assembly keyword:
   `SWINGARMENG=XSAFETY`

   Also affects: items *1, 2, 3*

5. The `CLAMPSCLOSED` signal, which is read by the rotation control Galil, must show that all four clamps are closed. This is safe to override if you know the clamps are firmly closed, or if the module is pointed at zenith so that the swingarm cannot flop around during rotation.

   Consequences of misused `XSAFETY` override:

- If the module is horizontal (or at low elevation), and the swingarm is significantly retracted from the V-blocks, then the drum will be badly out of balance during rotation. If the rotation drive reaches an overtorque condition while pushing the unbalanced drum against gravity, the motor will shut off and the drum will rotate back under gravity. The uncontrolled drum motion can be dangerous to any nearby personnel.
- If the swingarm is close to or in the V-blocks, then the drum will be reasonably well balanced and can still rotate without trouble at any elevation. However, if the swingarm is not restrained from motion, then at low elevations or in the handling cart, its compliance will allow it to sag or fall to the side — possibly dragging against the V-block surfaces — as the module rotates.

To override the restriction, use the rotation drive keyword:

```
ROTATENG=XSAFETY
```

Chapter 7

Recovery From Failures

The great majority of move restrictions are handled by correct sequencing of actions, and the most effective way to execute them is to use `M3AGENT`. This section addresses recovery from more serious problems that requires human intervention. Many of the recovery steps require one to set an engineering safety-override flag.

7.1 General Troubleshooting Tips

- The *k1dm3* service writes voluminous logs, using using*syslog(3)*. The configuration file `/etc/rsyslog.conf` directs the messages from *k1dm3* to files in `/kroot/var/log/`. All messages from dispatcher ***n*** go to file `k1dm3-`***n***`debuglog`; additionally, a copy of all messages with severity higher than `DEBUG` are put into `k1dm3-`***n***`log`.

  Additionally, the *k1dm3mon* service logs to `k1dm3mondebuglog` and `k1dm3monlog`. (The *k1dm3saf* service has similar logfiles, but its entries are generally barren, because pretty much all information of interest is put into its keywords, and there is nothing else to log.)

- The `XMV` keywords (see 12.4 and 12.5.1) are often useful for identifying problems that prevent motion. Don't just look at the `XMV` keyword at an assembly level, such as `SWINGARMXMV`, because sometimes important trouble-shooting values are in its component `XMV` keywords and not propagated up to the assembly level. The following command will show all the `XMV` keywords that are related to the swingarm and drum, respectively:

  `gshow -s k1dm3 %XMV,d:%1` (Swingarm)

  `gshow -s k1dm3 %XMV,d:%2` (Drum)

  These commands select all keywords ending in `XMV` that also belong to dispatcher #1 (`d:%1`) or #2 (`d:%2`).

  Example: With the swingarm retracted, we see:

  ```
  ARM_A_XMV = 48v power supply is 'Off'; Amplifier state;
     Swingarm docking pin not disengaged
  ARM_B_XMV =  (same as for A)
  ```

```
SWINGARMXMV =
```

In this case, the actuators can't move for perfectly normal reasons: the motor power is off; the Galil's motor power amplifier has a state problem (looking at `ARM_A_AST`, the Amplifier STate keyword, we see the problem is an undervoltage condition, because the power is off); and the swingarm dock is engaged. However, the the swingarm assembly level isn't showing any problem at all, because turning on motor power and disengaging the dock are all part of normal swingarm sequencing.

- The `MSG` keywords for each stage or assembly sometimes contain useful messages that explain how the system got to its present state, plus they always contain the text message that accompanies the error code that is put into the `ERR` keyword. The `ERM` keyword contains just the error messages, and leaves out the additional text commentary.

- The keyword history database (Lanclos and Deich, 2012) is an invaluable source of information. It's a nearly complete history of all keyword broadcasts from the *k1dm3*, *k1dm3saf*, and *k1dm3mon* KTL services. The simplest way to retrieve data from it is to use the *gshow* applications, which will query the history database instead of the live service whenever invoked with a `-date` option. Alternatively, you can make your own queries to the postgresql database containing the history data, but it is beyond the scope of this document to go into the details.

- The *k1dm3mon* monitor service (see chapter 9) has been configured to monitor roughly 20 conditions. If the cause of a problem isn't obvious from the *k1dm3* messages, it may be useful to check *k1dm3mon* to see if it has identified the cause.

  It is pretty easy to add new rules and conditions to *k1dm3mon*'s configuration, so if we have left out something that should be monitored, this can be readily addressed.

7.2 Low-level Actuator Operations

If for some reason you want to control the actuators with low-level Galil commands, here is a useful way to do it while keeping the swingarm dispatcher operational, so that you can continue to use the dispatcher keywords for monitoring and other controls. The basic trick is to use the `CTRL1AUX` keyword, which uses the dispatcher to pass raw Galil commands to the swingarm Galil; the response from the Galil is put into the same keyword, and can be read back using a *show* command. Many of the following examples use the *k1dm3.io* command, which simply combines a *modify*, waits 100 ms, and a *show* of the same keyword.

1. **Preliminaries.**

   You'll always need the swingarm Galil on, and you'll also want the 48V supply turned on:

   ```
   modify -s k1dm3 INNER24V=On
   modify -s k1dm3 INNER48V=On
   ```

   If the swingarm Galil was self-suspended (auto-suspended), it will auto-resume when `INNER24V` is turned on; but it will not auto-resume if it was not auto-suspended. In that case, unsuspend it by setting `DISP1SUSP` to the empty string:

```
modify -s k1dm3 DISP1SUSP=
```

After unsuspending, wait until `DISP1STA` is `Ready`. Then do:

```
modify -s k1dm3 CLAMPSPOS=Open DOCKPOS=Disengaged
```

2. **Gantry Mode.**

   The actuators are slaved using the Galil's "gantry" mode. To turn *on* gantry mode, slaving axis B to axis A, do:

   ```
   k1dm3.io CTRL1AUX="STAB;GA,CA;GR,1.0;GM,1"
   ```

   In gantry mode, commanding axis A will cause axis B to follow; you can test this by doing a small move, checking the position before and after, such as:

   ```
   k1dm3.io CTRL1AUX="SHAB;PRA=200;BGA"
   ```

   (Take care to choose a sensible direction of motion; if the swingarm is retracted, use `PRA=-200`.) If actuator B did not follow actuator A, then the gantry mode didn't work. Try commanding actuator B by hand, then re-check its position:

   ```
   k1dm3.io CTRL1AUX="SHB;PRB=200;BGB"
   ```

   To move actuator B without adjusting actuator A, and without breaking the slaving, use the `IP` (Increment Position) command:

   ```
   k1dm3.io CTRL1AUX="SHB;IPB=xxx"
   ```

3. **Unslaved Mode.**

   To turn *off* gantry mode, slaving axis B to axis A, do:

   ```
   k1dm3.io CTRL1AUX="STAB;GR,0;GM,0"
   ```

   To move both actuators when *not* slaved, use:

   ```
   k1dm3.io CTRL1AUX="SHAB;PRAB=xxx;BGAB"
   ```

7.3 Specific Issues

N.B. When you apply a safety override, check to be sure that nothing except the known issue is being overridden! See chapter 6 for the effects of enabling different safety overrides.

7.3.1 Dispatcher fails to reconnect to its Galil properly.

Indicated by: The dispatcher status, `DISP`*x*`STA` (*x* = dispatcher number: 1, 2, 3), cycles among `Connecting_to_device` → `Authenticating_with_device` → `CommsOk`, but never goes to `Finalized` → `Ready`.

Possible cause: bug in dispatcher

Resolution:

1. Try suspending/unsuspending the dispatcher:

```
modify -s k1dm3 DISPxSUSP="to force reconnect"
modify -s k1dm3 DISPxSUSP=""
```

2. If that fails, simply restart it:

```
k1dm3 restart dispx
```

7.3.2 Galil rejects motion due to Amplifier Fault

After clearing an E-Stop condition, the Galil still prevents motion because of an "amplifier fault".

Indicated by: `SWINGARMXMV` or `DRUMXMV` shows an amplifier fault.

Resolution: The amplifier fault occurred because 48V power had been abruptly removed by the E-Stop. To clear it, turn the motors off, then on. For rotation, use:

```
modify -s k1dm3 ROTATMOO=Off
```

```
modify -s k1dm3 ROTATMOO=On
```

For the swingarm, use

```
modify -s k1dm3 SWINGARMMOO=Off
```

```
modify -s k1dm3 SWINGARMMOO=On
```

7.3.3 Erroneous clamps-not-closed condition

The drum assembly refuses to rotate because the clamps are *erroneously* indicating not-closed. The rotation drive will not operate unless keyword `CLAMPSCLOSED` shows all clamps are closed. As described in section 1.1.2, the clamps-open limit switches are read by the swingarm Galil, but the full-closed signal is read by the rotation Galil. The full-closed signal has to pass through the inter-drum electrical contacts, which means the signal is only available when the inner drum is at `Cass/Stow`, `MirrorUp`, or one of the instrument positions.*

Indicated by: `ROTATXMV` says clamps aren't closed, and `CLAMPSCLOSED` isn't "`A,B,C1,C2`", but visual inspection shows that the clamps are in fact closed.

Possible Causes and Resolution:

- If dispatcher is restarted when not at a powered position, it will be unable to read the clamps-closed signal.

  After visually verifying that the clamps are in fact closed, you can address this using either of the following methods:

  1. You can set an engineering flag to permit rotation when the clamps are not indicating closed:

     ```
     modify -s k1dm3 ROTATENG=XSAFETY
     ```

     Then rotate to an instrument position, and `CLAMPSCLOSED` should update correctly. Clear the XSAFETY flag when you're done (it will auto-clear after 20 minutes):

*Unlike some other values, the `CLAMPSCLOSED` value is not cached across dispatcher restarts because it could lead to serious problems if the clamps are erroneously assumed to be closed.

```
modify -s k1dm3 ROTATENG=none
```

2. Alternatively, you can issue an engineering-level command that directly sets the internal variable which says the clamps are closed:

   ```
   modify -s k1dm3 DISP2DWIM='kset CLAMPSCLOSED 15'
   ```

   (The `CLAMPSCLOSED` keyword is a mask value, with one bit for each clamps; the value 15 means that all clamps are closed.)

- If an electrical problem causes the signal to be in error, you can temporarily rotate using one of the above techniques, always verifying that the clamps are actually closed.
- A clamp limit switch may need adjustment. You can use the `XSAFETY` override, described above, to rotate, after visually verifying that the clamps are in fact closed. Then adjust the limit switch per the procedure in LTN1040 (Ratliff, 2018b).
- Software error. A software error could allow the `CLAMPSCLOSED` keyword to be updated with the incorrect value, perhaps by updating at a time when the signal is not actually available. After visually verifying that the clamps are in fact closed, you can hand-set the correct value:

  ```
  modify -s k1dm3 DISP2DWIM="kset CLAMPSCLOSED 15"
  ```

7.3.4 Clamps are not closed

Drum assembly refuses to rotate because the clamps are *correctly* indicating not-closed.
Possible Causes and Resolution:

Low Air Pressure. The *k1dm3* dispatcher monitors the air pressure at the inner drum, and `ACTIVATE=Swingarm` will throw an error if air pressure doesn't reach 75 psi. However, when commanded to operate the clamps, it will always try to do so regardless of the pressure. (The *k1dm3mon* monitor will raise an alert if the air supply is engaged, but pressure remains below 75 psi.)

Solution: Correct the air supply pressure.

Clamp Is "Soft". The clamps do not require air pressure to stay firmly closed. However, when the clamps are open and the air supply is disconnected — which is the normal state when at Cassegrain — the air pressure can slowly leak to zero over a period of hours. This allows the clamps to sag slightly away from the "IsOpen" limit switch, and the clamp position will report `Unknown`.

In normal operations, this is not a problem. The *k1dm3* service begins all swingarm moves by connecting the air supply and commanding the clamps to open, thus ensuring the clamps are fully open.

Hardware Failure. If a clamp is unable to close due to a failure of plumbing, electronics, or the clamp itself, and the K1DM3 module is installed in the tertiary tower, we assume your goal is to safely remove the module from the tower in order to troubleshoot it on a handling cart.

The general approach is rotate the drum to the `Cass/Stow` (mirror-down) position, because that is always a stable position against gravity-induced rotation of the drum, move the swingarm to the deployed position in the V-blocks, secure the swingarm, and remove the module from the tower.

1. If the module is not at the `Cass/Stow` position, slew the telescope to zenith, where it's safe to rotate regardless of the clamp state, set the `XSAFETY` engineering flag, and rotate to `Cass/Stow`:

   ```
   modify -s k1dm3 ROTATENG=XSAFETY
   modify -s k1dm3 DRUMNAM=Stow
   ```

2. Move the telescope to horizon, and use the "Come-along" procedure (Tripsas et al., 2016) to bring the swingarm into the V-blocks.
3. Remove the module from the tower.

7.3.5 Air supply nozzle will not disconnect.

Possible Causes and Resolution:

Nozzle Sticking to Inner Drum. If the nozzle is left connected to the inner drum for several hours or longer, it can stick to the inner drum.

Therefore, you should always disconnect the nozzle after moving the swingarm. The standard `M3AGENT` sequencer does this, but if you connect the nozzle by directly using the `AIRSUPPPOS` keyword, you must remember to disconnect it before hours have passed.

Solution: In our (limited) experience with this situation, we have found that the nozzle does disconnect after several attempts to cycle its position:

```
modify -s k1dm3 AIRSUPPPOS=Disengage
modify -s k1dm3 AIRSUPPPOS=Engage
```

. . . iterate until success. . .

Solenoid Failure. If the air supply solenoid fails with the nozzle engaged, you must pull the module from the tower and repair or replace the solenoid. Do not attempt to rotate the module with the air supply nozzle engaged.

7.3.6 Need to operate swingarm, mirror facing down

On rare occasions, some repair task or other engineering task requires you to operate the swingarm in its mirror-down orientation on the handling cart or at low elevation in the tertiary tower. To do this:

- Use the comealong procedure (Tripsas et al., 2016) or similar to ensure the swingarm will not fall abruptly out of the V-blocks when the clamps are opened.
- Use the clamps' `XSAFETY` engineering flag to allow them to open despite the low elevation or being on the cart:

  ```
  modify -s k1dm3 CLAMPSENG=XSAFETY
  modify -s k1dm3 CLAMPSPOS=Open
  ```

- Carefully transfer the swingarm's weight from the comealong straps (or your strong arms, if that's what you used) to the actuators.
- You can now operate the swingarm, but you must enable its `XSAFETY` engineering flag first:

```
modify -s k1dm3 SWINGARMENG=XSAFETY
```

After moving the swingarm as needed, clear the engineering flags:

```
modify -s k1dm3 SWINGARMENG=None
```

7.3.7 Software lockout is active

If a software lockout is set, the dispatcher will refuse to move the corresponding stage. Each stage has a keyword named *stage*LCK. If this has any value other than "unlocked", a software lockout is active.

The safety monitor *k1dm3mon* may set software lockouts under some circumstances, but will not remove them, even when the issue clears. When safe to proceed, a human must explicitly clear the condition by clearing the keyword or setting it to `unlocked`. Note: if you clear an assembly's lock, it will also recursively clear any software locks on its members.

To clear software locks on both actuators and all clamps, use

```
modify -s k1dm3 SWINGARMLCK=unlocked
```

To clear software locks on the drum rotation drive, detent mechanism, and air supply to the inner drum, use:

```
modify -s k1dm3 DRUMLCK=unlocked
```

7.3.8 SwingArm Safety System shutdown: arm past deploy point

The swingarm is in the V-blocks, beyond the limit allowed by the Swingarm Safety System, causing an E-stop to be triggered, but is not in a hard stop. The condition will be reflected in the following keywords (these are three equivalent ways of viewing the same information):

`k1dm3.SAFETYBITS1` contains `EncOutOfRange`;
`k1dm3.SAF_EIR` is `No`;
`k1dm3saf.ENC_INRANGE` is `No`.

Recovery requires temporarily modifying the Swingarm Safety System's lower limit so that the E-Stop condition clears, then moving the swingarm into the safe range.

1. The V-block neutral position is with `ARM_A_RAW` = `ARM_B_RAW` = 0 motor encoder counts. The maximum allowed distance beyond 0 is 800 Renishaw counts, or approximately 630 motor encoder counts. Set a temporary new maximum distance using the greater of |`ARM_A_RAW`| and |`ARM_B_RAW`|, plus 200 counts; multiply by 1.27 to convert to Renishaw units, and call this number ***xxx***.

2. Apply the SASS Temporary Override Procedure (section 7.5) using `LoOff=`***xxx***.

7.3.9 Swingarm actuator is in a hard stop

This is like issue 7.3.8, above, but the swingarm is so deeply beyond the V-blocks that one of the swingarm actuators is in a hard stop. In that case, hands-on intervention is required; see LTN1039 (Ratliff, 2018a).

7.3.10 Swingarm Safety System shutdown: misaligned arms

If the swingarm actuators' Renishaw absolute encoders differ by a large enough amount, it will cause the Swingarm Safety System to apply an E-Stop. The condition will be reflected in three keywords (these are three equivalent ways of viewing the same information):

`k1dm3.SAFETYBITS1` contains `EncDisagree`;
`k1dm3.SAF_EAG` is `No`;
`k1dm3saf.ENC_AGREE` is `No`.

The safety system configuration allows a difference between the actuators of up to 0.5mm, or 2048 Renishaw encoder counts. This is large enough to rarely be triggered, yet small enough that the system can handle a series of 0.5 mm shifts without getting into a mechanically unsafe situation. That allows an engineer to experiment with several recovery steps, without entering disastrously bad territory.

Recovery requires temporarily modifying the Swingarm Safety System's maximum allowed difference so that the E-Stop condition clears, then correcting the misalignment. There is, deliberately, no handy one-line tool for changing the safety RIO settings. Instead you must telnet to the safety RIO and type in the Galil command directly. Proceed as follows:

1. The difference between the actuators' absolute encoders, in motor encoder units, is in keyword `k1dm3saf.ARMDIFF`*. Convert this to Renishaw encoder units units by multiplying by 1.27, and then add (say) 1000 extra counts to give you a bit of headroom; call the result ***xxx***.

2. Apply the SASS Temporary Override Procedure (section 7.5) using `Xdifmax=`***xxx***.

7.3.11 SASS: Arms misaligned *and* are past deploy point

The swingarm's arms are misaligned (7.3.10) *and* actuator A is too far into the V-block limit, indicated by `ARM_A_RAW` < -1000. In this case, `ALIGNARMS` will refuse to execute.

The general procedure is to first break the slaving of actuator B to actuator A, and then move the actuators, individually if possible, taking care to not exceed a safe difference between them.

1. Start by clearing the E-stop: apply the SASS Temporary Override Procedure as described in 7.3.10.

2. Now break the actuator slaving:

```
modify -s k1dm3 INNER48V=On
modify -s k1dm3 SWINGARMMOO=On
modify -s k1dm3 SWINGARMASM=Unlinked
```

3. Try to move the actuators individually. Whichever actuator's position is more negative should be commanded the right distance to move away from the V-blocks to align with the other actuator. In the following, ***d*** is the distance to move. Start with 100 counts, and verify that the actuator moves as expected:

*Note that `k1dm3saf.ARMDIFF` is the difference between the absolute encoders, converted to motor encoder units. Don't confuse this with `k1dm3.ARMDIFF`, which is the difference between the actuators' motor encoders

```
modify -s k1dm3 CTRL1AUX="SHA;SPA=200;PRA=d;BGA" or
modify -s k1dm3 CTRL1AUX="SHB;SPB=200;PRB=d;BGB"
```

After gaining confidence that you are making the right corrections, and that only the expected actuator is moving, you can change to using larger steps in `d`. The actuators should now be reasonably aligned.

