cosylab

Motion Control Software

System Design

Revision:	0.3
Status:	DRAFT
Repository:	`https://internal.cosylab.com/svn/acc/projects/GSI/`
Project:	`FAIR-GenericSlits`
Folder:	`trunk/doc/`
File:	`DOC-GSI-FAIR-MotionControl-Design.odt`
Owner:	Gasper Jansa
Last modification:	November 9, 2012
Created:	October 17 2012

Document History

Revision	Date	Author	Section	Modification
0.1	10/17/12	gjansa	All	Created.
0.2	11/06/12	gjansa	6.2.2	Added slits class definition.
0.3	11/09/12	gjansa	6	Updated FESA properties.

Confidentiality

This document is classified as a **`public document`**. As such, it or parts thereof are openly accessible to anyone listed in the Audience section, either in electronic or in any other form.

Scope

This document describes the design of the motion control software for FAIR.

Audience

All Cosylab and GSI/FAIR members involved in the project and users of the system.

References

[1] M. Levicnik, M-Box with PDC Motion control, Implementation, XX

[2] P. Kainberger et al., DS-Shrittmotoren, Geraetmodell und Softwareentwurf, version DS

[3] PMAC software reference manual: http://www.deltatau.com/fmenu/Turbo%20SRM.pdf

Table of Contents

Table of Figures

1 OVERVIEW

This design document describes motion control front-end software to be used in FAIR project. In addition to software running on front-end local control GUI application is also included which is used for motor commissioning.

MicroIOC-M-Box-PMAC, which is microIOC- based product and hosts powerful PMAC motor controller, is used as front-end controller [1].

The software architecture is shown on Figure 1.

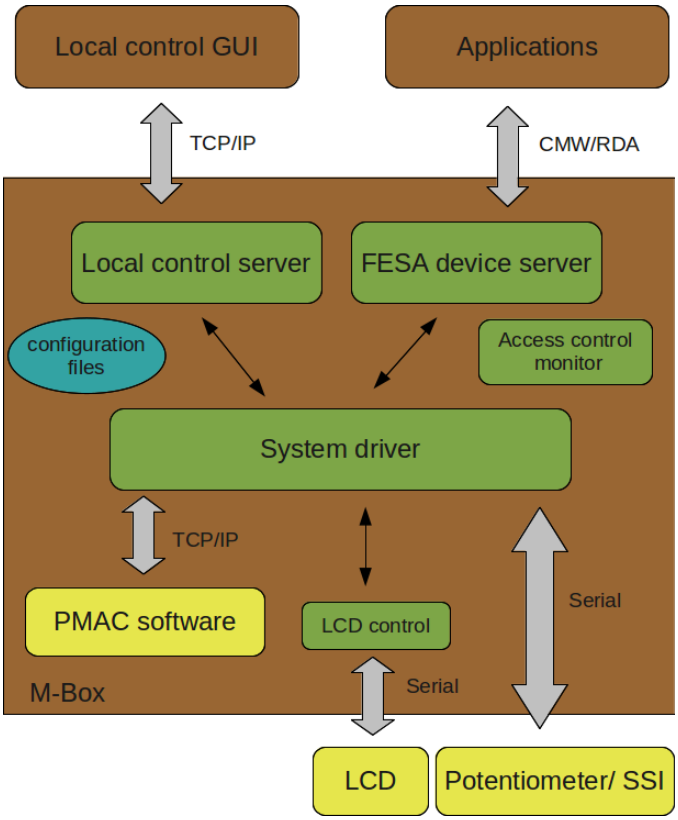

Figure 1: The architecture of motion control software

As shown on the figure above the software comprises of the following components:

- **PMAC software**
  - Low level software used to control single or pair of motors which resides on PMAC controller inside M-Box.
- **System driver**
  - C++ system driver that communicates with PMAC via TCP/IP
  - Communicates with any other additional device (e.g. encoders) via serial connection.
  - Transforms requests from upper level software to PMAC commands

  - Handles local/remote control connection
- **Local control server**
  - Local control server used for motor configuration
- **Local control GUI**
  - Local control GUI running for motor configuration
- **Access control monitor**
  - Used to implement access control fall-back and shared memory cleanup in case any of the application crashes unexpectedly.
- **LCD control**
  - Used for controlling motors locally from the LCD on the front panel of M-Box.
- **FESA device server**
  - FESA classes which model motor and pair of motors (slit)
- **Configuration files**
  - Files for system configuration

Each of these components is described in the following chapters.