7.3.12 SwingArm Safety System shutdown: overshot V-blocks

One or both of the swingarm actuators' Renishaw absolute encoders indicate the swingarm position is excessively past the normal position in the V-blocks, causing the Swingarm Safety System to apply an E-Stop. The condition will be reflected in three keywords (these are three equivalent ways of viewing the same information):

`k1dm3.SAFETYBITS1` contains `EncOutOfRange`;
`k1dm3.SAF_EIR` is `No`;
`k1dm3saf.ENC_INRANGE` is `No`.

This is a variation of the preceding issue; follow the directions above to recover.

7.3.13 SwingArm Safety System shutdown: impossible encoder value

One absolute encoder is mis-reading by a ridiculous amount, causing the Swingarm Safety System to apply an E-stop. The absolute encoder values, converted to swingarm motor encoder values, are shown in the `k1dm3saf` keywords `ARM_A_ENE` and `ARM_B_ENE`. One encoder value will be in the normal range, approx 0 . . . 399300, and the other will be far outside that range.

This situation was encountered 2-3 times during development and installation of K1DM3, and then started to occur more frequently after commissioning. It was eventually resolved by replacing the encoder that showed impossible values. In the interim, the workaround was to power-cycle the inner-drum systems, and hope that the encoder would then behave correctly:

```
modify -s k1dm3 INNER48V=Off
modify -s k1dm3 INNER24V=Off
```

Then: ACTIVATE=Swingarm/noinit -or- ACTIVATE=Swingarm

```
modify -s k1dm3 ACTIVATE=Swingarm
```

or

```
modify -s k1dm3 ACTIVATE=Swingarm/noinit
```

7.3.14 Docking pin becomes stuck

If the docking pin becomes stuck in the engaged position, try cycling it several times:

```
modify -s k1dm3 DOCKPOS=Engage
modify -s k1dm3 DOCKPOS=Disengage
```

7.3.15 Docking pin reports incorrect position.

The cause may be a misadjusted limit switch. If you can visually verify that the docking pin is engaged or disengaged, then you can *temporarily* and *carefully* tell the dispatcher to directly set the `DOCKPOS` keyword. Note: this is overridden whenever one of the dock's limit switches activates:

`modify -s k1dm3 DISP3DWIM="kset DOCKPOS 1"` (force Disengaged) *or*
`modify -s k1dm3 DISP3DWIM="kset DOCKPOS 2"` (force Engaged)

Be sure to clear the dock position to the `Unknown` value when you are done with this temporary setting:

`modify -s k1dm3 DISP3DWIM="kset DOCKPOS 0"` (force Unknown)

7.3.16 In-tower switch failure

The K1 tertiary tower contains a switch that is activated when the tertiary module is installed; when it's active, the `LOCATION` mask keyword's `Tower` bit is active. Normally, the *k1dm3* service will respond to TCS tertiary commands and set status information in TCS tertiary keywords, only when `LOCATION` = `Tower`.

If the switch fails so that the in-tower signal is erroneously false, the `M3ENG` engineering keyword's `USEDCS` flag can be set to direct *k1dm3* to respond to TCS, regardless of the `LOCATION` value. (Engineering flags always time out after 20 minutes, but you can reset the timer at any point using the `M3ENT` "ENgineering Timer" keyword.)

If the switch fails and indicates "true" when there is no module in the tower, or indicates true because the old M3 is installed in the tower, then there will be a conflict between the in-tower signal being true, and the module being pinned to the handling cart. The only workaround for this is to make a low-level command that temporarily tells sets the `LOCATION` value to `Cart` only, and clears the `Tower` signal:

`modify -s k1dm3 DISP1DWIM="ksetMacval LOCATION CART"`

This is only a temporary workaround: from time to time events will occur that cause *k1dm3* to reload the `LOCATION` value with the actual input signal values, so it will again show a conflict of `Tower` as well as `Cart`; in that case, you have to re-issue the above modify command.

7.4 Power Outage

The consequence of widespread loss of power will be somewhere between trivial and substantial inconvenience. The optimal recovery strategy depends on whether the swingarm was retracted, deploying, or deployed at the time of power loss, and whether the instrument host (`k1dm3server`) also lost power.

Note: if the facility air compressor is unpowered for an extensive length of time, air pressure may drop, and the drum detent mechanism can disengage. K1DM3 will not attempt to move the swingarm if the detent cannot be engaged.

If computer power was lost, the dispatchers would have been restarted at boot time. *Galildisp* does not start correctly if the controlled Galil is not up when the dispatcher starts, so the following procedure accounts for this possibility.

- The following procedure will recover correctly regardless of whether the module, the instrument host, or both lost power. It takes more steps than are required if one of them did not lose power.
- If possible, ensure the air supply is normal before proceeding.
- Before executing the recovery procedure, open the engineering gui, `k1dm3_gui`, and use it to monitor the recovery progress. From its menu bar, select the entry

  `M3Agent` → `Show log...`
- Before executing the recovery procedure, open the DM3 monitor gui, `k1dm3mon_gui`.

Execute this procedure from the instrument host, `k1dm3server`:

1. Restart the drum dispatcher:

   ```
   k1dm3 restart drum
   ```
2. Restart the dock dispatcher:

   ```
   k1dm3 restart dock
   ```
3. Initialize the drum.
   First, try to initialize the drum without motion:

   ```
   modify -s k1dm3 M3AGENT=NoMoveInitRotate
   ```

   Check the drum calibration state:

   ```
   gshow -s k1dm3 DRUMCAL
   ```

   If the value is not `homed`, then initialize by letting it rotate to read the distance-coded-reference marks:

   ```
   modify -s k1dm3 ACTIVATE=Rotate modify -s k1dm3 DRUMCAL=homed
   ```
4. Put the drum at the stow position, so that the swingarm has power:

   ```
   modify -s k1dm3 DRUMNAM=Stow
   ```
5. Restart the swingarm dispatcher (the `@@` suffix tells the start/stop script to automatically turn on the inner drum power, if it's not already on):

   ```
   k1dm3 restart@@ swingarm
   ```
6. Initialize the swingarm.

   ```
   modify -s k1dm3 M3AGENT=NoMoveInitEncoders
   ```
7. Check if the swingarm is at a secure position:

   ```
   gshow -s k1dm3 SWINGARMNAM
   ```

If the value is `Deploy` or `Retract`, it is secure, and you should skip the rest of this section. If the swingarm is is `Retracted,Unpinned`, then engaging the docking pin will secure it at the `Retract` position:

```
modify -s k1dm3 DOCKPOS=Engage
```

Otherwise, the swingarm position needs to be corrected. Put the telescope elevation to 67° before proceeding. When the telescope is at 67°, initialize the entire module. Reminder: this will finish by doing a move to the current tertiary destination:

```
modify -s k1dm3 M3AGENT=Init
```

This initialization step can fail if the actuators are significantly out of alignment. If so, follow the procedure 7.3.10 to recover.

8. At this point, the system should be fully operational. Check the status indicators on `k1dm3mon_gui`'s tabs. If any are not green checks, inspect the contents to determine additional actions.

7.5 SASS Temporary Override Procedure

When an E-stop has been triggered by the SwingArm Safety System (SASS) Galil, you may need to temporarily override the SASS configuration to permit motion out of the E-Stop condition. Section 7.3 discussed specific issues that you may need to handle using a temporary override; this section provides the details of the procedure.

Each of the settings that you can override is set in a Galil variable. The three settings that you might reasonably need to override are in the following variables:

`Xdifmax` The maximum allowed difference between the two absolute encoder positions. Problem indicated by: `ENC_AGREE` = `No`.

`LoOff` The maximum allowed distance beyond the zero-point in the V-blocks. Problem indicated by: `ENC_INRANGE` = `No` *and* `ARM_`*x*`_RAW` < 0.

`UpOff` The maximum allowed distance beyond the normal retract position. Problem indicated by: `ENC_INRANGE` = `No` *and* `ARM_`*x*`_RAW` > 399300.

As described in section 8.3, the safety RIO operates in isolation. The *k1dm3saf* dispatcher is a custom application (not a *galildisp* instance) that listens for UDP messages from the safety RIO, but doesn't have the ability to send any message to the RIO.

To execute a temporary override, do the following from the instrument host, *k1dm3server*:

1. `% telnet k1dm3-galil-safety`
   Press `Return` a few times to get the Galil's colon prompt, then adjust the necessary limits. Enter one or more of these on separate lines:
   `Xdifmax=`*xxx* *and/or* `LoOff=`*xxx* *and/or* `UpOff=`*xxx*

   This will immediately affect the running code, and the problem should clear. Do not exit this telnet session yet.

2. The E-Stop should now be clear. Cycle the motor power. (This will clear an "Amplifier Fault", if still showing in `SWINGARMXMV`, and in any case power must be on for the next step.):

   ```
   modify -s k1dm3 SWINGARMMOO=Off
   modify -s k1dm3 SWINGARMMOO=On
   ```

3. Set the motor encoders from the absolute encoders:

   ```
   modify -s k1dm3 SWINGARMCAL=Home
   ```

4. Align the actuator arms:

   ```
   modify -s k1dm3 ALIGNARMS=xxx
   ```

   Arm B will move to align with Arm A. Here, *xxx* is any amount greater than *k1dm3*'s `ARMDIFF` keyword. If the actual difference exceeds *xxx*, `ALIGNARMS` will refuse to proceed.

5. Check that *k1dm3*'s `ARMDIFF` is now small — it should be zero, but any amount less than 100 is fine.

6. If either arm has overshot the V-block position or the retract position, move it to the nearest position by commanding

   ```
   modify -s k1dm3 SWINGARMNAM="Deployed,Unclamped" or
   modify -s k1dm3 SWINGARMNAM="Retracted,Unpinned"
   ```

7. Reset the safety system to the normal limit: In the telnet session (start a new one if you exited the previous one), type

   ```
   RS
   ```

   to reset the safety Galil RIO and revert to its normal burned-in configuration. Your telnet session will be immediately terminated by the safety Galil.

Chapter 8

Swingarm Safety System (SASS)

8.1 Background

The K1DM3 swingarm depends on two actuators moving in close synchrony. The actuators can each apply approximately 2000 pounds of force, which is sufficient to permanently deform the swingarm if the two actuators are much more than a few mm out of alignment. Normally, this is enforced by configuring the B actuator to be slaved to the A actuator, using the Galil "gantry" mode. However, if something happens to cause the slaving to fail, allowing actuator A to slew at full speed (40,000 motor encoder counts/s, or $\sim$ 12.4 mm/s) while actuator B is stopped, the actuators would quickly become severely misaligned. This section discusses the safety system that provides an extra layer of protection against problems that can lead to severe actuator misalignment.

8.2 The Safety Monitor and Outputs

K1DM3 protects itself against several kinds of inter-actuator problems by using an independent, standalone Galil RIO that monitors the Renishaw absolute encoders mounted on each of the actuators. Code on board the Galil runs in a loop at about 50 Hz, and as long as conditions remain nominal, the loop toggles a a heartbeat signal; if there is any problem, the heartbeat is not toggled. Should the heartbeat signal stop, both the Electronic Lock-Out (ELO) and Abort signals are activated on board the swingarm Galil (see the Galil DMC-40x0 manual (Galil Motion Control, 2016), and K1DM3 electronics manual (Sandford, 2019), which immediately stops the actuators. From full speed, the actuator brakes will halt the swingarm within 3 mm.

The following checks are applied:

- The actuators must stay within 0.5 mm (2048 Renishaw counts) of each other. This is deliberately several-fold less than the tolerable "several mm," so that the system can handle several failed efforts to recover from a serious problem, without ever reaching the absolute maximum safe difference.
- The absolute encoders must be within the limits.

- Each absolute encoder has an error signal, to indicate a problem with reading the position. This signal must not be active.
- The actuator speed must be kept slow when approaching the v-blocks or docking point. The slow speed limit is 5000 motor counts/sec, or 6350 Renishaw counts/sec. Otherwise, the speed limit is 40000 motor counts/sec, or 50800 Renishaw counts/sec. (The actual speeds commanded by the control system are 3000 motor counts/sec for the slow speed, and 25000 motor counts/sec for the fast speed.)

The Galil RIO reads the encoders at fixed 25 ms intervals, whereas the safety loop is updating at 20 ms intervals. Since the safety system computes the current speed of the swingarm by differencing successive encoder values (and dividing by 25 ms), roughly one in eight speed estimates will erroneously be computed as zero. The safety system requires two above-limit estimates in a row before it triggers an overspeed event, so the worst-case time to trigger is 60 ms: one erroneous zero-speed measurement, then two successive overspeed measurements.

When a problem is detected, the Galil code sets a corresponding output bit and immediately branches back to the start of its monitor loop. That short-circuits the following tests – for example, if the encoders don't agree with each other, the tests for in-expected-range and speed-ok are not done. The sequence of tests is:

- If either encoder's status signal is indicating error, clear the encoders-OK bit and branch to start of monitor loop.
- Otherwise, if the encoder values disagree with each other, clear the encoders-agree bit and branch to start of monitor loop.
- Otherwise, if either encoder is out of its valid range, clear the encoders-in-range bit and branch to start of monitor loop.
- Otherwise, if the encoder speed is above threshold, clear the speed-is-ok bit and branch to start of monitor loop.
- Otherwise, set the all-is-ok bit.

In addition to the heartbeat signal that keeps the Abort and ELO signals from triggering, the safety system provides six output bits to indicate status directly to the swingarm control Galil. The following Galil output bits are used:

1: High if Renishaw encoders agree with each other.

2: High if encoders are within the expected range.

3: High if within speed limit.

4: High if swingarm is in V-block slow zone.

5: High if swingarm is in retract slow zone.

6: The heartbeat signal.

7: Combined all-is-ok output. Although the on-board firmware sets this output bit, it is not wired to anything.

8.3 *k1dm3saf*

All safety functions are implemented entirely on board the system's Galil RIO 47142, and do not depend on any external software. A KTL service, *k1dm3saf*, is provided to monitor the system's status and the Renishaw absolute encoder values, but never initiates any communications with the RIO, which stands alone in splendid isolation.

The RIO sends its "Data Record" (see the `DR` command in the RIO-47xxx manual) every second to UDP port 1222 on the instrument host, `192.168.23.100`, and sends additional messages with the current speed (computed by differencing successive encoder positions), maximum speed in the past second, count of overspeed conditions, and some timestamps. The *k1dm3saf* dispatcher listens for the UDP messsages from the Galil, and reports the status via its keywords (see chapter 13 for the complete keyword list). The RIO doesn't care if anyone is listening for these messages, so it is unaffected by whether the *k1dm3saf* service is running.

In addition to reporting the safety system status, the *k1dm3saf* service provides the Renishaw encoder absolute positions in one pair of keywords, plus a second pair of keywords that give the absolute encoder values in the equivalent swingarm actuator motor encoder values. This allows the swingarm actuators to be "homed" without any motion, by simply reading the keyword values from *k1dm3saf*, and setting the correct motor encoder values in the swingarm Galil.

When investigating an Electronic Lock-Out triggered by the safety system, the most useful keywords are probably the following:

- `CT_NOTOK`. This counts the number of times that one or both absolute encoders have an active error or warning flag. The counter increments each time the combined encoders' error and warning flags change from all-inactive to any active, and is reset to zero each time the RIO is powered up.

- `CT_DISAGREE`. This counts of the number of times that one or both absolute encoders have no active error or warning flags, but the encoders disagree with each other. The counter increments each time the encoders go from in-agreement to not-in-agreement, and is reset to zero each time the RIO is powered up. During power-up, it is common for the system to briefly pass through an encoders-disagree state during initialization, and therefore the apparent startup state is typically `CT_DISAGREE`=1, instead of 0.

- `CT_RANGE`. This counts of the number of times that the absolute encoders agree with each other, but are not in the valid range for the swingarm. The counter increments each time the encoders transition from in-range to not-in-range, and it is reset to zero each time the Galil is powered up.

- `CT_SPEED`. This keyword is a counter of the number of times that an over-speed condition is encountered. It is reset to zero each time the Galil is powered up.

- `ENCBAD0`, `ENCBAD1`. Whenever one of the above conditions occurs and the corresponding counter is incremented, the encoders' "V-block distances" are latched into these keywords. That is, the latched value is the distance of the encoder from its nominal in-V-block position, so it ranges from 0 counts when in the V-block to approximately 507,000 counts when retracted. (Due to the encoders-disagree state that is encountered during initialization, as described under `CT_DISAGREE`, above, these will typically have a non-zero startup value, which you should ignore.)

The *k1dm3saf* service also provides enumerated keywords that show the state of the encoders at the current moment in time: `ENC_AGREE`, `ENC_INRANGE`, `ENC_OK`, and `SPEED_OK` correspond to the counter keywords `CT_xxx`. However, error conditions can be transient, not persisting long enough to be captured in a keyword value. For example, an overspeed condition can occur, triggering an ELO, and the swingarm brought down to a stop (or just a safe speed) before the safety RIO's next broadcast of state data to the *k1dm3saf* dispatcher. As a result, the current-state data would not show that an error is active, whereas an error counter, having incremented, would show that the error occurred.

Chapter 9

The K1DM3 Monitor

The *k1dm3* KTL service is supplemented with a monitor daemon, *k1dm3mon*, that raises an alert when any of a range of possibly undesirable conditions occurs. For certain especially unsafe conditions, it also issues corrective commands to *k1dm3*.

The *k1dm3mon* service uses the *emir* application (Deich, 2014), which provides a standardized package for monitoring and responding to any conditions of interest.

As of Oct. 2018, the following checks are made:

Drum

- Alert if the drum dispatcher has a problem connecting to its Galil.
- Alert if slews are not done a maximum speed.
- Alert if the motor is on for more than 3 minutes.
- Alert if the detent disengages while the air supply is engaged; in which case also command the detent to re-engage, and set a software lockout forbidding rotation.
- Alert if the preferred move direction isn't `Quick` (shortest time).

Swingarm

- Alert if the swingarm dispatcher has a problem connecting to its Galil.
- Alert if either motor is on for more than 3 minutes.
- Alert if a motor-overtemp error occurs
- Alert if the swingarm is being deployed at other than El=67°.
- Alert if one of the SwingArm Safety System halt-motion conditions is tripped.
- Alert if the swingarm is in the tower, and is neither in the Retract nor Deploy positions, nor in the midst of a commanded motion.

Dock

- Alert if the dock dispatcher has a problem connecting to its Galil.
- Check that the dock is engaged if the swingarm is retracted.
- Check that the dock is retracted while the swingarm is in motion.
- Check that the swingarm dispatcher is the normal version that controls the dock, and is not the special engineering version that ignores the dock.

Other

- Monitor the air pressure at the inner drum.
- Monitor sources of heat:
  - Alert if any controllable power supply (48V drum motor power, 48V swingarm actuator power, or 24V inner drum power) is on for at least 5 minutes.
  - Alert if any solenoid control bits are in an active state for at least 5 minutes.
- Alert when a software lockout is active.

Chapter 10

Galildisp Configuration

All of the *k1dm3* service's dispatchers are instances of *galildisp*. *Galildisp* is a general-purpose Galil dispatcher; each instance of *galildisp* controls a single Galil DMC-*xxxx* or RIO-*xxxxx*.

Galildisp is mostly driven by configuration files, along with custom code as needed to supply functionality not built into the application. This section explains the layout and use of *k1dm3*'s configuration files, so that you can understand its behavior and modify it if desired.

10.1 Major Elements of Control

10.1.1 Motor Axes and Digi-Axes

The main unit of control in *galildisp* is the *axis*. Every axis has a name, and a collection of standardized keywords that begin with that name and end with a 3-letter suffix.

Galildisp axes come in three flavors:

1. Motor axes directly represent actual Galil axes, and are named A..H, in accordance with the usual Galil nomenclature. For example, the K1DM3 rotation motor is connected to Galil axis A, and is named the `ROTAT` stage. It has keywords such as `ROTATENC` (the motor encoder value), `ROTATRAW` (the "raw" load encoder value), and `ROTATSPD` (current speed in load encoder counts/sec).