2 PMAC SOFTWARE

PMAC software is divided into:

- settings and configuration
- motion programs
- PLC programs

Short description of the above is described in the following chapters.

2.1 SETTINGS AND CONFIGURATION

These are files with simple commands which get executed only when they are downloaded to PMAC. They will be used to:

- store the permanent motor configuration (e.g. PID parameters, channel addresses, ...)
- default values for motor settings (e.g. velocity, acceleration, movement range...)

2.2 MOTION PROGRAMS

When a motion program is being executed, the PMAC works through the program one move at a time, performing all the calculations up to that move. Motion programs are executed in coordinate systems, which motors are assigned to. There shall be :

- One motion program for each motor (8 motion programs)
- One motion program for each motor pair for slit positioning (4 motion programs)
- One motion program for each motor pair for executing homing procedure (4 motion programs) This depends on the type of feedback axis is using (incremental encoders require homing).

2.3 PLC PROGRAMS

PLC programs are special PMAC programs that operate asynchronously and with rapid repetition. They have the same logical constructs as the motion programs. There shall be:

- One PLC program that performs initialization upon PMAC power up. Used for I/O initialization, internal variables initialization, enabling all the motors, applying brake on the motors (if applicable)...

- One PLC for each motor that is executed upon specific event (command from upper layers of software e.g. FESA device server). They are used to stop the motion and apply brake (if applicable) and/or kill the motor.
- PLC that monitors the execution of motion programs and performs certain actions like changing the assignment of the motors to the correct coordinate systems.

3 SYSTEM DRIVER

System driver offers routines to operate single motor or pair of 2 motors. In addition to that it provides routines for system inspection (e.g. PMAC status, motor status,...) and routines to operate any other device used in the system (e.g encoders...).

The architecture of the driver is depicted on the figure below.

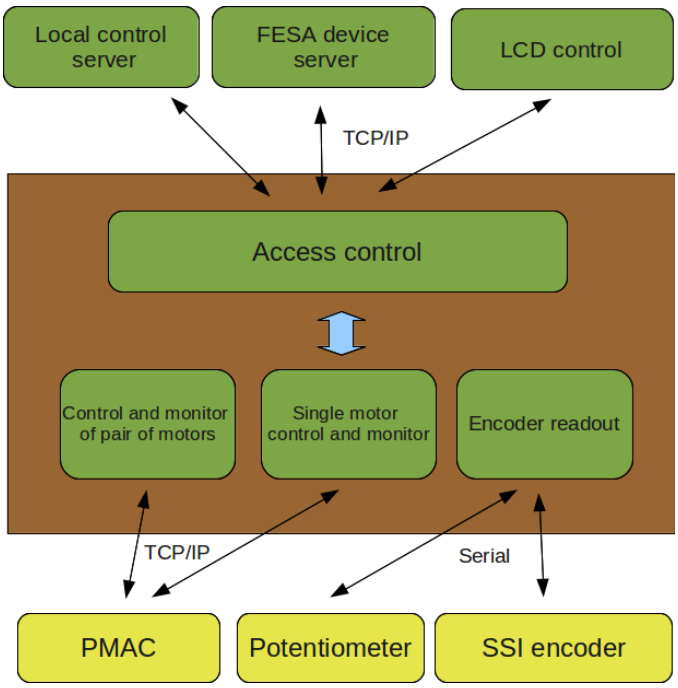

Figure 2: The architecture of the system driver

The architecture of the driver comprises:

- local/remote control logic controls access rights of the local and remote control. See Access control for details.
- Single motor monitor and control
- Control and monitor of pair of motors (slit)
- Readout of the encoders (potentiometer or SSI encoder)

Driver is written in C++ programming language. All API routines are synchronous meaning that they block until completed. This does not mean that the routine blocks until the motor operation is completed, e.g. moving of the motor, it means the routine will block until the execution of move command is completed. All API routines throw exception on error with message describing the problem. Driver uses 2 configuration files. For more on configuration see Configuration files.