2. Digi-axes represent components that are controlled with digital outputs, and provide feedback through digital inputs. They are named D0, D1, D2.... For example, K1DM3's rotation detent mechanism (digi-axis D1, stage name `DETENT`) uses two output signals to drive the detent mechanism, and two input switches to indicate if its fully retracted or fully engaged. The `DETENTPOS` keyword is used to both set and report its position.

3. Combo-axes, or assemblies, are collections of other axes, which can be motor axes, digi-axes, and/or other combo axes. Their axis names are C0, C1, C2,.... Their status keywords are computed from the status of their component axes. Some of their actions are automated (e.g. a stop command sent to the assembly's *aaa*`STP` keyword is simply repeated to each component *xxx*`STP` keyword). Others actions are

implemented using rules in a configuration file (e.g. a position keyword of an assembly is generally implemented by defining rules in a configuration file that give a sequence for commanding other keywords).

10.1.2 Non-Axis Keywords

Many analog and digital I/O keywords don't belong to any axis. Examples include temperature keywords that report data from analog temperature sensors, and digital outputs that turn on or off power supplies.

Compound keywords are computed from combinations of other keywords, such as the `ACTIVATE` keyword that reports whether the module is in one of the standard configurations for rotating, operating the swingarm, or observing.

10.1.3 Sequencers

A sequencer keyword is used to issue sequences of commands to other keywords — it's comparable to using an ordinary standalone shell script to issue modify commands from the command line, and indeed it may be implemented using a shell script. Sequencer keywords are either assembly or compound keywords, because those are the *galildisp* types that can execute user-defined sequences of modify's of other keywords. When a ktl write (modify) is invoked on a sequencer keyword, its user-defined command sequence is executed.

The configuration file either directs the dispatcher to follow simple "modifyRules" instructions placed in the configuration, or directs the dispatcher to invoke an external script specified in the configuration file.

A sequencer keyword has significant advantages over a simple standalone shell script, because the sequencer keyword is acting as part of the dispatcher, rather than acting as a client. Its advantages compared to a standalone shell script include:

- If the sequencer keyword is invoked by a manual action (e.g. a hand paddle) all modify's from the sequencer script will also run in a manual context, rather than being locked out because remote control has been locked out by the hand paddle.

- Status keywords can be automatically filled in by *galildisp*, including copying the script's *stdout* text. This greatly simplifies writing a sequencer script that is fully part of the dispatcher. Other clients can monitor the progress of the squencer script, by monitoring its optional message and error keywords.

- Cancellations can be demanded by *galildisp*, in response to a new countermanding keyword request. In contrast, *galildisp* doesn't "know" when there is a standalone script issuing commands and doesn't have a way to tell it to stop.

- Command timeouts can be enforced by *galildisp*, including the ability to kill an external sequencer application and cancel any ongoing modify's.

- Galildisp runs each external application under its own session id, so that it's easily managed and signalled as a unit.

10.2 Dispatcher and Controller Information

Information about dispatcher x ($x = 1, 2, 3$) is available in the keywords beginning `DISP`*x*; information about the Galil controller is available in the keywords beginning `CTRL`*x*.

For the most part, *galildisp* dispatchers operate independently with limited interdependence on each other. However, all dispatchers for a given service maintain a "back channel" communications link to support the following kinds of messaging:

- A dispatcher can monitor other dispatchers' keywords, as needed. For example, the `ACTIVATE` keyword is served by Dispatcher 2, but its value is determined by the values of keywords from all three dispatchers, so Dispatcher 2 monitors the necessary keywords from the other dispatchers.
- A dispatcher can execute an internal modify of any other dispatcher's keywords, which it does by sending a modify request across the back channel. For example, the `M3AGENT` sequencer is provided by Dispatcher 2, but has to issue modify commands to keywords of all dispatchers.
- Assemblies need to monitor state data for all their component stages. When an assembly is composed of stages from several dispatchers, the other dispatchers transmit the essential state data across the back channel.

This cross-dispatcher monitoring and modifying of keywords is handled transparently by the dispatchers, so that you can use them almost as easily as purely-local keywords.

10.3 Configuration Files

Each *galildisp* instance is configured using two sets of files: a dispatcher configuration file named *svc*`_dispatch_`*n*`.conf` (e.g. `k1dm3_dispatch_2.conf`), and XML files that define the keywords used in the service. The dispatcher *.conf* files are used only by the dispatcher, but the XML files are needed by both the dispatcher and the KTL client library.

For historical reasons — namely, *galildisp* dates back to a time when XML files weren't used by the KTL client library — the dispatcher mostly uses configuration files that are generated at build time from the XML, rather than directly reading the XML. Here, we focus on the files you may need to edit, and ignore the generated files that litter the build directories.

K1DM3 uses two flavors of macro substitution. Files ending in `.sin` are macro-substituted using `$RELDIR/etc/subst.tcl`. (This is not a Tcl-specfic tool; the .tcl suffix is used merely to distinguish this application from similar commands named `subst`.) `Subst.tcl` extracts definitions from the Makefile and substitutes them into the `.sin` file. The rule for converting `foo.sin` into `foo` is built into the Kroot build system, so it isn't explicitly called out in the Makefiles. The second kind of macro substitution invoked by the K1DM3 Makefiles is to apply `$RELDIR/bin/wapp` to each file with a `.in` suffix. *Wapp* uses macro definitions from one or more `.defs` files.

Wapp is a flexible and powerful tool, and a detailed description of it is beyond the scope of this document. A complete man page is available by invoking "`wapp -H`." However, when reading or editing the `.defs` files, you should at least know that it can switch on the fly between C-style (`#define var value`) and Makefile-style (`var = value`) macro definitions; by default it applies variable substitution to each item matching `$(var)`, and expression

substitution to each item matching `$((...))`. In some places, discussed below, the use of `$(var)` conflicts with similar expression parsing built into *galildisp*, and in that case we adjust *wapp* to use `$.var.` instead.

Finally, in a few cases, there are files named `foo.sin.in`, in which case *wapp* is applied to generate the `foo.sin` file, and then `subst.tcl` is used to generate the final file `foo`.

10.3.1 Template and Definition files

Most of the XML files are generated by doing macro-substitution into template files. This is for several reasons. First, XML is notoriously verbose, and unpleasant to maintain entirely by hand. Second, when *galildisp* is modified in ways that affect the keyword XML, it is far easier to update a single template file, than to hand-edit each affected XML file. Third, the same value may be used in many configuration locations — including both the *.conf* and XML files — and we only define each value in a single *.defs* file.

Warning: don't lose your edits by editing a generated file, which will of course eventually be overwritten. If you see files named *foo* and *foo.in*, it's pretty obvious that the former is generated from the latter. More precisely, any file that is not stored in svn (use "`svn info` *`file`*" to check) is either a generated file or some bit of cruft, and is not one of the files you should be editing to modify the *k1dm3* configuration.

10.3.2 The *.defs* and XML files

The XML and `.defs` files are found in `svn/kroot/kss/K1DM3/k1dm3/xml/`. XML entries that are intended to be used only by the dispatcher are either in the `dispatcher` subdirectory, or are placed in `serverside` nodes inside the main XML entries. The XML files follow the rules of the *dtune* KTL client library. You do not need to directly edit any XML files in order to modify a *k1dm3* dispatcher XML configuration; you need only edit the `.defs` files, or, less often, the corresponding `.xml.in` files.

Generally, each axis has its own well-commented `.defs` file, and the Makefile applies the definitions from *`FOO`*`.defs` to *`FOO`*`.xml.in`, to generate *`FOO`*`.xml`. As a general rule, any file `xml/Template/`*`FOO`*`.xml.in` is simply a copy of a `galildisp/Template` file.

Most of the per-axis files in turn include the `Common.defs` file. If you are uncertain precisely how an XML file is generated, simply delete it and type "make" to recreate it. If you want to know more about exactly what *wapp* is doing, you can hand-execute the *wapp* command, adding one of the options `-ed` (print expanded variable definitions), or or `-eed` (print expanded variable definitions plus expressions).

This distributed collection of configuration data can sometimes make it difficult to find the original definition for a value, and it sometimes takes some judicious use of *grep* and file inspection to find the source of a definition.

10.3.3 The *.conf* files

In addition to the XML files, each *galildisp* instance needs its *`svc`*`_dispatch_`*`n`*`.conf` file. The directory `svn/kroot/kss/K1DM3/k1dm3/configs/` contains the source for these files.

The Makefile generates the `.conf` file for dispatcher `n` by applying *wapp* to each file `k1dm3_dispatch_`*`n`*`.conf.in`. This pulls in many *`xxx`*`.cfg.in` files — one `xxx.cfg.in` file for each axis, and compound keywords are also configured in `.cfg.in` files. Configuration

blocks that are re-used in several places (such as the four clamps, which share a great deal of configuration), are further split out into sub-`.cfg.in` files.

10.4 Custom Dispatcher Code

K1dm3 has some features that are not built into *galildisp*, and those are handled in custom code. For example, the interactions with TCS are entirely handled in custom code.

Custom code files are in `svn/kroot/kss/K1DM3/k1dm3/dispatcher/`, and are named `k1dm3_dispatch_`*n*`.tcl` file.

Chapter 11

Utility Programs

The K1DM3 software is supplemented with a few utilities to help with monitoring or troubleshooting.

11.1 *k1dm3.summarize*

The *k1dm3.summarize* utility provides a quick look at recent operations of K1DM3. By default, it shows a brief summary of the sequencer actions since noon yesterday, in which each entry shows the start time, end time, and position, and any errors that caused the sequencer to throw an error. If a sequence includes moving the swingarm, then telescope elevation is included in the summary. The summary of sequences is followed by a very terse list of the sequence of just the positions reached by the sequencer. Finally, at the very end, is a single line that indicates the amount of compression in the summary: the number of lines read from the keyword history database, the number of output lines generated from that input, and the number of additional output lines generated by reading supplementary data.

It is a simple-minded tool, using the `M3RUN` keyword to show when the sequencer was running, and the *dcs1* keywords `TERT{INIT,MOVE,HALT,ABRT,STBY}C` to indicate what action was executing. If the action was triggered by a direction invocation of `M3AGENT`, *k1dm3.summarize* is not sophisticated enough to figure that out.

Example 1. The basic output for a night looks like this:

```
% k1dm3.summarize

2018-12-29T14:07:50.72  Begin action=Init   Start=Dest=Cass
2018-12-29T14:07:50.98  Begin EL=0.00
2018-12-29T14:08:09.63  End   EL=0.00
2018-12-29T14:08:09.63  End   action=Init   Position=Cass

2018-12-29T18:01:32.65  Begin action=Init   Start=Dest=Cass
2018-12-29T18:01:32.92  Begin EL=45.00
2018-12-29T18:01:50.08  End   EL=45.00
2018-12-29T18:01:50.08  End   action=Init   Position=Cass
```

```
2018-12-30T03:59:01.46  Begin action=Stby
2018-12-30T03:59:02.93  End   action=Stby   Position=Cass

~ Position Summary ~

2018-12-29T14:08:09.63  Cass

324/10 + 4
```

At the end of each sequence, the summarizer prints warning(s) if any of the following are true: an inner drum power supply was left on; the outer drum 48V supply was left on; there are active solenoid outputs; the air supply was left connected; or there is a software lockout. Note that it is normal, and harmless, to have one or more warnings when a move or init sequence is cancelled by another sequence: the conditions will immediately be corrected by the next sequence. It is not normal for any of these conditions to persist for an extended time, and the summarizer will exit with non-zero exit status if any warnings are active at the end of the summary period.

Options: The option `-h` prints help information. The `-s` option suppresses the main body of output, printing only the very terse position summary. The `-t` option prints the terse position summary at the top, before the main body, which is a nice layout for emails. The `-`*`n`* option stops output after encountering *n* End-of-Sequence markers. The `-v` option adds the `M3MSG` keyword to the listing. Additional `-v`'s add yet more keywords to the listing.

The data are extracted from the keyword history using *gshow*, and you can use its `-date` and `-window` options to adjust the selected date range; see `gshow -h -date` and `gshow -h -window` for their usage details.

Example 2. Here is a sample from a move that failed. Note that the high-level error information is included in the summary:

```
 % k1dm3.summarize  -date 2018-12-19T10:00 -date "dec 19 10:02:30"

2018-12-19T10:01:50.86  Begin action=Move   Start=Cass, Dest=RNas
2018-12-19T10:01:51.11  Begin EL=67.00
2018-12-19T10:02:09.20  ERR_DOCK_NOT_DISENGAGED /
2018-12-19T10:02:10.27  ERR_CHILD / Error in program: child process exited abnormally (1)
2018-12-19T10:02:10.27  End   EL=67.00   ELMIN=67   ELMAX=67
2018-12-19T10:02:10.27  End   action=Move   Position=Unknown
```

Adding a `-v` option makes the output more verbose, by including the same information that would have been displayed in *Tcsgui* during the move.

Example 3. Adding verbose flag `-v`:

```
 % k1dm3.summarize -v -date 2018-12-19T10:00 -window 2.5min
2018-12-19T10:01:50.86  Begin action=Move   Start=Cass, Dest=RNas
2018-12-19T10:01:51.72  ACQUIRING:   Dest is rnas
2018-12-19T10:01:53.67  SLEWING:     Activate Swingarm
2018-12-19T10:01:54.19  SLEWING:     check detent
```

```
2018-12-19T10:01:54.38  SLEWING:     check air
2018-12-19T10:01:55.98  SLEWING:     Inner drum power
2018-12-19T10:02:01.52  SLEWING:     Unsuspend swingarm
2018-12-19T10:02:01.75  SLEWING:     Unsuspend Swingarm
2018-12-19T10:02:04.46  SLEWING:     Disengage dock
2018-12-19T10:02:06.12  SLEWING:     Set swingarm from abs encoders
2018-12-19T10:02:07.61  SLEWING:     Slave actuators
2018-12-19T10:02:08.30  SLEWING:     Actuator power
2018-12-19T10:02:08.82  SLEWING:     Actuator init
2018-12-19T10:02:09.05  SLEWING:     SWINGARMINI : fail
2018-12-19T10:02:09.16  FAULT:       SWINGARMINI : fail
2018-12-19T10:02:09.20  ERR_DOCK_NOT_DISENGAGED /
2018-12-19T10:02:09.36  FAULT:       ACTIVATE : fail
2018-12-19T10:02:09.63  FAULT:       Swingarm not activated
2018-12-19T10:02:10.27  ERR_CHILD / Error in program: child process exited abnormally (1)
2018-12-19T10:02:10.27  End   action=Move   Position=Unknown
```

If you use two `-v` flags for very verbose output (`-vv` or `-v -v`), the output will include all the normal sequencer messages that go into keywords `ACTIVMSG` and `M3MSG`, plus the *dcs1* keywords `TERTERRS`, `TERTSTST`, and `TERT`*xxxx*`C`:

```
 % k1dm3.summarize -v -v -date 2018-12-19T10:00 -window 2.5min
2018-12-19T10:01:50.74  TERTDEST: rnas
2018-12-19T10:01:50.84  TERTMOVEC: yes
2018-12-19T10:01:50.86  M3MSG: m3.move: begin
2018-12-19T10:01:50.86  M3MSG: $RELDIR/sbin/k1dm3/m3.move TERTMOVEC 1 YES Yes
2018-12-19T10:01:50.86  M3RUN: true
2018-12-19T10:01:50.86  Begin action=Move   Start=Cass, Dest=RNas
2018-12-19T10:01:50.93  M3MSG: m3.move (3265) executing on behalf of TERTMOVEC
2018-12-19T10:01:51.11  TERTMOVEC: no
2018-12-19T10:01:51.20  M3MSG: Starting at k1dm3.TERTPOSN=Cass
```

[...*many elided lines*...]

```
2018-12-19T10:02:08.82  TERTERRS: Actuator init
2018-12-19T10:02:08.85  ACTIVMSG: setting SWINGARMINI = yes
2018-12-19T10:02:08.90  ACTIVMSG: modify SWINGARMINI="yes" failed: ERR_DOCK_NOT_DISENGAGED
2018-12-19T10:02:08.96  ACTIVMSG: setDCSstatus (dcs1):  "216" / "SWINGARMINI : fail" / "fault" /
2018-12-19T10:02:09.05  TERTERRS: SWINGARMINI : fail
2018-12-19T10:02:09.16  TERTSTST: FAULT
2018-12-19T10:02:09.20  ACTIVMSG: Error in program: child process exited abnormally
2018-12-19T10:02:09.20  ACTIVMSG:
2018-12-19T10:02:09.20  M3ERM: ERR_DOCK_NOT_DISENGAGED /
```

[...*more elided lines*...]

```
2018-12-19T10:02:10.27  M3RUN: false
2018-12-19T10:02:10.27  End   action=Move   Position=Unknown
```

You can add a third `-v` for yet more verbose output.

Finally, you can add additional *k1dm3* keywords to the end of the *k1dm3.summarize* command line, and these keywords will also be printed. In the above example, where the docking pin failed to disengage correctly, you might want to repeat the command with all `DOCK*` keywords added to the output:

```
% k1dm3.summarize -date 2018-12-19T10:00 -window 2.5min DOCK%
```

[...]

11.2 *k1dm3.status.email*

This simple shell script is designed for use as a cron job. It is invoked as

`k1dm3.status.email [-x]` *`addr ...`*

It runs *k1dm3.summarize* and emails the results to the addresses given on the command line. The `-x` option suppresses the email except when the summarizer finished with warning(s) about power supplies or solenoids left active, or an active software lockout.

11.3 *k1dm3.io*

This little one-liner script simply invokes the *service.io* application, which in turn is just another interface to *show* and *modify*. It's useful because it simplifies a common engineering usage when digging into a running *galildisp* instance.

Galildisp has a number of keywords that accept a command or request by writing the keyword, and then the response is retrieved by reading the same keyword. For example, you can write Galil code directly to the Galil, by modifying the keyword `CTRL`*x*`AUX`. The dispatcher will copy the string to the Galil, and all text received from the Galil in the next 20 ms is assumed to be a response to that command, and is copied back into the `CTRL`*x*`AUX` keyword, which can then be read from the keyword.

If you use `k1dm3.io` *`kwd...`*, it simply invokes *show* to display the keywords. However, if you use

`k1dm3.io` *`kwd='value'`*

then the *value* is written to the keyword, *k1dm3.io* waits 100 ms for a response, and then reads the updated keyword value. This is a very convenient way to interact with a running dispatcher.

Example:

```
% k1dm3.io CTRL1AUX='MG "hello"'
modify -s k1dm3 CTRL1AUX='MG "hello"'
setting CTRL1AUX = MG "hello" (wait)
hello\n:
```

(The italicized lines are printed by *k1dm3.io* and *modify*; the text "`hello\n:`" is the response from the Galil, which we can read from keyword `CTRL1AUX`.)

See `service.io -h` for more information on how to use this command.

Chapter 12

k1dm3 KTL Service Keywords

This section summarizes all the keywords provided by the *k1dm3* KTL service. Additional information — such as data type, data range, allowed values — is available using *gshow* to query the keyword data, using its various help options.

The *k1dm3* service is made up of three dispatchers:

- Dispatcher 1 controls the swingarm Galil.
- Dispatcher 2 controls the drum Galil.
- Dispatcher 3 controls the dock Galil.

12.1 Dispatcher Keywords

Each *galildisp* dispatcher has a set of about 25 standard keywords to describe the dispatcher itself. (Some of these keywords aren't used by the *k1dm3* dispatchers, and they are basically just noise keywords. Sorry.) In the following keyword names, each ***x*** stands for the dispatcher number (1, 2, 3):

`DISP`***x***`_BITGP` Space-separated list of all "bitgroups" controlled by this dispatcher. For *k1dm3* , this is always empty: *k1dm3* has no bitgroups.

`DISP`***x***`_COMBO` Internal to *galildisp*, each assembly is modeled in a manner similar to a motor axis. This keyword gives an alternating list of each assembly axis name and its assembly name.

`DISP`***x***`_COMBX` List of top-level assemblies: assemblies that are not themselves components of other assemblies.

`DISP`***x***`_DIGI` Space-separated list of all digistages controlled by this dispatcher.

`DISP`***x***`_MOTOR` Space-separated list of all motor stages controlled by this dispatcher.

`DISP`***x***`BCH` Used at dispatcher startup, to set up the back channel communications between dispatchers.

`DISP`***x***`CLK` Dispatcher clock: seconds since dispatcher began running.

`DISP`*x*`DBG` Dispatcher debug level. A KTL bitmask. Used for debugging *galildisp* itself.

`DISP`*x*`DISABL` Disables the StdStage system for controlling axes, and instead log each call. Used for debugging *galildisp* itself.

`DISP`*x*`DWIM` A debug keyword: semantics change on a whim; it does whatever I want it to do to help in debugging. Not for ordinary users.

`DISP`*x*`ERR` Dispatcher's last error number associated with the overall dispatcher state. Closely related to `DISP`*x*`MSG`.

`DISP`*x*`HOM` Name of host on which the dispatcher is running.

`DISP`*x*`HUP` Tells dispatcher to do typical SIGHUP-like actions, e.g. reload Raw/Ord/Nam mappings from their config files. Recommendation: do not use this; instead, directly write the *stage*`RON` keyword to perform updates (these will be automatically copied to the runtime config file).

`DISP`*x*`LDF` A debug keyword: names a plain file to be source'd by the dispatcher. The path is relative to `$HOME/`*svcnam*`/`*dispnam*; the file must be owned by the same account that is running the dispatcher; and the file must not be group or world writable.

`DISP`*x*`LOG` Dispatcher *x* log file name. For *k1dm3*, this is `key:k1dm3-`*x*`log`, which says to look up the key in the the MUSIC services file; that file says the key's value is `syslog:k1dm3-`*x*`[%p]:local6`; this means to log using *syslog(3)*, with facility `LOCAL6`, and "program name" `k1dm3-`*x*. In turn, the syslog configuration file `/etc/rsyslog.conf` says that messages with that program name are recorded in `$KROOT/var/log/`*programname*. (Whew).

`DISP`*x*`MSG` K1DM3 dispatcher status message. Last status message associated with overall dispatcher state, typically indicating communications status with its networked device.

`DISP`*x*`NOTE` A convenience keyword: a user can write anything of interest in this keyword. Not used by the dispatcher itself.

`DISP`*x*`PID` Dispatcher *x* process id.

`DISP`*x*`PSE` When written with value n, the modify command waits (pauses) for n milliseconds, then returns. The dispatcher itself doesn't pause; it's provided to allow scripts to pause in a way that can be interrupted just like any modify command.

`DISP`*x*`REQ` All modify requests for dispatcher *x* are recorded in this keyword as "*kwd binary-value*". This is done just before executing the request. By querying the keyword history database, one can retrieve a timestamped record of all commands to the dispatcher.

`DISP`*x*`RSS` The RSS keyword holds the current memory Resident Set Size, as returned by *getrusage(2)*.

`DISP`*x*`SCK` This keyword is incremented and broadcast following all other keyword updates that are triggered by the Galil's most recent data record. Thus, when this keyword is broadcast, all other keywords are in sync. (`SCK` == Self-Consistent Keywords).

`DISP`*x*`SHUTDN` Shutdown command. Tells dispatcher to exit, after stopping any software running on the Galil.

`DISP`*x*`STA` Overall dispatcher software status. Not for showing per-stage hardware, electronics, or on-board Galil trouble. Typically used for showing network communications status or other issues affecting overall operation of dispatcher. It always transitions to `Finalized`, then `Ready` as its final steps after connecting to the Galil.

`DISP`*x*`SUSP` When written with a non-blank string, this tells the dispatcher to suspend communications with Galil, and reject stage and I/O modify commands. When written with a blank string, tells dispatcher to resume normal functioning. The `DISP`*x*`STA` keyword will show `Suspended` when dispatcher is suspended.

`DISP`*x*`THREAD` A debugging keyword. *Galildisp* uses threads of execution on board the Galil. This keyword is a space-separated list of alternating Galil thread number and the dispatcher's terse thread name.

12.2 Controller Keywords

Each *galildisp* dispatcher has the following set of keywords to describe the state of the Galil controller. In the following keyword names, each *x* stands for the dispatcher number (1, 2, 3):

`CTRL`*x*`ADR` Controller *x* motor controller address (IP address or path).

`CTRL`*x*`AUX` K1DM3 Controller *x* auxiliary I/O method. Anything written to this is copied, verbatim, to the Galil; anything read in response to this is returned in this keyword.

`CTRL`*x*`CLK` Galil Controller *x* pseudo-clock: the last-received value of the sample counter of the Galil.

`CTRL`*x*`ERR` The last error number associated with the Controller *x* state. Closely related to `CTRL`*x*`MSG`.

`CTRL`*x*`GSS` The `GSS` keyword provides supplementary status info gleaned from periodic polling of the Galil. For *k1dm3*, this is used to test and report if the Abort input is active. Ordinary users don't need to pay attention to `CTRL`*x*`GSS`, because the keyword value is monitored by the standard per-stage reporting keywords.

`CTRL`*x*`MSG` The last status message associated with overall controller*x* state. Closely related to `CTRL`*x*`ERR`.

`CTRL`*x*`THD` The Controller *x* thread status. A Galil DMC-40x0 controller has 8 threads available, and *galildisp* uses several of them. This shows the thread run state: the value is a list of *sta_0 sta_1 sta_2 ... sta_7*, where *sta_i* is 1 [0] according as the thread is [is not] running. Primarily for debugging the dispatcher.

12.3 Digi-Axis Keywords

Stage Prefixes: DETENT, AIRSUPP, CLAMP_x

A digi-axis stage *dax* has up to 18 standard keywords. The most important keyword is ***dax*POS**, which is used to command and report the stage position. Other important keywords are ***dax*XMV**, which reports any reasons that the stage is forbidden from moving; and ***dax*ERR** and ***dax*ERM**, which report any error code and error message.

The full set of keywords is:

***dax*CMT** An arbitrary comment, for use by clients of the service to exchange or record info. Not used by the dispatcher itself. See also ***dax*TAG**.

***dax*CON** `Manual` means a manual request is executing; otherwise the value is `Auto`.

***dax*DCC** DisConnected Cables message. Blank if all cables are connected. Contains the string "`motion disabled`" if the disconnected cable is required for stage motion.

***dax*ENG** Enable engineering modes. Not used by the *k1dm3* digi-stages.

***dax*ENT** Time before engineering flags automatically clear.

***dax*ERM** Error Message. The `ERM` keyword is tied closely to the `ERR` keyword: it is intended to always have the error message associated with the error in `ERR`. Unlike the `MSG` keyword, the `ERM` keyword is not intended to receive other, non-error messages.

***dax*ERR** Last error number — it is the last KTL status value. Usually, this is closely related to the `MSG` keyword.

***dax*LCK** Software lock switch. If set to "`unlocked`", motion keywords can be modified; otherwise, they cannot. As a convenience, setting it to the empty string "" is shorthand for "`unlocked`".

***dax*LIM** Stage "limit" status. A digi-stage's position is determined by a collection of one or more switches. If no switch is active, the value is "`Not in a limit`"; if multiple switches are active, the value is "`Err Multiple Active`"; otherwise, the value indicates which single switch is active.

***dax*MSG** Message keyword. This keyword receives the same error messages that are copied to the `ERM` keyword, but also receives non-error messages, such as status messages while executing a sequence.

***dax*POS** Name of the current ordinal position. Writing this value commands the digi-stage to move to that position. Valid values are:

`AIRSUPP` stage: `Disenagaged`, `Engaged`

`DETENT` stage: `Disenagaged`, `Engaged`

`CLAMP_`*x* stage: `Closed`, `Open`

***dax*PSE** Provides a "pause" capability: when written with value n, does nothing for n ms, then returns. Identical to the `DISP`*x*`PSE` keyword; see the description of that keyword for details. Only provided with `DETENT`; not provided with `AIRSUPP` or `CLAMP_`*x*.

***dax*RCD** (Remote Control Disablers). Blank if remote keyword control can be enabled; else a terse explanation.

***dax*STA** Control status: `Locked`, `Ready`, `Moving`, `Fault`

***dax*STP** Request stop. When written with any value: aborts the sequence of output commands that move a stage. When read: returns the last value written.

***dax*TAG** An arbitrary comment, for use by clients of the service to exchange or record info. Not used by the dispatcher itself. See also ***dax*CMT**.

***dax*TRG** Name of the target position, updated each time a new value is written to the `ORD` keyword.

***dax*XMV** Shows all the reasons that this assembly (probably) cannot move. Some reasons are not necessarily disabling — for example, a component stage may be in a limit that prevents motion in one direction only.

12.4 Motor-Axis Keywords

*Stage Prefixes: ROTAT, ARM_A_, ARM_B_*

K1DM3 has three* motor stages — `ROTAT`, which is the drum rotation drive; and `ARM_A_` and `ARM_B_`, which are the two swingarm actuators. In general, *galildisp* tries to treat all motor axes alike, each with a large set of similar keywords, differing only in details. In practice, the `ARM_B_` stage is always slaved to `ARM_A_` except when initializing the actuators, and only the latter is commanded to operate the swingarm.

Each motor stage *mot* has an extensive set of over 50 keywords. The most important keywords for controlling these stages include (a) ***mot*INI** and ***mot*CAL**, used for stage initialization and homing the stage; (b) ***mot*RAW**, used for reporting and commanding the stage position in "raw" encoder counts; (c) ***mot*ORD** and ***mot*NAM**, used to report and command the stage position by ordinal number or name; (d) ***mot*ERR**, ***mot*ERM** and ***mot*MSG**, used to report the stage's error codes and messages; and (e) ***mot*XMV**, which reports the reasons the stage is refusing to move.

The full set of keywords is:

***mot*AST**. Amplifier status bits that affect this axis, as documented under the `TA` command in the Galil DMC-40x0 manual. The Galil DMC-40x0 can report over-temperature, under-voltage, over-voltage, over-current, and whether there is an active "electronic lockout," all on a per-amplifier (not per-axis) basis. Per axis, the DMC-40x0 can report if there is a Hall sensor error, and whether it hit its peak current.

***mot*CAL**. Calibration state. Setting `CAL=reset` forces the stage to be unhomed. Setting `CAL=homed` causes a home sequence if the stage is unhomed.

* N.B. The *k1dm3* service has an additional axis named `ROTB`. This is *only* used to provide an interface to the rotation drive's load encoder and index marks, which are, oddly, wired to the Galil's B axis auxiliary encoder input, even though the rotation drive itself is otherwise wired to the Galil's A axis. Most `ROTB` keywords other than `ROTBRAW` and those that report the latching of the index marks are meaningless. Furthermore, *galildisp* automatically copies the axis B load encoder position into the corresponding axis A position, so for all normal work you can entirely ignore all `ROTB` keywords.

mot`CED`.
The `ROTATCED` keyword gives the difference between the actual rotation motor encoder value and the expected motor value that is computed from the load encoder value:

$$\texttt{ROTATCED} = \texttt{ROTATENC} - \texttt{ROTATENE}$$

The swingarm actuator controls (`ARM_A_` and `ARM_B_`) only use the motor encoders — the Renishaw absolute encoders are only read by the safety system — and so there are no `CED` keywords for the swingarm.

mot`CMT`. An arbitrary comment, for use by clients of the service to exchange or record info. See also *mot*`TAG`.

mot`CON` `Manual` means a manual request is executing; otherwise the value is `Auto`.

mot`DCC` DisConnected Cables message. Blank if all cables are connected. Contains the string "`motion disabled`" if the disconnected cable is required for stage motion.

mot`ENC`
`ROTATENC`: For the rotation drive, this is the motor encoder value.

`ARM_`*x*`_ENC`: The swingarm actuators only use the motor encoder for control, and so the motor encoder is used for both the `ENC` and `RAW` values (the swingarm's Renishaw absolute encoders are read and monitored by the safety system, but are not directly connected to the swingarm Galil controller).

mot`ENE`
`ROTATENE`: For the rotation drive, this is the expected `ENC` value, as derived from the `RAW` value, allowing easy comparison of the `RAW` and `ENC` values. When written, tells stage to move to the `RAW` position such that `ENE` will have this value.

There are no `ARM_`*x*`_ENE` keywords, because the swingarm actuators only use the motor encoder for control, and copy its value to both the `ARM_`*x*`_RAW` and `ARM_`*x*`_ENC` keywords.

mot`ENG` Enable engineering modes. This is a bitmask keyword containing engineering flags. Whenever this keyword is written, the *mot*`ENT` (engineering timer) keyword is re-initialized to 1200 s, and the the engineering flags remain active for the next *mot*`ENT` seconds. The flags are as follows:

`COMPINI` Allows the stage to do its special initialization (see keyword *mot*`INI`), regardless of the setting of the parent assembly's *asy*`ASM` keyword. (Related: see the parent assembly's *asy*`ENG.XASM` flag.)

`XHOME` Allows the stage to move when unhomed, and suppresses any configuration rules that would trigger automatic homing.

`XLIMITS` Allows motion if software limits exceeded. This tells *galildisp* to ignore the configured soft limits for the axis, and to clear those limits from the Galil DMC-40x0 when it next commands the stage.

`XFORBIDDEN` Allows motion into a "forbidden" range. For K1DM3, the only forbidden range is a small region near the gap in the Renishaw encoder tape. Without this flag, you aren't allowed to set a target position that is within the forbidden range.

`ZPX` Enables use of the `ZPX` keyword. See that keyword for details.

`XSAFETY` When the `XSAFETY` flag is set, various safe-to-move checks are ignored, as detailed in section 6.4. It should go without saying that this must be used with great caution.

mot`ENT` Time before engineering mode self-clears. Can be written at any time to extend the life of the ENG mode bits, up to the maximum allowed value of 1200 s.

mot`ERM` Error Message. ERM is tied closely to ERR: it is intended to always have the error message associated with the error in ERR. Unlike MSG, the ERM keyword is not intended to receive other, non-error messages.

mot`ERR` Last error number – it is the last KTL status value. Usually, closely related to `ERM` and `MSG`.

mot`GSC` Stage stopcode, as reported by Galil's `SC` command. This is an enum keyword, and its binary value is the same as the Galil stopcode value. Note that several of Galil's so-called stopcodes actually indicate motion; see the related `GSR` keyword.

mot`GSR` Indicates whether the stopcode (`GSC` keyword) is a 'running' or 'stopped' indicator. (For example, code 0 means motors are running independently, and code 60 means an "electronic cam" sequence is running.)

mot`GSS` The `GSS` keyword provides supplementary per-axis status info gleaned from periodic polling of the Galil, and that does not come through the Galil's "data records." It is in a terse compact form, as it is generally meant for use by the dispatcher rather than a user, but is nonetheless available in keyword form so that the data can be used for engineering purposes during system development.

The keyword value is a string of alternating Galil command names and their values. For example, the value for axis *D* might show:

```
BR 0 OE 1 BC 2 BX 3 BI -1 BA 1 ER 500
```

This means that the Galil variable `_BXD` = 3, `_ERD` = 500, and so on.

mot`GST` This is a bitmask keyword showing the stage's axis status bits, pretty much exactly as reported by the Galil in the data record. This is an engineering keyword that is not used in normal operations.

mot`GSW` This is a bitmask keyword showing the stage's axis switch bits, equivalent to the value reported by Galil's `TS` command. This is an engineering keyword that is not used in normal operations.

mot`GTE` Current motor positioning error, equivalent to the value reported by the Galil's `TE` command.

mot`LCK` Software lock switch. If set to "`unlocked`", motion keywords can be modified; otherwise, they cannot. As a convenience, setting it to the empty string "" is shorthand for "`unlocked`".

mot`LIM` Stage limit status. A bitmask keyword that indicates if soft limits, primary limits, and/or secondary limits are active. (The "primary" limits give the state of the Galil's limit inputs for the axis. The "secondary" limits, if provided, are additional signals to indicate that the stage overshot its normal limit and had to be stopped by some

independent method such as directly motor power, bypassing normal Galil motor commands.)

`ROTATLIM` The rotation stage has no limits, so this keyword has no particular value.

`ARM_`*x*`_LIM` Each swingarm actuator has limit switches, which are wired into the Galil in the standard way. However, they should never trigger, because the swingarm's independent safety system (see chapter 8) will always trip before the limit switches trip.

mot`MAP` Computes the mapping of a keyword value from one suffix to another. The use is to write the keyword with a value of the form

fromval fromsuffix tosuffix

The next 'show' command will return the mapping.

Example: `modify -s k1dm3 ROTATMAP='123 RAW VAL'`
Result: `RAW=123 -> VAL=0.615`

Example: `modify -s k1dm3 ROTATMAP='LNas NAM RAW'`
Result: `NAM=LNas -> ORD=1 -> RAW=682235`

mot`MOD` Sets the axis control mode: *galildisp* is modal, with the following permitted values:

Halt: Motion is not allowed; stops stage if it is already moving.

Pos: Stage may be commanded to a specific position.

Jog: Stage may be commanded to a specific speed.

Track: smoothly track a stream of PT or PVT demands. This mode is not used with K1DM3.

mot`MOE` When read, reports whether the motor for this stage is left on (and servoing) or off after a move.

When written, the value can be `Off`, `On`, or `Default` (use the configured-in default value).

For all K1DM3 stages, the configured default value is `Off`.

mot`MOO` When read, reports whether the motor for this stage is currently on or off.

The `MOO` keyword can be written only in `Pos` mode. Setting the value to `off` will turn off the motor; setting the value to `on` will turn on the motor and begin servo'ing to hold position.

A value of `off` can be written at any time, and will have different effects depending on the state of the stage, but will always result in the motor being turned off as soon as possible.

Writing the `MOO` keyword does not change whether the motor is turned off at the end of future moves — *mot*`MOE` will determine whether the motor is servo'ing or off at the end of the next move.

mot`MSG` The `MSG` usually holds the error message associated with the `ERR` error code (as does the `ERM` keyword), but it can also hold non-error messages from the dispatcher, such as status messages while executing a sequence.

mot`MSP` By default, a stage does ordinary moves and homing at its configured-in maximum speed (see the `SPD` keyword). The `MSP` keyword may be set to any value at or below that maximum, and until it is reset, moves and homing will be limited to `MSP`.

Example: the `SWINGARM` assembly's sequencer uses `MSP` to change the actuators to a slow speed when entering the V-blocks or dock.

mot`NAM` When at a named position, this keyword gives the name, or `Unknown` otherwise. The corresponding ordinal position is given by the `ORD` keyword, or −999 when not at an ordinal position. The mapping between the raw, ordinal, and named positions is given by the `RON` keyword.

See appendix A for tables of the drum and swingarm positions.

mot`NPX` The `NPX` (Named PrefiX) keyword is used for selecting subsets of named keywords. It is not used by the *k1dm3* dispatchers.

A client application can set the `NPX` value as a way of saying, "this subset of positions should be displayed to users;" or the dispatcher can simplify the representation of removable components by directly setting the `NPX` value to indicate which elements are currently installed.

When translating from `RAW` to `ORD` or `NAM`, the dispatcher ignores any entries for which the name doesn't begin with the `NPX` value. When translating a request to move to a new named position, if the requested name doesn't begin with the `NPX` value, it is automatically prefixed to the requested name.

mot`ORD` When at a named position, this keyword gives the ordinal value of that position, or −999 otherwise.