3.1 API

The following is proposed API for the driver which supports both single motor and pair of motors (slit) manipulation

```
    /**
     * Sets mode to remote. Changing mode to remote is allowed only by local
control.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if operation not allowed  (if call is performed
from remote mode)
     */
    void setRemote() throw(MBoxException);
    /**
     * Sets mode to local control. Changing mode to local control is allowed
only by local control.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if operation not allowed (if call is performed
from remote mode)
     */
    void setLocalControl() throw(MBoxException);
    /**
     * Sets mode to local configuration. Changing mode to local configuration
is allowed only by local control.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if operation not allowed (if call is performed
from remote mode)
     */
    void setLocalConfiguration() throw(MBoxException);
    /**
     * Reads mode.
     *
     * @returns true if the mode is local control.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     */
    bool isLocalControl() throw(MBoxException);
    /**
     * Reads mode.
     *
     * @returns true if the mode is local configuration.
```

```
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     */
    bool isLocalConfiguration() throw(MBoxException);
    /**
     * Reads mode.
     *
     * @returns true if the mode is remote.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     */
    bool isRemote() throw(MBoxException);
    /**
     * Reads version of this driver.
     *
     * @returns version of this driver represented as string.
     */
    string version();
    /**
     * Reads motor status. For details on individual motor status bits see
Software Reference Manual: Turbo
     * PMAC/PMAC2 3Ax-01.937-xSxx, May 24, 2004, p.p. 325
     *
     * @param name name of the motor
     * @param status1 first part of motor status
     * @param status2 second part of motor status
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if named motor not configured
     * @throws MBoxException if internal driver error occurs
     * @throws MBoxException if error in communication with PMAC
     * @throws MBoxException if PMAC responds with error code
     */
    void readMotorStatus(const string name, unsigned int& status1, unsigned
int& status2) throw(MBoxException);
    /**
     * Reads pair status (coordinate system status). For details on coordinate
system status bits see Software Reference Manual: Turbo
```

```
     * PMAC/PMAC2 3Ax-01.937-xSxx, May 24, 2004, p.p. 329
     *
     * @param name name of the motor
     * @param status1 first part of pair status
     * @param status2 second part of pair status
     * @param status3 second part of pair status
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if named pair of motor not configured
     * @throws MBoxException if internal driver error occurs
     * @throws MBoxException if error in communication with PMAC
     * @throws MBoxException if PMAC responds with error code
     */
    void readPairStatus(const string name, unsigned int& status1, unsigned int&
status2, unsigned int& status3) throw(MBoxException);
    /**
     * Reads PMAC status. For details on PMAC status bits see Software
Reference Manual: Turbo
     * PMAC/PMAC2 3Ax-01.937-xSxx, May 24, 2004, p.p. 334
     *
     * @param[in] status1 first part of PMAC status
     * @param[out] status2 second part of PMAC status
     * @param[out] status3 second part of PMAC status
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if internal driver error occurs
     * @throws MBoxException if error in communication with PMAC
     * @throws MBoxException if PMAC responds with error code
     */
    void readPmacStatus(unsigned int& status1, unsigned int& status2, unsigned
int& status3) throw(MBoxException);
    /**
     * Moves pair of motors. Motors are moved to set positions.
     *
     * @param[in] name name of the motor pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
```

```
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code
     */
    void movePair(const string name) throw(MBoxException);
    /**
     * Moves pair of motors to positive or negative limit (soft limit is used
as positive or negative limit).
     *
     * @param[in] name name of the motor pair
     * @param[in] positiveLimit1 flag to indicate the movement to positive or
negative limit. If true positive limit is used for motor 1.
     * @param[in] positiveLimit2 flag to indicate the movement to positive or
negative limit. If true positive limit is used for motor 2.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code
     */
    void movePairToLimit(const string name, const bool positiveLimit1, const
bool positiveLimit2) throw(MBoxException);
    /**
     * Stops pair of motors.
     *
     * @param[in] name name of the motor pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code
     *
     */
    void stopPair(const string name) throw(MBoxException);
```

```
    /**
     * Kills pair. De-activates pair of motors and applies brake.
     *
     * @param[in] name name of the motor pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code
     */
    void killPair(const string name) throw(MBoxException);
    /**
     * Set motor pair center in mm.
     *
     * @param[in] name name of the motor pair.
     * @param[in] center new center to be set.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code
     */
    void setPairCenter(const string name, const float center)
throw(MBoxException);
    /**
     * Set motor pair gap in mm.
     *
     * @param[in] name name of the motor pair.
     * @param[in] gap new gap to be set.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
```

```
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code
     */
    void setPairGap(const string name, float gap) throw(MBoxException);
    /**
     * Reads motor pair set center in mm.
     *
     * @param[in] name name of the motor pair.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @return set motor pair center in mm.
     */
    float readPairSetCenter(const string name) throw(MBoxException);
    /**
     * Reads motor pair set gap in mm.
     *