See the `NAM` keyword, above, for K1DM3's named and ordinal positions.

mot`PSE` When written with value n, the modify command waits (pauses) for n milliseconds, then returns. The dispatcher itself doesn't pause; it's provided to allow scripts to pause in a way that can be interrupted just like any modify command.

mot`RAW` Position as measured by the stage's more accurate encoder (cf. the `ENC` keyword, which is the stage's secondary, less accurate encoder).

For the `ROTAT` stage, the `RAW` value is the Renishaw load encoder, and the `ENC` value is the motor encoder.

The `ARM_`*x*`_` actuators are controlled only using the motor encoder (the absolute encoder signal is read by the safety system, and is not connected to the swingarm DMC-40x0), and the motor encoder is used for both the `RAW` and `ENC` values.

mot`RCD` (Remote Control Disablers) If remote (keyword) control is disabled, this explains why. The keyword is blank if remote control is allowed.

mot`REL` Relative move of RAW keyword. On read: return integrated motion since last zeroed. Zeroed when: new pos cmd given (POS mode); or transition from TSP=0 (JOG mode); or any track update (TRACK mode).

mot`RON` RAW→ORD→NAM keyword mapping. This is a list of alternating RAW, ORD, and NAM values; it may be updated on the fly to reflect physical changes in a stage,

such as updates to filters in a filter wheel. The first character of the keyword is the separator between fields.

Whenever a client writes a new `RON` value, the dispatcher permanently caches it in `$RELDIR/var/`*servicename*`/mapRON.cfg`, and the previous file is copied to `$RELDIR/var/`*servicename*`/Old/mapRON.cfg.`*timestamp*.

N.B. Whenever the dispatcher is restarted, the contents of the `mapRON.cfg` file are used instead of the more-or-less static configuration files. To avoid future confusion, it is *strongly* recommended that, after the new `RON` data are tested and confirmed, the static configuration files be updated to match the contents of the `mapRON.cfg` file.

When a client reads a `RON` keyword, or when the dispatcher broadcasts one, the fields are separated with tabs, e.g.:

Tab 1234 *Tab* 1 *Tab* Name1 *Tab* 4567 *Tab* 2 *Tab* Name2

When writing the keyword to update the current `RON` mapping, any character can be used that's not used in any field data. For example, the above `RON` string could be sent to the dispatcher using:

```
modify -s servicename FOORON=,1234,1,Name1,4567,2,Name2
or
modify -s servicename FOORON=X1234X1XName1X4567X2XName2
```

mot`SPD` Stage velocity, in `RAW` units/s.

For the `ROTAT` stage, the `SPD` value is Renishaw load encoder counts/s.

For `ARM_`*x*`_` actuators, the `SPD` values are motor encoder counts/s.

mot`STA` Control status. This is an `enum` keyword; the possible values are:

Locked Software lockout.

Calibrating

Not Calibrated Stage needs to calibrate before other motion is allowed.

Halted Stage is stopped and in the `Halt` mode.

Ready Stage is ready to be commanded.

Stopping

Moving Stage is doing a `Pos`-mode move.

Jogging Stage is moving in `Jog`-mode.

Slewing Used in `Track` mode; not used with K1DM3.

Acquiring Used in `Track` mode; not used with K1DM3.

Tracking Used in `Track` mode; not used with K1DM3.

TrackSusp Used in `Track` mode; not used with K1DM3.

At Max Used in `Track` mode; not used with K1DM3.

At Min Used in `Track` mode; not used with K1DM3.

Local/Stopped Local (manual) mode, and stopped.

Local/Jogging Local (manual) mode, and executing a jog command.

Local/Moving Local (manual) mode, and moving to a specific position.

Fault

Disabled

Initing Stage is executing its special init command.

mot`STP` Request stop. This is a string keyword. When written with any value, the stage is commanded to stop. Unlike setting the stage's control mode to `Halt`, the `STP` keyword leaves the mode unchanged.

When read: returns the last value written; useful if clients put an explanatory message into the keyword when commanding the stop.

mot`SWT` (Stopped and Within Tolerance) The keyword value is `inPosition` if stage completed a move and is within the configured positioning tolerance.

mot`TAG` An arbitrary comment string, for use by clients of the service to exchange or record information. See also *mot*`CMT`.

mot`TNM` (Target NaM) This is the `NAM` value that corresponds to the current target ordinal value (see *mot*`TRD`).

mot`TOR` Motor torque. The binary value is the output voltage to the current amplifier for the servomotor.

For both the `ROTAT` and `ARM_`*x*`_` stages, the ascii units are amperes of output current to the motor. Note: the dispatcher doesn't update the `TOR` keyword until the torque changes by the configuration-specified tolerance level.

mot`TOS` Same as TOR while the Galil stopcode (see *mot*`GSC`) is not a 'running' value; not updated while stopcode has a 'running' value. This is occasionally useful to keep track of the stationary holding torque, especially in situations where the stage is not stationary for extended periods of time.

mot`TRD` (TaRget orD) Ordinal position of the current target position, *mot*`TRG`.

mot`TRG` The stage's `RAW` position target.

mot`TRL` The target position, as a relative move (i.e., distance from current position). It is 0 when not in `Pos` mode.

mot`TRN` For rotary stages, this gives the number of full turns, computed as raw counts divided by the counts per rotation (hence, this can be negative). It is undefined for linear stages. It is writable, so that applications can set the count to zero or another desired value.

mot`TSP` Target speed, in `SPD` units. This indicates the speed at which the controller is attempting to drive the stage, while `SPD` is a measure of the instantaneous velocity of the load encoder.

mot`TVA` The target position (*mot*`TRG`), in `VAL` units.

mot`TVX` The target position (*mot*`TRG`), in `VAX` units.

mot VAL The VAL keyword contains the RAW value after conversion to physically interesting units.

For the ROTAT stage, the VAL units are mm of motion at the drum's Renishaw tape.

For the ARM_*x*_ stages, the VAL units are mm of actuator extension.

mot VAX The VAX keyword contains the RAW value after conversion to "externally" interesting units.

For the ROTAT stage, the VAX units are degrees of rotation.

For the ARM_*x*_ stages, the VAX units are degrees of swingarm rotation, where:

0 degrees is in the plane perpendicular to the light path;

45 degrees is the clamped-in-V-block position;

90 degrees is the mirror parallel to the light path (or, mirror is at the horizontal position when on the handling cart); and

104.5 degrees is the fully-retracted, Cassegrain position.

mot VEL Velocity, in VAL units/s, as derived from the SPD value.

Hence, units are mm/s for both the drum rotation and swingarm actuators.

mot VEX Velocity, in VAX units/s, as derived from the SPD value.

Hence, units are deg/s for both the drum rotation and swingarm actuators.

mot XMV Shows all the reasons that this stage (probably) cannot move. Some reasons are not necessarily disabling — for example, a stage may be in a limit that prevents motion in one direction only.

mot ZPX Setting *mot* ZPX=*n* says to force the raw value *mot* RAW=*n*, and set *mot* CAL=homed. This bypasses normal homing, and is only enabled when the ZPX bit is set in *mot* ENG.

The K1DM3 drum is not allowed to rotate when the swingarm is retracted, because the drum will be grossly out of balance. This can create a deadlock situation, since the swingarm can't be moved unless the drum position is known, and normal homing of the drum requires rotation. To avoid deadlock, the drum has two special limit switches that enable homing without rotation. One switch triggers when the drum is at the MirrorUp position, and the other triggers at the Cass/Stow (mirror facing down) position, which are the only positions at which the swingarm is allowed to be retracted. If the detent is engaged and either one of these switches is active, then the drum's position is known precisely, and the ROTAT stage can be homed by using the ZPX capability.

The swingarm actuators are always homed using the ZPX capability. The absolute position, converted to motor encoder units, is provided by the safety system through the *k1dm3saf* keywords ARM_*x*_ENE. These keyword values are applied to the swingarm actuators using the *k1dm3* ARM_*x*_ZPX keywords.

12.5 Assembly (Combo-Axis) Keywords

Assemblies come in two flavors: motor-like and digi-like. A motor-like assembly provides a subset of standard motor-stage keywords (section 12.4) for monitor and control of the assembly, plus a handful of keywords for assembly management. Similarly, a digi-like assembly provides a subset of standard digi-stage keywords (section 12.3), plus a few keywords for assembly management.

The `DRUM` and `SWINGARM` assemblies are motor-like assemblies, while the `CLAMPS` and `M3MAN` assemblies are digi-like.

12.5.1 Motor-like Assemblies

Assembly Prefixes: `DRUM`, `SWINGARM`

The keywords for a motor-like assembly *asy* are:

***asy*`ASM` The `ASM` keyword controls whether the assembly's components may be directly commanded via keywords, or whether they may only be commanded from the assembly itself. If written with `linked`, components may only move as part of this assembly. If written with `unlinked`, all components may be individually commanded. Linking and unlinking can optionally also have side effects, implemented by per-assembly custom code.

The `DRUM` module doesn't normally operate with `ASM=linked`. It is OK to command components individually, because they will refuse to move if it is not safe to proceed.

On the other hand, the `SWINGARM` always operates with `ASM=linked`, because problems will quickly arise if the two swingarm actuators are not moving in synchrony. The custom code for the `SWINGARMASM` keyword does the following:

`linked` When set to linked, the `SWINGARM` assembly slaves the `ARM_B_` actuator to the `ARM_A_` actuator by putting the Galil axes into "gantry" mode (see Galil commands `GR` and `GM` in the DMC-40x0 manual). The `SWINGARM` assembly's rules for motion always begin by setting its own `ASM` keyword to `linked`, thus ensuring gantry mode.

`unlinked` When set to unlinked, the `SWINGARM` sends commands to the Galil to *turn off* gantry mode. This would only make sense for certain low-level engineering work, and should never be used for ordinary control of the swingarm.

***asy*`CAL` Same use as a motor stage `CAL`. When written, applies `CAL` to all components. When read, returns a combined `CAL` calibrate state for all component axes.

***asy*`CMP` Space-separated list of components that make up this combo stage.

***asy*`CMT` Same use as a motor stage `CMT`.

***asy*`CON` Same use as a motor stage `CON`.

***asy*`ENG` Enable engineering modes. Whenever this keyword is written, the engineering bits remain active for the next ***asy*`ENT` seconds. The bits are as follows:

`COMPINI` Same as for motor stage.

`XHOME` Same as for motor stage.

`XLIMITS` Same as for motor stage.

`XASM` = allow commands to be issued to child components regardless of *asy* `ASM` setting. This is different than setting *asy* `ASM=unlinked`, because the latter has side effects such as unslaving stages. This keeps all stage configuration in the ASM setting.

`XFORBIDDEN` Same as for motor stage.

`ZPX` Same as for motor stage.

`XSAFETY` Same as for motor stage.

asy `ENT` Same use as a motor stage `ENT`.

asy `ERM` Same use as a motor stage `ERM`.

asy `INI` Same use as a motor stage `INI`. When written, applies `INI` to all components. When read, returns a combined `INI` calibrate state for all component axes.

asy `LCK` Same use as a motor stage `LCK`. Recursively applies to components.

asy `MOD` Conceptually, has the same function as a motor stage `MOD`. However, the only values are `Halt` and `Pos`.

asy `MOE` Same use as a motor stage `MOE`. When read, can have the additional value `Mixed`, meaning some components are `On` and others are `Off`.

asy `MOO` Same use as a motor stage `MOO`. When read, can have the additional value `Mixed`, meaning some components are `On` and others are `Off`.

asy `NAM` Name of the current position, or `Unknown` if not at a named position.

asy `NMS` All valid `NAM`'s, as a tab-separated list. It may be updated on the fly to reflect physical changes in a stage, such as updates to filters in a filter wheel. The first character of the keyword is the separator between the values.

For `DRUM`: the valid names are the same as for `ROTATNAM`.

For `SWINGARM`: the valid names are:

`Retract` Swingarm is retracted, and docking pin is engaged.

`Retracted,Unpinned` Swingarm is retracted, but docking pin is not engaged.

`Mirror90` Mirror is at the 90° position, which is face-up and horizontal when on the handling cart. Should be used when removing mirror for recoating.

`Deploy` Swingarm is in the V-blocks and the clamps are closed.

`Deployed,Unclamped` Swingarm is in the V-blocks but the clamps are not closed.

`NotFullyDeployedButClamped` Swingarm isn't quite within tolerance for the V-block position, but the clamps are closed.

`Unknown` None of the above.

asy `PSE` Same use as a motor stage `PSE`.

***asy*RAW** Same use as a motor stage RAW. Its exact definition is set by configuration-file rules.

For DRUMRAW: the value is copied from ROTATRAW.

For SWINGARMRAW: the value is copied from ARM_A_RAW.

***asy*REL** Same use as a motor stage REL.

***asy*STA** Same use as a motor stage STA, but doesn't include the track-mode status values.

***asy*STP** Same use as a motor stage STP.

***asy*SWT** Same use as a motor stage SWT.

***asy*TAG** Same use as a motor stage TAG.

***asy*TNM** Similar to a motor stage TNM value, it's the NAM target. It is blank if the most recent move command was not by setting NAM.

***asy*TRG** Similar to a motor stage TRG value, it's the RAW position target. It is blank if the most recent move command was not by setting RAW.

***asy*TRL** Same use as a motor stage TRL.

***asy*VAL** Same use as a motor stage VAL.

For DRUMVAL: the value is copied from ROTATVAL.

For SWINGARMVAL: the value is copied from ARM_A_VAL.

***asy*VAX** Same use as a motor stage VAX.

For DRUMVAX: the value is copied from ROTATVAX.

For SWINGARMVAX: the value is copied from ARM_A_VAX.

***asy*VX2** Same use as a motor stage VX2.

For DRUMVX2: not used.

For SWINGARMVX2: the value is copied from ARM_A_VX2.

***asy*XMV** Shows all the reasons that this assembly (probably) cannot move. Some reasons are not necessarily disabling — for example, a component stage may be in a limit that prevents motion in one direction only. The assembly XMV is the union of all its components' XMV values, excluding those that are configured as remediable through normal component sequencing. For example, the swingarm actuator XMV keywords will show that the actuators can't move if the clamps are closed, but the SWINGARMXMV keyword won't show this, because its sequencer will open the clamps before attempting to move the actuators.

12.5.2 Digi-like Assemblies

Assembly Prefixes: CLAMPS, M3MAN

The keywords for a digi-like assembly *dig* are:

***dig*ASM** The ASM keyword controls whether the assembly's components may be directly commanded via keywords, or whether they may only be commanded from the assembly itself. If written with **linked**, components may only move as part of this assembly. If written with **unlinked**, all components may be individually commanded. Linking and unlinking can optionally also have side effects, implemented by per-assembly custom code.

The K1DM3 digi-like assemblies do not normally operate with **ASM=linked**. It is OK to command components individually; they will refuse to move if it is not safe to proceed.

***dig*CMP** Space-separated list of components that make up this combo stage

***dig*CMT** Same use as a digi-stage CMT.

***dig*CON** Same use as a digi-stage CON.

***dig*ENG** Engineering flags. Whenever this keyword is written, the engineering bits remain active for the next ***dig*ENT** seconds. The bits are as follows:

COMPINI Same as for digi-stage.

XHOME Same as for digi-stage.

XLIMITS Same as for digi-stage.

XASM = allow commands to be issued to child components regardless of ***asy*ASM** setting. This is different than setting ***asy*ASM=unlinked**, because the latter has side effects such as unslaving stages. This keeps all stage configuration in the ASM setting.

XFORBIDDEN Same as for digi-stage.

ZPX Same as for digi-stage.

XSAFETY Same as for digi-stage.

***dig*ENT** Same use as a digi-stage ENT.

***dig*ERM** Same use as a digi-stage ERM.

***dig*ERR** Same use as a digi-stage ERR.

***dig*LCK** Same use as a digi-stage LCK. Recursively applies to components.

***dig*MSG** Same use as a digi-stage MSG.

***dig*POS** Name of the current ordinal position, or **Unknown** if not at an ordinal position.

For CLAMPS, the values are CLOSED and OPEN.

For M3MAN, the values are identical to the M3AGENT values given in section 4.2.

***dig*PSE** Same use as a digi-stage PSE.

***dig*STA** Similar to a digi-stage STA, but it also has calibrating-type status values to handle the case when a motor component is part of a digi-type stage.

***dig*STP** Same use as a digi-stage STP.

***dig*`TAG` Same use as a digi-stage `TAG`.**

***dig*`TRG` Same use as a digi-stage `TRG`.**

***dig*`XMV` Shows all the reasons that this assembly (probably) cannot move. Some reasons are not necessarily disabling — for example, a component stage may be in a limit that prevents motion in one direction only. The ***dig*`XMV` is the union of all its components' XMV values, excluding those that are configured as remediable through normal component sequencing.

12.6 Status from Safety System

As described in chapter 8, the safety system reports its status using six output bits that are connected to inputs of the swingarm control Galil. This allows the *k1dm3* service to directly know the cause of any safety triggers, without needing to consult the separate, supplementary safety system service *k1dm3saf*.

The safety input states are available via these *k1dm3* keywords:

`SAF_EAG` `Yes` if the two encoders agree with each other; otherwise `No`

`SAF_EIR` `Yes` if the encoders are within the valid range; otherwise `No`

`SAF_EOK` `Yes` if the two encoders are OK (have no error signal); otherwise `No`

`SAF_ESP` `Yes` if the encoder speed is within the speed limit; otherwise `No`

`SAF_HBT` `Yes` if the heartbeat signal is pulsing; `No` otherwise. The keyword shows if the heartbeat has timed out, not the actual pulses of the heartbeat signal.

`SAF_RER` `Yes` if the absolute encoders show the swingarm is in the slow zone near the fully-retracted point (RER = REtract Region); otherwise `No`.

`SAF_VBR` `Yes` if the absolute encoders show the swingarm is in the slow zone near the V-blocks (VBR = V-Block Region); otherwise `No`.

`SAFETYBITS1` A mask keyword that shows the combined state of all the alarm bits. It will either be `AllOK` or one or more of `EncNotOK`, `EncDisagree`, `EncOutOfRange`, `SpeedNotOK`, and `HeartbeatFail`.

`SAFETYBITS2` A mask keyword that shows the combined state of the position bits. It should be just one of `FastRegion`, `VblockRegion`, or `RetractRegion`.

12.7 TCS Interface Keywords

When K1DM3 is on a handling cart, it ignores the telescope control service *dcs1*. But when K1DM3 is installed in the tertiary tower, it monitors the *dcs1* tertiary mirror keywords, `TERT`***xxx***, and moves as demanded by the usual `Halt`, `Init`, `Standby`, and `Move` rules.

The read-only keyword `USEDCS` indicates if *k1dm3* is using the *dcs1* service. When `USEDCS` is true, K1DM3 "mirrors" the following *dcs1* keywords. For each of the following keywords, there is an equivalent *k1dm3* keyword of the same name, and K1DM3 copies the *dcs1* value into the *k1dm3* value:

- ROTTIME. Serves as a "heartbeat" keyword for the TCS rotator control daemon.
- EL Telescope elevation.
- TERTDEST. Tertiary destination, an enum.
- TERTPOSN. Current tertiary position, an enum.
- TERTERRS. Error message; also used for general-purpose messages.
- TERTERVL. Integer error code.
- TERTSTAT. Tertiary status, an enum.
- TERTSTST. Tertiary status, as a string.
- TERTABRT, TERTABRTC. Abort.
- TERTHALT, TERTHALTC. Halt.
- TERTINIT, TERTINITC. Init.
- TERTMOVE, TERTMOVEC. Move.
- TERTSTBY, TERTSTBYC. Standby.

When any of the *k1dm3* keywords TERT{ABRT,HALT,INIT,STBY,MOVE}C transitions from false to true (or no to yes), the corresponding "modify" sequencer is executed (see fig 4.1; sections 4.2 and 3.6). This enables one to nearly exactly duplicate the actions triggered by the TCS, simply by toggling the value in the appropriate *k1dm3* keyword.

When USEDCS is true, the sequencers send their status information to the *dcs1* TERT*xxx* keywords, which are then mirrored into the corresponding *k1dm3* keywords. On the other hand, when USEDCS is false, the sequencers do not write to the *dcs1* keywords, but instead write directly to the *k1dm3* keywords.

12.8 Major Sequencer Keywords

The two major sequencer keywords, ACTIVATE and M3AGENT, each have a group of supplementary keywords to provide feedback as they execute. The supplementary keywords are named are named ACTIV*xxx* and M3*xxx*, respectively.

The sequencers are described in detail elsewhere (M3AGENT in section 4.2, and ACTIVATE in section 3.5); their values in brief are:

ACTIVATE:

- Ready for observing: Observe
- Activate swingarm and initialize actuators: Swingarm
- Activate swingarm but don't initialize actuators: Swingarm/noinit
- Ready to rotate: Rotate
- Ready for transport in/out of tower: Transport

M3AGENT:

- TCS-like actions: Halt, Init, Stby, Stop
- Instrument positions: LNas, RNas, LBC1, RBC1, LBC2, RBC2, Cass
- Supplementary positions: Stowed, Mirror90, MirrorUp
- Other motions: StepPos, StepNeg, JogPos, JogNeg
- Special actions: NoMoveInitRotate, NoMoveInitEncoders, EngrTest

The supplementary keywords are:

M3ENG, M3ENT. The M3ENG keyword provides engineering flags, and the M3ENT keyword provides a corresponding timer. When you set flags in M3ENG, the timer is set to 20 minutes. The engineering flags remain active until the timer expires. The flags are:

- USEDCS. Normally, if LOCATION is not Tower, then k1dm3 does not respond to TCS tertiary commands, nor does it report status in *dcs1* TERT*xxx* keywords. Setting the USEDCS engineering flag tells k1dm3 to respond to TCS tertiary commands just as if it is in the tower. This can be used for limited on-deck testing of TCS-related functionality.
- Engr1, Engr2, Engr3. If one of these is set, and M3AGENT is set to EngrTest, then the sequencer invokes the corresponding test script *m3.engrTest1*, *m3.engrTest2*, or *m3.engrTest3*.

ACTIVERM, M3ERM. These show the last error message for the executing sequence.

ACTIVLOG, M3LOG. At the end of each ACTIVATE or M3AGENT execution, the complete stream of its *xxx*MSG output is copied into the *xxx*LOG keyword, thus giving a one-keyword account of the full execution of the sequence.

ACTIVMSG, M3MSG. Each ordinary line printed by an ACTIVATE or M3AGENT sequence is copied into this keyword, so that it provides a running account of the executing sequence. In addition, the dispatcher puts some status information into the keyword, such as noting the start and stop of the sequence.

ACTIVRUN, M3RUN. The boolean *xxx*RUN keywords are set to true just before exec'ing the sequencer's external script, and set to false when the script exits.

For the convenience of the engineering GUI, the boolean keywords ACTIVATE_*xxx*, where *xxx* is one of {ARM, DRUM, OBS, TRANS}, are true if ACTIVATE has the corresponding value, and can be written to command the corresponding state.

12.9 Elevation Keywords

EL_DEPLOY The telescope elevation, in degrees, at which the swingarm should be deployed. Note: during testing in December 2017, the mirror positioning repeatability was characterized at El=67° only. Repeatability at other elevations is not known, nor whether the mirror deploys into a slightly different position at other elevations.

EL_DMD An extremely terse string keyword that summarizes the combined EL_NEED and EL_SATISFIES values:

- '' An empty string means no elevation change is required.
- *xx* The current command requires the telescope to slew in elevation to *xx* deg.
- 45-*xx* The current command requires the telescope to slew to any elevation between 45° and *xx* deg.

EL_DMDX An very terse string keyword that summarizes why EL_DMD is non-blank. Its values can be:

- EL != *xx* Elevation isn't *xx* deg.

- `EL <` *xx* Elevation is less than *xx* deg.
- `EL >` *xx* Elevation is greater than *xx* deg.
- '' An empty string means that `EL_NEED` isn't `Deploy` or `Retract`, or else `EL_SATISFIES` meets `EL_NEED`.

`EL_MAXWAIT` Maximum amount of time, in seconds, the sequencer will wait for the telescope to reach the elevation specified by `EL_DMD`, before it gives up and throws an error. Valid range: 0–180.

`EL_NEED` An enumerated keyword, indicating the telescope elevation required by the currently-executing Init or Move:

- `Any` = any elevation is ok.
- `Cart` = The module is on a handling cart, so telescope elevation is irrelevant.
- `Deploy` = Elevation must be at `EL_DEPLOY`.
- `Retract` = Elevation must be between 45° and `EL_RETRACT` deg.

`EL_RETRACT` The telescope elevation, in degrees, below which the swingarm should be retracted.

`EL_SATISFIES` A mask keyword, indicating if the telescope elevation is suitable for deploy or retract operations:

- `None`: swingarm may not be moved at this elevation.
- `Cart`: The module is on a handling cart, so telescope elevation is irrelevant.
- `Deploy`: The telescope elevation is at `EL_DEPLOY`, suitable for deploying the swingarm. Generally, `EL_DEPLOY` ≤ `EL_RETRACT`, so `Retract` will generally be set whenever `Deploy` is set. Will not be set if module is on handling cart.
- `Retract`: The telescope elevation is suitable for retracting the swingarm: 45° ≤ Elevation ≤ `EL_RETRACT`. Will not be set if module is on handling cart.
- `Activate`: The telescope elevation is suitable for activating the swingarm: 45° ≤ Elevation. Will not be set if module is on handling cart.

`ELALERT` A string keyword that is blank if `ELMIN` and `ELMAX` are nominal, else a terse message indicating their problematic values. See the configuration rules for the details of when this is evaluated. Resets to blank whenever swingarm reaches retract position. If a client application sets the value to `ack`, the dispatcher will disable further updates of the keyword until it is blanked at next full retract.

`ELMIN`,`ELMAX` The minimum and maximum telescope elevation during the last deploy operation. Allows one to check if the elevation stayed at `EL_DEPLOY` during the entire motion.

12.10 Environment Keywords

`EBOX_3_3V`, `EBOX_3_3V_R`: E-box 3.3V supply voltage, and "raw" analog input voltage before scaling to correct range.

`EBOX_5V`, `EBOX_5V_R`: E-box 5V supply voltage, and "raw" analog input voltage before scaling to correct range.

`EBOX_AIR_T`, `EBOX_AIR_TR`: Air temperature in E-box sensor, deg C, and "raw" analog input voltage before conversion to deg C.

12.11 Other Keywords

`ANALOG`*d*`_`*n*, where *d* is a dispatcher number (Swingarm=1, Drum=2), and *n* is a Galil analog input number (1..8). These keywords give the "raw" analog input voltage as reported directly by the Galil, before conversion to the specific keyword value.

`AIRPRESSX` "Smart" air pressure report.

- `NoInnerAir` = air supply is not connected to the inner drum;
- `NoInnerPower` = the inner drum power is off;
- `DispNotReady` = the swingarm dispatcher is not Ready;
- otherwise = `INR_AIRPRESS`.

`ARMDIFF` Difference between Arm A and Arm B, motor encoder counts. Not to be confused with *k1dm3saf*'s `ARMDIFF` keyword, which is the difference between the absolute encoders, converted to motor encoder units.

`CABLES` This bitmask keyword shows which cables are connected. The possible cable bits are: `TwrInnerSig`, `TwrInnerPwr`, `TwrRotMot`, `TwrRotEnc`, `TwrRotSig`, `SvcInnerSig`, `SvcInnerPwr`, `SvcRotMot`, `SvcRotEnc`, `SvcRotSig`, `Paddle`. Here, `Twr` means the in-tower cables, `Svc` means the "service" cable set used when operating the module on the handling cart, `Rot` refers to rotation (drum) cables, `Inner` means inner-drum cables, `Mot` means motor-power cables, `Enc` means encoder cables, and `Sig` means other signal cables.

`CABLESET` This enum keyword is derived from `CABLES`, an shows which cableset is connected. The possible values are: `None` (No cables), `Telescope` (the in-tower cableset is fully connected), `Service` (the service cableset for on-handling-cart operations is fully connected), `Mixed` (a mixture of telescope and service cables are connected), `Incomplete` (an incomplete set of cables is connected), `Paddle` (the hand paddle and nothing else is connected).

`CLAMP_`*a*`_CLO`, where *a* is one of {`A`, `B`, `C1`, or `C2`}. This is a convenience keyword that is `Yes` or `No` according as `CLAMPSCLOSED` (see below) indicates the corresponding clamp is closed. These are used as pseudo-limit bits for the clamp digi-stages: instead of an actual clamp-is-closed limit switch, the dispatcher substitutes the value of this keyword.

`CLAMPCURR` This indicates the combined current flow through the individual closed-clamp signals. When closed, clamps {*A,B,C1,C2*} generate signals of $\{1, 2, 4, 8\} \times 660\,mV$, respectively. The analog voltages are combined to produce a single analog value that is decoded by the drum subsystem. This value should be $(1 \ldots 15) \times 660\,\mathrm{mV}$. Its value is only meaningful when the module is at one of the instrument positions (where there is analog connectivity to the inner drum), and not moving, so that the signal is stable.

`CLAMPSTEADY` True when the `CLAMPCURR` value is steady. This is used as part of the rule for determining when it's valid to copy the "raw" clamps-closed value, `CLAMPUNFILT`, to the actual `CLAMPSCLOSED` keyword.

`CLAMPUNFILT` This is the decoded value of `CLAMPCURR`, without regard to whether the signal is steady or the drum is at an instrument position. It is a mask keyword with fields `A`, `B`, `C1`, `C2`, or `None` (when no field is active).

`CLAMPSCLOSED` This is the same value as `CLAMPUNFILT`, except it is updated only when the signal is steady and the module is at an instrument position.

`DEADMAN{1,2,3}` Each Galil is running a deadman timer thread that must be reset by the dispatcher every second or so. If the timer expires, the deadman thread stops all axes, and sets an output bit. The `DEADMAN`*i* boolean keyword reports the state of the output bit.

`DOCK_HALL` The swingarm dock system has a Hall-effect sensor that was installed to support an alternative homing scheme, in case the preferred method did not work. Well, our preferred method worked fine, so the hall sensor is not needed. Nonetheless, this keyword exists and reports the state of that Hall sensor.

`DOCKBITSOUT` Tells the state of the docking pin output control bits.

`DRUMSTA_STOP` Boolean convenience keyword that reports whether the `DRUM` assembly status is a "stopped" status.

`ESTOP` The current E-stop state.

`GETAIR` An air supply sequencer. It engages the air supply if not engaged, then waits for pressure to exceed 75 psi. Values:

- `Unknown` = we can't read `INR_AIRPRESS`.
- `NotReady` = `INR_AIRPRESS` is low.
- `Ready` = `INR_AIRPRESS` $\geq$ 75 psi.

`INBLOCK11` Value of Swingarm dispatcher (dispatcher 1) input bit block 1 (I/O bits are reported by the Galil in blocks of 8, beginning at 0).

`INNER24V` Turn on/off 24V power to inner drum.

`INNER48V` Turn on/off 48V power for swingarm motors.

`OUTER48V` Turn on/off 48V power for rotation motor.

`INR_AIRPRESS` Air pressure, as read at the inner drum.

`LK_`*xxx* Drum dispatcher's mirror of last known (LK) values for a variety of swingarm dispatcher keywords. This provides a readable, useful value even when the swingarm dispatcher is suspended, for keywords `ARM_`*i*`_{GSS,INI}`, and `SWINGARM{ASM,MOD,NAM,TNM}`.

`LOCATION` K1DM3 physical location: `Tower` if the in-tower signal is active; `Cart` if the pinned-to-cart signal is active; else `none`. This is a mask keyword, and it's an error if neither or both location signals are active.

`MAN_OVERRIDE` Normally, keyword commands are rejected while the hand paddle is connected. When `MAN_OVERRIDE` is set to n, then keyword commands are accepted for the next n seconds. *Warning:* it is unsafe to enable keyword commands when others believe they are locked out! This should only be used with care.

`OVERSPEED` This boolean keyword goes true whenever `k1dm3saf.CT_SPEED` increments, which indicates that an overspeed condition occurred. It is reset to false when the motors are next turned on.

`OVERTEMP` A mask keyword that indicates if any swingarm overtemp signals are active

`PADDLE` This is a mask keyword that shows which hand-paddle bits are active.

`PENDINGSTOP` Monitors the total number of executing or pending `DRUMSTP` and `SWINGARMSTP` commands.

`PINNED` A mask keyword that reports which pins are connected: `None`, `Cart` (drum pinned to cart), `Drums` (drums pinned together).

`ROTATSW` The drum has two supplementary position switches that indicate if the drum is near the MirrorUp or Cass/Stow positions. This mask keyword indicates which of these is active, if either.

`ROTSTA_STOP` Boolean convenience keyword that reports whether the `ROTAT` assembly status is a "stopped" status.

`ROTATSW` State of position switches on drum. The drum has two coarse position switches — one trips at the `MirrorUp` orientation; the other at the `Cass/Stow` orientation. If one of these switches is active, and the drum's detent mechanism is engaged, then the dispatcher can assert the drum's precise position without needing to do any rotation to read the Renishaw tape's reference marks. This is crucial for bootstrapping the system if it powers up with the swingarm in an elevated position, since rotation is forbidden in that case.

`SWINGSTA_STOP` Boolean convenience keyword that reports whether the `SWING` assembly status is a "stopped" status.

`WITHDOCK` Indicates if the swingarm is configured to control the dock.

Chapter 13

k1dm3saf KTL Service Keywords

This section summarizes all the keywords provided by *k1dm3saf*. Additional information — such as data type, data range, allowed values — is available using *gshow* to query the keyword data, using its various help options.

The *k1dm3saf* service has a single dispatcher, which listens for status messages sent to UDP port 1222 by the Swingarm Safety System's Galil RIO. To minimize the possibility of interfering with the safety system's operations, the dispatcher never sends anything to the RIO.

The *k1dm3saf* keywords are:

Dispatcher Self-State

`DISPOCLK` Dispatcher clock; increments every second.

`DISPODBG` Not currently used by dispatcher.

`DISPODWIM` Debugging interface. See code for details.

`DISPOERR` Last dispatcher error number. (IRL, this keyword never changes from 0).

`DISPOHOM` Hostname on which the dispatcher is running.

`DISPOMSG` Last general status message.

`DISPOPID` Process ID of the dispatcher.

`DISPOREQ` Last "modify" request received by the dispatcher.

`DISPOSHUTDN` When written with any value, tells dispatcher to exit.

`DISPOSTA` Either `Initializing` (brief startup time value, generally too brief to be logged), or `Ready`.

Safety Status

`ALL_OK` Overall state of safety monitor: `Unknown`, `No`, or `Yes`.

`CT_DISAGREE` Count of occurrences of the two Renishaw encoders disagreeing with each other. Zero'd when the dispatcher is restarted.

`CT_NOTOK` Count of occurrences of an encoder's status bits being not-ok. Zero'd when the dispatcher is restarted.

`CT_RANGE` Count of occurrences of the encoders being out of the permitted range of values. Zero'd when the dispatcher is restarted.

`CT_SPEED` Count of occurrences of overspeed condition. Zero'd when the dispatcher is restarted.

`ENC_AGREE` `Yes` if the two encoders agree with each other; otherwise `No` or `Unknown`.

`ENC_INRANGE` `Yes` if the encoders are in the permitted range; otherwise `No` or `Unknown`.

`LOOPSTATUS` Status of the code running on board the Galil. One of:

`OK`: loop is running as expected.

`No Data`: no data has been received from the Galil in 3 seconds.

`Bad Record`: the Galil sent an invalid "Data Record".

`Not Running`: data records are being auto-broadcast by the Galil and received by the dispatcher, but the on-board code loop is not running.

`SPEED_OK` `Yes` if the speed is within the speed limit; otherwise `No` or `Unknown`.

`SUMMARY` A terse list of all the safety status keywords that are in a not-OK state, alternating with their values.

The `CT_xxxx` keywords are useful because some conditions can be transitory, triggering an Abort/ELO on the swingarm Galil but lasting for only a fraction of a second. The counter keywords are persistent, and so will show the event that occurred even if the problem already went away.

Encoder Data

`ARMDIFF` The difference of the absolute encoders in motor encoder units: `(ARM_A_ENE - ARM_B_ENE)`. Not to be confused with *k1dm3*'s `ARMDIFF` keyword, which is the difference between the motor encoders.

`ARM_A_ENE` The expected actuator A encoder value, derived from `ENCODER0`.

`ARM_B_ENE` The expected actuator B encoder value, derived from `ENCODER1`.

`ENCODER0`. The RIO's Encoder 0 value. This is the encoder mounted on actuator A.

`ENCODER1`. The RIO's Encoder 1 value. This is the encoder mounted on actuator B.

`ENCBAD0`. Distance of Encoder 0 from the V-block when the last problem triggered, in units of absolute encoder counts. That is, it is 0 when in the V-block, and something like 507,000 counts in the Retract position. The `ENCBAD`*x* keywords are meaningful only when an error occurs, in which case one of the counter keywords (`CT_`*xxx*) will have incremented at the same time that the `ENCBAD`*x* keywords are set. (Note that the `CT_`*xxx* keywords are reset at each safety RIO power-up; they only indicate errors when they increment.)
N.B. Each time the safety RIO is powered up and initializes, it briefly transitions through an error state, triggering the recording of the absolute encoder values (not their distance from the V-blocks) into the `ENCBAD`*x* keywords. Always use the `CT_`*xxx* keywords to check whether an error occurred when values were last latched into the `ENCBAD`*x* keywords.

`ENCBAD1`. Distance of Encoder 1 from the V-block when the last problem triggered, in units of absolute encoder counts. That is, it is 0 when in the V-block, and something like 507,000 counts in the Retract position. See the additional discussion under `ENCBAD0`.

`MAXSPEED` The maximum value of `SPEED` over the previous second.

`SPEED` The current speed of the swingarm, in absolute encoder counts/sec, as computed by differencing successive encoder values. The Galil RIO reads the encoders at 25 ms intervals, whereas the safety loop is updating at 20 ms intervals, so roughly every eight reading will show speed=0.

Supplementary

`INPUTS` All 16 digital inputs, presented as a hexadecimal value.

`OUTPUTS` All 16 digital outputs, presented as a hexadecimal value.

`REGION` General location of the swingarm, according to the region-indicating output bits `#4` and `#5` (see section 8.2). Normal values are `VBlock`, `Retract`, or `High Speed`. The value `Unknown` is used when `LOOPSTATUS` is not `OK`, and the value `ErrorBothActive` means that both bits `#4` and `#5` are active.

`SAMPNUM` The "sample number" from the data record of the Galil RIO. For our purposes, it's a heartbeat signal.

`SE0`, `SE1`. These are the values of the `_SE0` and `_SE1` operands from the Galil RIO, and report the status values of the absolute encoders. Possible values are `OK`; `Encoder timeout` (encoder did not set the "start bit" within $30\mu s$); `CRC Error` (CRC checksum error in data from encoder); `Encoder error` and `Encoder warning`.

`ZC`, `ZD`. These are the values of the Galil RIO "user variables" of the same name, which are included in the data record. The on-board code packs the values of these Galil RIO operands into `ZC`: `_SE0`, `_SE1`, and `_RS`. The on-board code uses `ZD` as a loop counter for trips through the safety monitoring code.

Chapter 14

k1dm3mon KTL Service Keywords

This section summarizes the keywords provided by *k1dm3mon*. Additional information — such as data type, data range, allowed values — is available using *gshow* to query the keyword data, using its various help options.

As mentioned in section 9, the *k1dm3mon* service is an instance of the *emir* application. A general discussion of *emir*'s concepts of *groups* and *conditions* is found in (Deich, 2014). Here, we outline the groups and conditions that make up *k1dm3mon*.