     * @param[in] name name of the motor pair.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @return set motor pair gap in mm.
     */
    float readPairSetGap(const string name) throw(MBoxException);
    /**
     * Reads motor pair actual center in mm.
     *
     * @param[in] name name of the motor pair.
     *
```

```
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair or motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @return actual motor pair center in mm.
     */
    float readPairCenter(const string name) throw(MBoxException);
    /**
     * Reads motor pair actual gap in mm.
     *
     * @param[in] name name of the motor pair.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair of motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @return actual motor pair gap in mm.
     */
    float readPairGap(const string name) throw(MBoxException);
    /**
     * Moves motor pair center relative in mm.
     *
     * @param[in] name name of the motor pair.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair of motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void movePairCenterRelative(const string name, const float units)
throw(MBoxException);
    /**
```

```
     * Moves motor pair gap relative in mm.
     *
     * @param[in] name name of the motor pair.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair of motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void movePairGapRelative(const string name, float units)
throw(MBoxException);
    /**
     * Resets motor pair set and actual PMAC position in mm.
     *
     * @param[in] name name name of the motor pair
     * @param[in] position1 position for first motor of pair
     * @param[in] position2 position of second motor of pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair of motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void resetPairPosition(const string name, const float position1, const
float position2) throw(MBoxException);
    /**
     * Enables/disables middle switch for motor pair. This is used to determine
if hardware middle switch is present or not. If it is not
     * present then software middle distance check is enabled. This can only be
called from local configuration mode.
     *
     * @param[in] name name of the motor pair
     * @param[in] enabled if true hardware middle switch is present
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
```

```
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair of motors not configured.
     */
    void setPairMiddleSwitchEnabled(const string name, const bool enabled)
throw(MBoxException);
    /**
     * Reads if middle switch is enabled/disabled.
     *
     * @param[in] name name of the motor pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair of motors not configured.
     *
     * @returns true if hardware middle switch is enabled for given pair of
motors, false otherwise. If not enabled and pair of motors
     * is configured then software middle distance check is enabled.
     */
    bool readPairMiddleSwitchEnabled(const string name) throw(MBoxException);
    /**
     * Enables/disables motor pair. If motor pair is disabled manipulation of
motor pair is not allowed.
     * This can only be called from local configuration mode.
     *
     * @param[in] name name of the motor pair
     * @param[in] enabled if true motor pair is enabled.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair of motors not configured.
     */
    void setPairEnabled(const string name, const bool enabled)
throw(MBoxException);
    /**
     * Reads if motor pair is enabled/disabled. If motor pair is disabled
manipulation of motor pair is not allowed.
     *
     * @param[in] name name of the motor pair
     *
```

```
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair of motors not configured.
     *
     * @returns true if motor pair is enabled, false otherwise.
     */
    bool readPairEnabled(const string name) throw(MBoxException);
    /**
     * Sets polarity for middle switch for motor pair. This can only be called
from local configuration mode.
     * This is only used if hardware middle switch is enabled.
     *
     * @param[in] name name of the motor pair
     * @param[in] polarity if true middle switch is high, low otherwise
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named pair of motors not configured.
     */
    void setPairMiddleSwitchPolarity(const string name, const bool polarity)
throw(MBoxException);
    /**
     * Reads middle switch polarity. This is only used if hardware middle
switch is enabled.
     *
     * @param[in] name name of the motor pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair of motors not configured.
     *
     * @returns true if middle switch polarity is high , false otherwise.
     */
    bool readPairMiddleSwitchPolarity(const string name) throw(MBoxException);
    /**
     * Sets pair minimum spacing in mm. This is only used if software middle
distance check is enabled.
     *
```

```
     *
     * @param[in] name name of the motor pair
     * @param[in] spacing spacing in mm
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed (if call is performed
from remote when mode is set to one of the two local modes).
     * @throws MBoxException if named pair of motors not configured.
     */
    void setPairMinimumSpacing(const string name, const float spacing)
throw(MBoxException);
    /**
     * Reads if minimum spacing in motor counts. This is only used if software
middle distance check is enabled.
     *
     * @param[in] name name of the motor pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named pair of motors not configured.
     *
     * @returns minimum spacing for given motor pair in mm.
     */
    float readPairMinimumSpacing(const string name) throw(MBoxException);
    /**
     * Reads the status if minimum spacing was violated and hence the motors
stopped.
     *
     * @param[in] name name of the motor pair
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if named pair of motors not configured
     * @throws MBoxException if error in communication with PMAC
     * @throws MBoxException if PMAC responds with error code
     *
     * @returns minimum spacing for given motor pair in motor counts
     */