K1dm3mon is made up of two dispatchers. Dispatcher #1 is the main dispatcher, containing all monitoring except for reporting whether the *k1dm3* and *k1dm3saf* dispatchers' heartbeats are present. Dispatcher #2 monitors those heartbeats.

Each condition *cond* has the following keywords:

***cond*REM** Brief description of what the condition monitors.

***cond*STA** Current status of the condition. Values can be:

Standard: OK, NOTICE, WARNING, ERROR, CRITICAL.

Exceptional: Priming, DISABLED, CANT_COMPUTE, NO_HEARTBEAT.

The standard values are those that are directly set by the condition. The exceptional values are set by the *emir* daemon, and give a reason that the condition is not being evaluated normally: Priming means the the daemon is initializing; DISABLED means that a user has disabled updates by setting the ***cond*DIS** keyword (either directly, or via a parent group's DIS keyword); NO_HEARTBEAT means that the condition can't be evaluated because it depends on a keyword from a service whose heartbeat has stopped; CANT_COMPUTE means that the condition can't be evaluated because it depends on another condition or keyword that is currently unavailable, typically due to NO_HEARTBEAT.

***cond*ACK** An active alarm is any condition or group whose status is not OK or DISABLED. The ***cond*ACK** keyword is used to acknowledge the alarm. Unlike the ***cond*DIS** keyword, an acknowledged condition is still active, but it won't send email or texts, and other workers can see that the alarm has been acknowledged and is being handled (presumably) by someone else.

*cond*DIS Disables a condition. The value is either a timestamp (e.g. 4pm or 2020-12-03T12:13:14), or a duration with the word hence added, e.g. 4 hours hence). The condition will be ignored until the timer expires (or is reset by setting it blank or to a time in the past).

*cond*MSG Message string to accompany the *cond*STA.

Each group *grp* has the following keywords:

*grp*MEM A list of the group's member groups and conditions.

*grp*CNO An integer, it reports the number of conditions that are not in an OK state (CNO = Conditions-Not-Ok), recursively including all child conditions.

*grp*ACK Like the per-condition ACK, but applied recursively to its members.

*grp*DIS Group disable setting. Like the per-condition DIS, but applied recursively to its members.

*grp*MSG Group status message. This is computed from the combined messages of its members.

*grp*REM Brief description of the group.

*grp*STA Group status code. This is generally computed as the "worst" status among its members.

The *k1dm3mon* groups and conditions are as follows:

Group DISP2TOPLEVEL: monitors the "heartbeat" keywords from the K1DM3 dispatchers and the keyword history service.

- Condition HISTORY: State of keyword history service
- Group GALILS: monitors for the various Galil dispatchers
  - Condition DM3DRUM: Drum (rotation) Galil
  - Condition DM3DOCK: Dock RIO
  - Condition DM3SWINGARM: Swingarm Galil
  - Condition DM3SAFETY: Safety System RIO

Group DISP1TOPLEVEL: monitors everything except heartbeat keywords.

- Group SWINGARM: Various swingarm checks
  - Condition DM3_1CONN: Check for reconnection problem.
  - Condition MOTORON_A: Check for excessive motor-on time, actuator A.
  - Condition MOTORON_B: Check for excessive motor-on time, actuator B.
  - Condition MOTOR_OVERTEMP: Actuator over-temp status.
  - Condition ELEVATION: Swingarm elevation monitor. Alert if deploy is occurring at other than 67°.
  - Condition SECURED: In-tower swingarm position check. Warns if neither deployed+clamped, nor retracted+docked.

Group SAF_COUNTERS: Safety condition counters. Keeps track of the *k1dm3saf* counters that reflect the following problems:

Condition ENC_DISAGREE: Swingarm encoder agreement.

Condition ENC_OORANGE: Swingarm encoder out of range.

Condition ENC_NOTOK: Swingarm encoder error flags.

Condition ENC_OVERSPEED: Swingarm encoder overspeed.

Group DRUM: Rotation Checks.

Condition DM3_2CONN: Check for reconnection problem.

Condition ROTMAXSPD: Check that slews are done at high speed. (Catches rare situations in which speed was set low for some engineering tasks, then not reset to full speed.)

Condition MOTORON_DRUM: Check for excessive motor-on time.

Condition DETENT_MONITOR: When air is engaged, detent must be engaged.

Condition MOVEDIR: Preferred move direction should be Quick. (The MOVEDIR keyword can direct rotation to always be positive, always be negative, or take the least-time path. Least-time should always be used except for certain engineering purposes.)

Group DOCK: Conditions on docking

Condition DM3_3CONN: Check for reconnection problem.

Condition DOCK_RETRACT: Check that dock is engaged at retract.

Condition DOCK_MOVING: Check that dock is retracted during motion.

Condition WITHOUTDOCK: Check that swingarm configuration isn't ignoring the dock.

Group OTHER: Other Conditions

Condition PRESSURE_MONITOR: Alert on low air pressure.

Condition SWLOCK: Alert if there's a SWINGARM or DRUM software lockout.

Group POWER: Monitor sources of heat.

Group PWRSUPPLIES: Power Supplies

Condition OUTER48V: Alert if drum motor power has been on for over 5 minutes.

Condition INNER24V: Alert if inner drum 24V supply has been on for over 5 minutes.

Condition INNER48V: Alert if inner drum 48V supply has been on for over 5 minutes.

Group SOLENOIDS: Solenoid Active Bits.

Condition DRUMBITS: Alert if any outer drum solenoid output bits have been active for over 5 minutes.

Condition SWINGARMBITS: Alert if any inner drum solenoid output bits have been active for over 5 minutes.

Condition DOCKBITS: Alert if any dock solenoid output bits have been active for over 5 minutes.

Appendix A

Useful Numbers

A.1 Network Addresses

128.171.95.33	Instrument host, outside interface.
192.168.23.100	Instrument host, internal interface.
192.168.23.101	Safety System RIO-47142.
192.168.23.111	Swingarm DMC-4040.
192.168.23.112	Rotation DMC-4040.
192.168.23.113	Dock RIO-47142.

A.2 Swingarm Safety System

Swingarm Safety System

Arm A, V-block position	8971191 Renishaw encoder counts
Arm B, V-block position	9841984 Renishaw encoder counts
Safety Trippoints	
Max distance beyond V-block	800 Renishaw counts $\sim$ 630 motor counts
Max distance beyond Retracted	1900 Renishaw counts $\approx$ 1500 motor encoder counts
Size of slow speed zones:	10 mm of actuator motion = 40960 load encoder counts $\approx$ 32250 motor encoder counts
Max speed, slow zone	6350 Renishaw counts/sec = 5000 motor counts/sec
Max speed, fast zone	50800 Renishaw counts/sec = 40000 motor counts/sec
Max encoder difference	2048 Renishaw counts $\approx$ 1610 motor counts

A.3 Swingarm Actuators

Motor encoder	8192 counts/rev = 8192 counts/0.1” ≈ 3225.2 counts/mm
Renishaw load encoder	4096 counts/mm
Renishaw:motor encoder ratio	1.27
Backlash	≈ 200 motor encoder counts ≈ 60μ
Torque limit	7.7 amp continuous, 22.5 amp peak
Retracted position	104.5 deg from X-Y plane (Z-axis is the drum’s axis of rotation) = 507111 Renishaw counts = 399300 motor counts
Speed in slow zone	3810 Renishaw counts/sec = 3000 motor counts/sec
Speed in fast zone	31750 Renishaw counts/sec = 25000 motor counts/sec
Size of slow-speed zones	50800 Renishaw counts = 40000 motor counts ≈ 5.3 cm of swingarm motion
Positioning tolerance	50 motor counts ≈ 39 Renishaw counts ≈ 15.5μ

Swingarm positions

Position (`ROTATORD`)	Name (`ROTATNAM`)
0	Deploy
1	Retract
2	Mirror90

A.4 Drum

Renishaw encoder resolution	200 counts/mm
Renishaw:motor encoder ratio	$\sim$1:5.116651 $\approx$ 0.19544
Gear head ratio	60:1
Ring/pinion gear ratio	15.12:1
Motor encoder counts/revolution	4096
Backlash	$\approx$ 1900 motor encoder counts
Full Circle	Expect: $60 \times 15.12 \times 4096 = 3715891.2$ motor counts. Measured: $\sim$ 726256 Renishaw counts $\approx$ 3716000 motor counts
Usable Renishaw tape range	16000 – 735000 Renishaw counts (719000 counts) The physical tape gap is about 1-2mm, but the readhead is physically a few cm long, and the readout is unreliable anywhere near the gap. The dispatcher acts as if a region of about 3.6 cm of tape is unusable, and rehomes whenever it crosses that segment of tape.
Positioning tolerance	5 load encoder counts $= 25\ \mu$ $\approx$ 25.6 motor encoder counts
Rotation speed	40000 motor encoder counts/sec $\approx$ 39 mm/sec $\approx$ 7817 load encoder counts/sec

V-block positions

Position (`ROTATORD`)	Name (`ROTATNAM`)	Angle (`ROTATVAX`)
1	LNas	89.802
2	LBC1	49.943
3	LBC2	38.552
4	MirrorUp	−0.429
5	RBC2	−38.641
6	RBC1	−50.224
7	RNas	−90.101
8	Cass/Stow	179.518

A.5 Typical Move Times

— Between —	Time (s)
LNas, RNas	62
LNas, Cass	118
LNas, LBC1	23
LNas, LBC2	28
RNas, Cass	112
RNas, LBC2	48
Cass, Stow	80
LBC1, LBC2	18
LBC1, Stow	54

Actual move times vary, primarily due to the variable time to detect that the swingarm Galil has powered up, for the swingarm dispatcher to make its initial connect, and the time for pneumatic components to move.

A.6 Other Numbers

Clamps-closed analog signal:	The clamps-closed analog signals from clamps {A,B,C1,C2} are, respectively, {1,2,4,8} ×660 mV.

Appendix B

Actuator Calibration and Position Adjustment

This appendix reviews the positioning tolerances, and explains how to re-adjust or calibrate sensors if they become out of position or fail. Position tolerances are set tightly for engineering purposes, even though the science requirements are much looser.

B.1 Swingarm Absolute Encoders

The procedure for replacing and/or aligning a readhead is covered in Technical Note 872-LTN2006, Ratliff et al. (2019). Here we provide some additional background information.

The swingarm is calibrated using the Renishaw absolute encoders, but thereafter the motion is controlled using only the motor encoders. The positioning tolerance for the actuators is 50 motor encoder counts, or $\sim 15.5\mu$. In practice, the actuators position to 1-2 motor encoder counts. The actual variation reported by the load encoders, when the swingarm is moved repeatedly into and out of the V-blocks, is about 50 load encoder counts, or about 12μ. This is due partly to backlash in the actuators, and partly due to the variability caused by the compliant hinge.

The two Renishaw absolute encoder tape strips have different ranges: when the swingarm is in the V-blocks, the Arm A encoder reads approx 8,972,000 counts, and the Arm-B encoder reads approximately 9,842,000 counts. Both of them count in the reverse direction from the motor encoders: the motor encoder counts increase as the swingarm retracts; the absolute encoder tape counts decrease. Those values are burned into the code on board the safety RIO, and are used by the *k1dm3saf* KTL service when it converts absolute encoder position to the corresponding actuator motor encoder position, but the values are not known to any software outside of *k1dm3saf*.

Note that:

- If the Renishaw readhead mounting is shifted slightly, the stored values for the absolute encoder positions must change accordingly.
- The Renishaw encoder tape is attached to one of the anti-rotation arms of the actuator, which in turn is clamped to to the actuator shaft. If the clamped position changes at all, the stored values must change accordingly.

- Of course, if a readhead or the encoder tape is replaced, the stored values must be updated.

B.2 Drum Readhead

The drum orientation is measured by a Renishaw incremental encoder, with distance-coded reference marks for homing.

The drum rotation tolerance* is 5 load encoder counts, or 25μ. In practice, the drive can position to within 1 count of the target position, but of course the final position is determined by the detent engaging in a v-block.

If the drum readhead position is disturbed, the reported drum orientation will change by the same amount. When commanded to an instrument position, the dispatcher will rotate the drum to a position that has the same amount of error. If the error is small, the detent mechanism will pull the drum into position when the detent is engaged. If the error is more than 25μ (the drum positioning tolerance), the dispatcher will report this position as "unknown", which will prevent normal operation of the module. Thus, if the drum readhead position is changed, all of the configured-in instrument positions must be modified by a corresponding amount.

Always ensure the readhead position is changed by less than a few mm before attempting recalibration. If changed by more than a few mm, you would have to use a different, more painstaking procedure to re-determine the instrument positions, making sure the detent isn't engaging outside the valid V-block regions; it is simpler to re-position the readhead to within a few mm.

After ensuring the readhead position has changed by less than a few mm, the most direct way to correct the settings is as follows:

1. Command the drum to each instrument position.
2. At each position, when the detent mechanism engages, it will pull the drum into the correct position. (This method is only suitable when the shift is ≤ 1 cm, to ensure that the detent mechanism is only operated when it's near the actual v-block position.)
3. Record the `ROTATRAW` value for the position.
4. Use the `ROTATRON` keyword (see 12.4) to update the mapping between named and encoder positions.
5. After updating `ROTATRON`, command the module to each of the drum positions again, and confirm the new `ROTATRON` positions are correct.
6. After confirming that `ROTATRON` is correct, update the `RON_LIST` definition in the svn source file `kroot/kss/K1DM3/k1dm3/xml/ROTAT.defs`, then change directory to `kroot/kss/K1DM3/k1dm3/`, `make install`, `kdeploy`, and `svn commit` so that the configuration data in svn matches the cached `ROTATRON` values.

* A tolerance of x counts means that the dispatcher considers a move to have succeeded if the achieved position is within x counts of the target position.

Appendix C

Modifying *k1dm3saf* Galil Code

On rare occasions, you may need to modify the code burned into the Safety System Galil RIO. For example, you may decide permanently change any of the parameters configured into the Galil, such as the size of the "slow" zones.

The safety system Galil code is in the svn repository location

```
kroot/kss/K1DM3/k1dm3saf/galil/
```

To change a configuration parameter, edit the file `safetyGalil.defs`. Most of the parameters are defined directly in this heavily-commented file.

To change the Galil logic, edit `safetyGalil.code.in`. Embedded in that file are lines such as

```
REM ...
```

which is comment text, and is stripped before downloading to the Galil, and lines that look like:

```
REQUIRE hiband $(HIBAND)
```

which might read, after substitution of the macro:

```
REQUIRE hiband 40960
```

The `REQUIRE` lines, are used by our Galil downloader application, *gdownload*, to know that Galil burned-in variables should have specific values, such as `hiband` = `40960`.

After editing the `safetyGalil.`*xxx* files, you must generate the updated code and download it to the Galil as follows:

1. On the build host, `k1dm3build`, generate the code for the galil from the `*.in` and `*.defs` files, and install on the instrument host:

   ```
   % make install
   % kdeploy -a
   ```

2. On the instrument host, `k1dm3server`, download the code to the Galil, and burn it in:

(a) Stop any currently-running code:

```
% telnet k1dm3-galil-safety
: ST
```

(b) Download the new code to the RIO:

```
% safety.galil.installer download
```

Answer `y` to the question about proceeding.

If you failed to stop the running code before doing the 'make download', you'll get a diagnostic like the following:

```
:
:MG_NO
 6.0000
:Can't download: threads are running! MG _NO = 6.0000
make: *** [dl-safety] Error 1
```

In that case, telnet to the Galil and do `ST`, as shown above, in step 2a.

If the download doesn't finish correctly, you'll get a diagnostic such as the following:

```
::1 Unrecognized command
:
Failed to get ":: 0\n:" ack in 2000 ms; got: "::1 Unrecognized command
:"
```

In that case, repeat the download command (step 2b), and the download will always succeed (in my experience).

(c) If any Galil variables (parameters) are changed, you'll see a diagnostic similar to the following:

```
------------------------------
Require X0retr=506984, but on Galil X0retr=504650.0000
22/23 variables matched
```

In that case, you must telnet to the Galil, hand-enter the required values, and burn them in using `BV`. For example, continuing the example above, you'd type:

```
% telnet k1dm3-galil-safety
: X0retr=506984
: BV
```

(d) Then burn in new code, reset the Galil to trigger any `#AUTO` code, and burn in any parameters that were adjusted by the `#AUTO` code:

```
: BP
: RS
: BN
```

3. Verify that the code on board the Galil still matches the expected code, after doing the reset:

```
% safety.galil.installer verify
```

Answer `y` to the question about proceeding.

If you forgot to burn variables (step 2c) or burn the code (step 2d), you'll see mismatches here.

Appendix D

Modifying *k1dm3* Galil Code

From time to time, an update to the *galildisp* application will require updated code on board the Galil.

D.1 Introduction

Galildisp stores the running version of the on-board code in:

`$RELDIR/var/`*`svc`*`/`*`svc`*`_dispatch_`*`n`*`/download.galil`

For K1DM3, *svc* is `k1dm3`, and *n* is the dispatcher number: `1` for swingarm, `2` for drum, or `3` for dock.

You can test if an update to *galildisp* has changed the code by generating the new version and comparing it to the existing version. To generate the code required by the currently-installed version of *galildisp*, use:

`$RELDIR/bin/galildisp.tcl -gx` *`path svc dispnum`*

For example, for *k1dm3* dispatchers #1 and #2 (swingarm control and rotation control, respectively), use:

```
$RELDIR/bin/galildisp.tcl -gx "foo1" k1dm3 1
$RELDIR/bin/galildisp.tcl -gx "foo2" k1dm3 2
```

In this example, *galildisp.tcl* will generate the code, leave it in files *foo1* and *foo2*, respectively, and then exit. (If you pass an empty string for the *path*, *galildisp.tcl* will install the code into its runtime location.)

The generated code always begins with a Galil no-op instruction that contains a checksum:

`NO CKSUM` *`nnnnnn`*

By comparing the first line of a file you generate (*foo*n) with the first line of the existing runtime file, you will know if there are any changes required to the onboard code.

When a *galildisp* dispatcher is (re)started, it always generates the code that it expects to find on board, burned into the Galil. If that version of the code isn't in the `download.galil` file, it moves the previous version to the `Old` subdirectory, and freshens `download.galil` with the correct code. The dispatcher then connects to the Galil, compares the on-board checksum to the checksum in the on-disk file, and quits if the checksums don't match.

D.2 Downloading Code for *Galildisp*

Generally, the sequence for downloading and burning-in fresh code for the Galil is:

- Stop the dispatcher if it is running.
- Stop any code currently running on the Galil.
- Download the new code.
- (optional) Burn in the new code.
- (optional) Execute `RS` to execute any `#AUTO` routine from the new code.
- (optional) Execute `BN` and `BV` to burn in parameters and variables set by the Galil automatic routine `#AUTO`.
- Verify that the code on board the Galil matches the source file.
- Restart the stopped dispatcher.

In detail, the steps are:

1. Stop the dispatcher if it is running, using one of the following:

   % `k1dm3 stop swingarm` *or*
   % `k1dm3 stop drum` *or*
   % `k1dm3 stop dock`

2. Stop any code currently running on the Galil, being sure to select the correct Galil:*

   - % `telnet k1dm3-galil-arm` *or*
     % `telnet k1dm3-galil-drum` *or*
     % `telnet k1dm3-galil-dock`
   - Press Return a couple of times until you get a colon (`:`) prompt.
   - Enter `ST`
   - Wait for the next colon prompt.
   - Quit telnet.

3. Download the new code.

   Run the correct command to download to the correct Galil:

   % `k1dm3.galil.installer download drum` *or*
   % `k1dm3.galil.installer download swingarm` *or*
   % `k1dm3.galil.installer download dock`

   Answer `y` to the question about proceeding.

   The command will execute the correct download command, and will include a check that the serial number has the expected value.