    bool readPairMinimumSpacingStatus(const string name) throw(MBoxException);
```

```
    /**
     * Returns names of the motors configured for a given motor pair.
     *
     * @param[in] name name of the motor pair
     * @param[out] motorName1 name of the first motor
     * @param[out] motorName2 name of the second motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if named pair of motors not configured
     */
    void getPairMotors(const string name, string& motorName1, string&
motorName2) throw(MBoxException);
    /**
     * Stops all motors controlled by M-Box. To stop a particular pair of
motors use  #stopPair and to stop only one motor use #stopMotor.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if operation not allowed (if call is performed
from remote when mode is set to one of the two local modes)
     * @throws MBoxException if error in communication with PMAC
     * @throws MBoxException if PMAC responds with error code
     */
    void stopAllMotors() throw(MBoxException);
    /**
     * Moves one motor. Motor is moved to set position. See
#setMotorSetPosition for details on setting motor set position.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out
     * @throws MBoxException if operation not allowed
     * @throws MBoxException if named motor not configured
     * @throws MBoxException if internal driver error occurs
     * @throws MBoxException if error in communication with PMAC
     * @throws MBoxException if PMAC responds with error code
     */
    void moveMotor(const string name) throw(MBoxException);
    /**
```

```
     * Moves one motor relative for motor counts. Move is done relative to PMAC
internal readback.
     *
     * @param[in] name name of the motor
     * @param[in] counts number of counts to move
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void moveMotorRelative(const string name, const float units)
throw(MBoxException);
    /**
     * Moves motor to positive or negative limit. (soft limit is used as
positive or negative limit).
     *
     * @param[in] name name of the motor.
     * @param[in] positiveLimit flag to indicate the movement to positive or
negative limit. If true positive limit is used.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void moveMotorToLimit(const string name, const bool positiveLimit)
throw(MBoxException);
    /**
     * Stops motor.
     *
     * @param[in] name name of the motor.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
```

```
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void stopMotor(const string name) throw(MBoxException);
    /**
     * Stops motor. Deactivates the motor and applies brake.
     *
     * @param[in] name name of the motor.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void killMotor(const string name) throw(MBoxException);
    /**
     * Set motor set position in mm.
     *
     * @param[in] name name of the motor.
     * @param[in] position position
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void setMotorSetPosition(const string name,const float position)
throw(MBoxException);
    /**
     * Reads motor set position in mm.
     *
     * @param[in] name name of the motor.
     *
```

```
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns motor set position in mm.
     */
    float readMotorSetPosition(const string name) throw(MBoxException);
    /**
     * Reads motor PMAC actual position in mm.
     *
     * @param[in] name name of the motor.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns motor PMAC actual position in mm.
     */
    float readMotorPmacActualPosition(const string name) throw(MBoxException);
    /**
     * Reads motor actual position in mm.
     *
     * @param[in] name name of the motor.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns motor actual position in mm.
     */
```

```
   float readMotorActualPositionInEGU(const string name) throw(MBoxException);
   /**
    * Reads motor potentiometer position in volts.
    *
    * @param[in] name name of the motor.
    *
    * @throws MBoxException if shared memory not initialized or locking the
shared memory times out.
    * @throws MBoxException if named motor not configured.
    * @throws MBoxException if other internal driver error occurs.
    * @throws MBoxException if error in communication with potentiometer.
    *
    * @returns motor potentiometer position in volts.
    */
   float readMotorPotentiometerPosition(const string name)
throw(MBoxException);
   /**
    * Reads motor potentiometer reference in volts.
    *
    * @param[in] name name of the motor.
    *
    * @throws MBoxException if shared memory not initialized or locking the
shared memory times out.
    * @throws MBoxException if named motor not configured.
    * @throws MBoxException if other internal driver error occurs.
    * @throws MBoxException if error in communication with potentiometer.
    *
    * @returns motor potentiometer reference in volts.
    */
   float readMotorPotentiometerReference(const string name)
throw(MBoxException);
   /**
    * Reads motor position from SSI encoder.
    *
    * @param[in] name name of the motor p
    *
    * @throws MBoxException if shared memory not initialized or locking the
shared memory times out.
    * @throws MBoxException if named motor not configured.
```

```
     * @throws MBoxException if other internal driver error occurs.
     * @throws MBoxException if error in communication with SSI encoder.
     *
     * @returns motor positon from SSI encoder.
     */
    float readMotorSSIPosition(const string name) throw(MBoxException);
    /**
     * Resets motor actual PMAC position in motor counts.
     *
     * @param[in] name name of the motor
     * @param[in] position position to reset to.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void resetMotorPosition(const string name, const float position)
throw(MBoxException);
    /**
     * Set motor velocity in mm/sec for one motor.
     *
     * @param[in] name name of the motor
     * @param[in] velocity velocity for motor in mm/sec
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void setMotorVelocity(const string name, const float velocity)
throw(MBoxException);
    /**
```

```
     * Reads motor velocity in mm/sec.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns velocity in mm/sec
     */
    float readMotorVelocity(const string name) throw(MBoxException);
    /**
     * Set motor acceleration time in sec.
     *
     * @param[in] name name of the motor
     * @param[in] accelerationTime acceleration time for motor in sec
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void setMotorAccelerationTime(const string name, const float
accelerationTime) throw(MBoxException);
    /**
     * Reads motor acceleration time in sec.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
```

```
     *
     * @returns motor acceleration time in sec.
     */
    float readMotorAccelerationTime(const string name) throw(MBoxException);
    /**
     * Set motor drive direction.
     *
     * @param[in] name name of the motor
     * @param[in] driveDirection motor drive direction. True means positive,
false means negative.
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void setMotorDriveDirection(const string name, bool driveDirection)
throw(MBoxException);
    /**
     * Reads motor drive direction.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns true if motor driver direction is clockwise, false otherwise.
     */
    bool readMotorDriveDirection(const string name) throw(MBoxException);
    /**
     * Set motor pulse width in usec. Note, due to how PMAC handles pulse
width, this setting is applied for groups of 4 motors only.
```

```
     * E.g. if it is changed for one motor in group 1-4 it will be applied for
all 4 motors. It is similar for motors 5-8.
     *
     * @param[in] name name of the motor
     * @param[in] pulseWidth pulse width
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void setMotorPulseWidth(const string name, const float pulseWidth)
throw(MBoxException);
    /**
     * Reads motor pulse width in usec. Note, due to how PMAC handles pulse
width, this setting is valid for groups of 4 motors only.
     * E.g. if it is changed for one motor in group 1-4 it will be applied for
all 4 motors. It is similar for motors 5-8.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns pulse width in usec.
     */
    float readMotorPulseWidth(const string name) throw(MBoxException);
    /**
     * Set motor polarity.
     *
     * @param[in] name name of the motor
     * @param[in] pulsePolarity pulse polarity
     *
```

```
     * @throws AccessControlException if shared memory error occurs or locking
the shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void setMotorPulsePolarity(const string name, const bool pulsePolarity)
throw(MBoxException);
    /**
     * Reads motor pulse polarity.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns pulse polarity.
     */
    bool readMotorPulsePolarity(const string name) throw(MBoxException);
    /**
     * Set motor limit switch enabled. If limit switch are enabled, PMAC will
stop motors on soft limits.
     *
     * @param[in] name name of the motor
     * @param[in] enabled true if limit switch enabled, false otherwise
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if operation not allowed.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
```

```
   void setMotorLimitSwitchEnabled(const string name, const bool enabled)
throw(MBoxException);
   /**
    * Reads motor limit switch enabled/disabled flag.
    *
    * @param[in] name name of the motor
    *
    * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
    * @throws MBoxException if named motor not configured.
    * @throws MBoxException if internal driver error occurs.
    * @throws MBoxException if error in communication with PMAC.
    * @throws MBoxException if PMAC responds with error code.
    *
    * @returns true if motor limit switch is enabled, false otherwise
    */
   bool readMotorLimitSwitchEnabled(const string name) throw(MBoxException);
   /**
    * Set motor minimum position in motor counts.
    *
    * @param[in] name name of the motor
    * @param[in] min minimum position is motor counts
    *
    * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
    * @throws MBoxException if named motor not configured.
    * @throws MBoxException if internal driver error occurs.
    * @throws MBoxException if error in communication with PMAC.
    * @throws MBoxException if PMAC responds with error code.
    */
   void setMotorMinPosition(const string name, const float min)
throw(MBoxException);
   /**
    * Set motor maximum position in mm.
    *
    * @param[in] name name of the motor
    * @param[in] max maximum position is mm
    *
    * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
```

```
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     */
    void setMotorMaxPosition(const string name, const float max)
throw(MBoxException);
    /**
     * Reads motor minimum position in mm.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns minimum position in mm.
     */
    float readMotorMinPosition(const string name) throw(MBoxException);
    /**
     * Reads motor maximum position in mm.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns maximum position in mm.
     */
    float readMotorMaxPosition(const string name) throw(MBoxException);
    /**
     * Reads motor break status.
```

```
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns motor brake status. True means brake is off. False means break
is on.
     */
    bool readMotorBreakStatus(const string name) throw(MBoxException);
    /**
     * Reads motor overheat status.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns motor overheat status. True when overheat, false otherwise.
     */
    bool readMotorOverheatStatus(const string name) throw(MBoxException);
    /**
     * Reads motor interlock status.
     *
     * @param[in] name name of the motor
     *
     * @throws MBoxException if shared memory error occurs or locking the
shared memory times out.
     * @throws MBoxException if named motor not configured.
     * @throws MBoxException if internal driver error occurs.
     * @throws MBoxException if error in communication with PMAC.
     * @throws MBoxException if PMAC responds with error code.
     *
     * @returns motor interlock status. True when interlock, false otherwise.
     */
    bool readMotorAxisIntStatus(const string name) throw(MBoxException);
```