   Serial numbers and Galil IP addresses are defined in

   `svn/kroot/kss/K1DM3/Hosts/k1dm3Hosts.defs`

* For convenience, the `k1dm3server:/etc/hosts` file defines the following hostnames: `k1dm3-galil-arm`, `k1dm3-galil-drum`, and `k1dm3-galil-dock`. However, these are not used by the K1DM3 software itself, which always uses the IP addresses defined in `svn/kroot/kss/K1DM3/Hosts/k1dm3Hosts.defs`. These entries *should* match the convenience entries in `/etc/hosts`.

Special case: if you are preparing a new Galil that doesn't have the address of one of the in-use Galil's, you can hand-issue the download command:

```
$RELDIR/bin/gdownload --download galilIPaddress codepath
```

e.g.

```
$RELDIR/bin/gdownload --download 192.168.23.222 \
     $RELDIR/var/k1dm3/k1dm3_dispatch_1/download.galil
```

where *galilIPaddress* is the new Galil's IP address, and *codepath* is the path to the Galil code. Double-check that you are using the correct IP address!

4. Optional steps: Burn in the code, then reset the Galil (causing the Galil's `#AUTO` routine to run), and burn in variables and parameters. If you don't burn them in, they won't persist across a Galil power cycle or reset. In turn, the `#AUTO` routine may have set parameters or variables that should always persist, whether or not `#AUTO` runs, so you should probably also do `BN` (burn parameters) and `BV` (burn variables).

   The K1DM3 E-stop switch activates the Abort input on the Galil. By default, this stops programs as well as axes, and so it will stop the `#AUTO` routine from running. So, in order to be able to run `#AUTO`, we need to configure the Galil to *not* stop programs from running, which is done with `CN,,,,1`, and then burn that setting so that it survives a reboot.

   To do these steps:

   - Telnet to the Galil:
     % `telnet k1dm3-galil-arm` *or*
     % `telnet k1dm3-galil-drum` *or*
     % `telnet k1dm3-galil-dock`
   - Press Return a couple of times until you get a colon prompt.
     After each of the following commands, wait for a colon prompt. (Some of these commands will take up to a couple of seconds to return a prompt.)
   - Burn the program: `BP`
   - Set no-stop-programs, and burn: `CN,,,,1;BN`
   - Reset the Galil, and let its `#AUTO` run: `RS`
     Wait for 1 second after the fresh prompt, then:
   - Burn in the parameters set by `#AUTO`: `BN`
   - Burn in the variables set by `#AUTO`: `BV`
   - Quit telnet.

5. Finally, check that the correct code is still present after any reset.

   If you used `k1dm3.galil.installer`, then you can simply do one of:

   % `k1dm3.galil.installer verify drum` *or*
   % `k1dm3.galil.installer verify swingarm` *or*
   % `k1dm3.galil.installer verify dock`

   Otherwise, you can type in the command yourself:

```
$RELDIR/bin/gdownload --verify galilIPaddress codepath
```

6. Restart the dispatcher that was stopped in step1:

   % `k1dm3 start [ swingarm|drum|dock ]`

Appendix E

How to do a Fresh Install

From time to time it may be necessary to do a completely fresh install of the K1DM3 code. This might occur after the instrument computer is replaced, or after the OS is updated, or simply to test a new major software release.

> *Note:* the `$KROOT/var` directory is not simply impermanent data. Be sure to preserve the existing `$KROOT/var`, and copy it to the new installation. You may discard the older logs if you want from `$KROOT/var/log/`, but do not discard the data in `$KROOT/var/`*`svc`*`/` and `$KROOT/var/state/`. Just as the operating system stores various data files in `/var/`, such as cron files in `/var/spool/cron/`, so do the K1DM3 daemons store non-volatile user-supplied configuration data in subdirectories of `$KROOT/var`.
>
> For example, *k1dm3* stores its user-configurable keywords, such as `EL_MAXWAIT` (the maximum time allowed for the telescope to reach the required elevation for deploy or retract), under `$KROOT/var/k1dm3/`. Similarly, *k1dm3mon* stores its user-configurable keywords, such as `MAX_MOTOR_ON` (the maximum time motor power may be on before raising an alert), under `$KROOT/var/state/k1dm3mon/`.

E.1 Backups

The following are the minimum system backups that should be done for the instrument computer, in order to assure a seamless transition if the K1DM3 software needs to be re-installed:

- All changed system configuration items.
- The directories `/kroot/var/k1dm3/` and `/kroot/var/state/k1dm3mon/`. Recommendation: back up these directories daily.

E.2 System Configuration

The svn repository directory `K1DM3/Doc/` has a set of files that fully document the instrument host configuration as it was during instrument development and testing at UCO, in the files named `HostAdmin`*`xxxx`*`.txt`. Copies of all changed files are in the subdirectory

`ModifiedFilesPreship/`. However, the OS was re-installed at Keck to suit WMKO's preferred configuration, so at this writing, the files in the repository are somewhat dated. Among other things, the set of installed packages is smaller than the set used at UCO; the network interfaces are naturally configured differently; no Postgres daemon is running on the instrument host; and of course *dcsRotSim* is not in use on the production server.

Nonetheless, the `HostAdminxxxx.txt` files contain important system configuration changes that should be preserved across host replacements, so until they are superceded by WMKO-provided configuration information, they should still be consulted for configuring a new system.

E.3 Software Installation

Software installation is straightforward:

1. On the new instrument host, `k1dm3server`, use your backups to restore the directories `/kroot/var/k1dm3/` and `/kroot/var/state/k1dm3mon/` to their last-known state.
2. On the build host, `k1dm3build`, as user `k1dm3bld`, check out the module `k1dm3` from the repository.
3. Build and deploy the software following the standard WMKO deploy instructions.

Bibliography

Deich, W. T. S.
2014. *EMIR*: a configurable hierarchical system for event monitoring and incident response. In *Proc. SPIE*, volume 9152.

Galil Motion Control
2016. RIO-47xxx User Manual, rev. 1.0q.

Lanclos, K. and W. T. S. Deich
2012. A Complete History of Everything. In *Proc. SPIE*, volume 8451.

Lupton, W.
1997. Updated DCS Keyword Reference Manual (partial).

Ratliff, C.
2018a. K1DM3 Design Note: 872-LTN1039. *UCO Technical Notes*, 872(1039).

Ratliff, C.
2018b. K1DM3 Design Note: 872-LTN1040. *UCO Technical Notes*, 872(1040).

Ratliff, C., W. Deich, and J. Cabak
2019. K1DM3 Design Note: 872-LTN2006. *UCO Technical Notes*, 872(2006).

Sandford, D.
2019. K1DM3 Electronics — placeholder reference. *UCO Design Notes??*

Tripsas, A., D. Cowley, and J. Cabak
2016. K1DM3 Design Note: Come-along Procedure at Horizon. *UCO Design Notes*.

Subject Index

`M3AGENT`, 29

Adjusting Encoders, 109
Assemblies
    `CLAMPS`, 22, 89
    `DRUM`, 23, 87
    `M3MAN`, 23, 89
    `SWINGARM`, 22, 87
Assembly, 22

Bypassing Sequencers, 42

`Cass`, *see* Positions
Components, 21
Compound keyword, 22
Control
    Low-level, How-to, 48

Deploy
    Elevation, 13, 44, 93
`Deploy`, *see* aso Positions13
Digi-stage, 21
Dispatcher, 14
    No-Dock version, 19
    Standard Interfaces, 29
    Starting/Stopping, 17
    Suspending, 19
Drum
    Minimum conditions, 42
    Overview, 6

Electronic lockout, *see* Lockout, Electronic (ELO)
Elevation
    Deploy, 13, 44, 93
    Retract, 13, 94
Elevation Requirement, 12
ELO, *see* Lockout, Electronic (ELO)
    (Electronic lockout), 62

Galil, 13
Galildisp, 14, 66
Graphical Interfaces, 36
    Tcsgui, 36
    k1dm3_gui, 38
    k1dm3mon_gui, 40
    Engineering, 38
GUI's, 36

Hand Paddle, 29, 33

Interfaces
    Graphical, 36
Interlocks, 41
    `XMV` keywords, 41

k1dm3 service, 21
k1dm3.io, 74
k1dm3.summarize, 71
k1dm3_gui, 38
k1dm3mon, 64
k1dm3mon_gui, 40
k1dm3saf service, 62
Keyword List, *see* Index of Keywords

`LBC1`, *see* Positions
`LBC2`, *see* Positions
`LNas`, *see* Positions
Lockout, software
    Recovery, 53
Low-level Control
    How-to, 48

`M3AGENT`, *see* Sequencer

Manual Operations, 29, 33
Minimum Operating Conditions, 42
Mirror90, *see* Positions
MirrorUp, *see* Positions
Monitor Service, 64
Motion Controllers, 13
Motor stage, 21

Network, 15
No-Dock Dispatcher, 19

Overrides
  XSAFETY flag, 44
  Clamps, 44, 45
  Drum, 46
  Non-overrideable, 43
  Swingarm, 44, 45

Position Tolerances, 109
Positions
  Cass, 13
  Deploy, 13
  LBC1, 13
  LBC2, 13
  LNas, 13
  Mirror90, 13
  MirrorUp, 13
  RBC1, 13
  RBC2, 13
  Retract, 13
  LNas, 13
  Stowed, 13
Power Outage, 56

RBC1, *see* Positions
RBC2, *see* Positions
Recovery
  After power outage, 56
  General Tips, 47
Recovery – Specific issues
  Air supply stuck, 52
  Amplifier fault, 50
  Dispatcher reconnect fails, 49
  Dock pin reporting incorrect, 56
  Dock pin stuck, 55
  Galil reconnect fails, 49
  In-tower switch failure, 56
  Mirror-down operation, 52
  Misaligned actuators
    Arms not past deploy point, 54
    Arms past deploy point, 54
  Software lockout, 53
  Swingarm encoder – bad value, 55
  Swingarm in hard stop, 53
  Swingarm overshot V-blocks, 55
  Swingarm past deploy point, 53
  Unclosed clamps, 51
    false alarm, 50
  USEDCS engineering flag, 56
Replacing Encoders, 109
Restarts
  Periodic, 20
Retract
  Elevation, 13, 94
Retract, *see* Positions
RNas, *see* Positions
Routine Operations
  M3AGENT, 29
  Hand Paddle, 29
  TCS, 29

SASS, *see* Swingarm Safety System
Sequencer, 23
  m3.*xxxx*, 32
  advantages, 24
  Bypassing, 42
  example, 23
  monitoring, 27
Sequencers
  ACTIVATE, 24
  M3AGENT, 27, 32
Services
  k1dm3, 14, 21
  k1dm3mon, 14, 64
  k1dm3saf, 14, 62
  Starting/Stopping, 17
Software lockout
  Recovery, 53
Stages
  AIRSUPP, 78
  ARM_A_, 79
  ARM_B_, 79
  CLAMP_*x*, 78
  DETENT, 78
  ROTAT, 79
  ROTB, 79
Stowed, *see* Positions

Suspending Dispatchers, 19
Swingarm
    Minimum conditions, 43
    Overview, 9
Swingarm Safety System, 60

TCS, 27, 31
Tcsgui, 36
Telescope
    Elevation, *see* Elevation
Temporary Override, 58
Tolerances, 109

USEDCS engineering flag, 31, 56
Utilities
    k1dm3.io, 74
    k1dm3.summarize, 71

XMV keywords, 41

KTL Keyword Index

Emir Standard Keywords
- *cond*ACK, 101
- *cond*DIS, 102
- *cond*MSG, 102
- *cond*REM, 101
- *cond*STA, 101
- *grp*ACK, 102
- *grp*CNO, 102
- *grp*DIS, 102
- *grp*MEM, 102
- *grp*MSG, 102

Galildisp Standard Controller Keywords
- CTRL*x*ADR, 77
- CTRL*x*AUX, 77
- CTRL*x*CLK, 77
- CTRL*x*ERR, 77
- CTRL*x*GSS, 77
- CTRL*x*MSG, 77
- CTRL*x*THD, 77

Galildisp Standard Digi-axis Keywords
- *dax*CMT, 78
- *dax*CON, 78
- *dax*DCC, 78
- *dax*ENG, 78
- *dax*ENT, 78
- *dax*ERM, 78
- *dax*ERR, 78
- *dax*LCK, 78
- *dax*LIM, 78
- *dax*MSG, 78
- *dax*POS, 78
- *dax*PSE, 78
- *dax*RCD, 79
- *dax*STA, 79
- *dax*STP, 79
- *dax*TAG, 79
- *dax*TRG, 79
- *dax*XMV, 79

Galildisp Standard Digi-like Assembly Keywords
- *dig*ASM, 90
- *dig*CMP, 90
- *dig*CMT, 90
- *dig*CON, 90
- *dig*ENG, 90
- *dig*ENT, 90
- *dig*ERM, 90
- *dig*ERR, 90
- *dig*LCK, 90
- *dig*MSG, 90
- *dig*POS, 90
- *dig*PSE, 90
- *dig*STA, 90
- *dig*STP, 90
- *dig*TAG, 91
- *dig*TRG, 91
- *dig*XMV, 91

Galildisp Standard Dispatcher Keywords
- DISP*x*_BITGP, 75
- DISP*x*_COMBO, 75
- DISP*x*_COMBX, 75
- DISP*x*_DIGI, 75
- DISP*x*_MOTOR, 75
- DISP*x*BCH, 75
- DISP*x*CLK, 75
- DISP*x*DBG, 76
- DISP*x*DISABL, 76
- DISP*x*DWIM, 76
- DISP*x*ERR, 76
- DISP*x*HOM, 76
- DISP*x*HUP, 76

DISP*x*LDF, 76
DISP*x*LOG, 76
DISP*x*MSG, 76
DISP*x*NOTE, 76
DISP*x*PID, 76
DISP*x*PSE, 76
DISP*x*REQ, 76
DISP*x*RSS, 76
DISP*x*SCK, 76
DISP*x*SHUTDN, 77
DISP*x*STA, 77
DISP*x*SUSP, 77
DISP*x*THREAD, 77

Galildisp Standard Motor axis Keywords

*mot*AST, 79
*mot*CAL, 79
*mot*CED, 80
*mot*CMT, 80
*mot*CON, 80
*mot*DCC, 80
*mot*ENC, 80
*mot*ENE, 80
*mot*ENG, 80
*mot*ENT, 81
*mot*ERM, 81
*mot*ERR, 81
*mot*GSC, 81
*mot*GSR, 81
*mot*GSS, 81
*mot*GST, 81
*mot*GSW, 81
*mot*GTE, 81
*mot*LCK, 81
*mot*LIM, 81
*mot*MAP, 82
*mot*MOD, 82
*mot*MOE, 82
*mot*MOO, 82
*mot*MSG, 82
*mot*MSP, 83
*mot*NAM, 83
*mot*NPX, 83
*mot*ORD, 83
*mot*PSE, 83
*mot*RAW, 83
*mot*RCD, 83
*mot*REL, 83
*mot*RON, 83
*mot*SPD, 84
*mot*STA, 84
*mot*STP, 85
*mot*SWT, 85
*mot*TAG, 85
*mot*TNM, 85
*mot*TOR, 85
*mot*TOS, 85
*mot*TRD, 85
*mot*TRG, 85
*mot*TRL, 85
*mot*TRN, 85
*mot*TSP, 85
*mot*TVA, 85
*mot*TVX, 85
*mot*VAL, 86
*mot*VAX, 86
*mot*VEL, 86
*mot*VEX, 86
*mot*XMV, 86
*mot*ZPX, 86

Galildisp Standard Motor-like Assembly Keywords

*asy*ASM, 87
*asy*CAL, 87
*asy*CMP, 87
*asy*CMT, 87
*asy*CON, 87
*asy*ENG, 87
*asy*ENT, 88
*asy*ERM, 88
*asy*INI, 88
*asy*LCK, 88
*asy*MOD, 88
*asy*MOE, 88
*asy*MOO, 88
*asy*NAM, 88
*asy*NMS, 88
*asy*PSE, 88
*asy*RAW, 89
*asy*REL, 89
*asy*STA, 89
*asy*STP, 89
*asy*SWT, 89
*asy*TAG, 89
*asy*TNM, 89
*asy*TRG, 89
*asy*TRL, 89

*asy*VAL, 89
*asy*VAX, 89
*asy*VX2, 89
*asy*XMV, 89

K1dm3 Other Keywords

ACTIVATE, 92
ACTIVATE_*xxx*, 93
ACTIVATE_ARM, 93
ACTIVATE_DRUM, 93
ACTIVATE_OBS, 93
ACTIVATE_TRANS, 93
ACTIVERM, 93
ACTIVLOG, 93
ACTIVMSG, 93
ACTIVRUN, 93
AIRPRESSX, 95
ANALOG*d*_*n*, 95
ARMDIFF, 95
CABLES, 95
CABLESET, 95
CLAMP_*a*_CLO, 95
CLAMPCURR, 95
CLAMPSCLOSED, 96
CLAMPSTEADY, 96
CLAMPUNFILT, 96
DEADMAN{1,2,3}, 96
DOCK_HALL, 96
DOCKBITSOUT, 96
DRUMSTA_STOP, 96
EBOX_3_3V, 94
EBOX_3_3V_R, 94
EBOX_5V, 95
EBOX_5V_R, 95
EBOX_AIR_T, 95
EBOX_AIR_TR, 95
EL, 92
EL_DEPLOY, 12, 93
EL_DMD, 93
EL_DMDX, 93
EL_MAXWAIT, 12, 94
EL_NEED, 94
EL_RETRACT, 12, 94
EL_SATISFIES, 94
ELALERT, 94
ELMAX, 94
ELMIN, 94
ESTOP, 96
GETAIR, 96
INBLOCK11, 96
INNER24V, 96
INNER48V, 96
INR_AIRPRESS, 96
LK_*xxx*, 96
LOCATION, 96
M3AGENT, 92
M3ENG, 31, 93
M3ENT, 93
M3ERM, 93
M3LOG, 93
M3MSG, 93
M3RUN, 93
MAN_OVERRIDE, 97
OUTER48V, 96
OVERSPEED, 97
OVERTEMP, 97
PADDLE, 97
PENDINGSTOP, 97
PINNED, 97
ROTATSW, 97
ROTSTA_STOP, 97
ROTTIME, 92
SAF_EAG, 91
SAF_EIR, 91
SAF_EOK, 91
SAF_ESP, 91
SAF_HBT, 91
SAF_RER, 91
SAF_VBR, 91
SAFETYBITS1, 91
SAFETYBITS2, 91
SWINGSTA_STOP, 97
TERTABRT, 92
TERTABRTC, 31
TERTDEST, 92
TERTERRS, 92
TERTERVL, 92
TERTHALT, 92
TERTHALTC, 31
TERTINIT, 92
TERTINITC, 31
TERTMOVE, 92
TERTMOVEC, 31
TERTPOSN, 92
TERTSTAT, 92
TERTSTBY, 92

TERTSTBYC, 31
TERTSTST, 92
USEDCS, 91
WITHDOCK, 97
K1dm3saf Keywords
ALL_OK, 98
ARM_A_ENE, 99
ARM_B_ENE, 99
ARMDIFF, 99
CT_NOTOK, 99
CT_RANGE, 99
CT_SPEED, 99
DISPOCLK, 98
DISPODBG, 98
DISPODWIM, 98
DISPOERR, 98
DISPOHOM, 98
DISPOMSG, 98
DISPOPID, 98
DISPOREQ, 98
DISPOSHUTDN, 98
DISPOSTA, 98
ENC_AGREE, 99
ENC_INRANGE, 99
ENCBAD0, 99
ENCBAD1, 100
ENCODER0, 99
ENCODER1, 99
INPUTS, 100
LOOPSTATUS, 99
MAXSPEED, 100
OUTPUTS, 100
REGION, 100
SAMPNUM, 100
SE0, 100
SE1, 100
SPEED, 100
SPEED_OK, 99
SUMMARY, 99
ZC, 100
ZD, 100