3.2 ACCESS CONTROL

Since more than one process is used to control motors on M-Box access control has to be implemented. The job of the access control is to:

- prevent concurrent access
- grant access rights for certain operation depending on the nature of the caller

The following processes will use be used to manipulate motors on M-Box:

- FESA class
- Local control server
- M-Box front panel LCD application

Access control is implemented using shared memory. A single object is created in shared memory which holds the following information:

- current mode which can be one of:
  - Remote control, used by FESA class
  - Local control, used by local control server and LCD application, FESA class has read only access
  - Local configuration, used by local control to modify configuration, LCD application and FESA class have read only access
- state of the semaphore used to prevent concurrent access to the low level drivers (Ethernet, serial)
- locked counter, used to clean up shared memory in case 'user' of the shared memory object crashed. See Access control monitor for details.
- Heart beat counter, used to implement fallback from local control mode to remote mode (used if local control mode operator forgets to change the mode). Local configuration mode does not have a fallback mode as this can be potentiality dangerous if the configuration of the motors was changed but not verified.

4 LOCAL CONTROL SERVER AND GUI

Maintenance persons need separate maintenance access. The access is needed to:

- Configure motor parameters
- Have access to all diagnostic data
- Operate individual motors
- Operate pair of motors

Maintenance access to devices installed in the accelerator will be needed:

- In case of trouble shooting (problems in accelerator operation)
- In case of reconfiguring in shutdown periods

Maintenance access is independent of control system infrastructure (control system middleware, central data bases, ...) and is possible via direct network connection to M-Box.

Local control consists of two parts as shown on figure below:

- Local control server
  - C++ Ethernet socket server implementation which uses system driver to manipulate motors.
- Local control GUI
  - Java GUI which can be run from remote computer. It communicates with local control server via Ethernet. Described in details below.

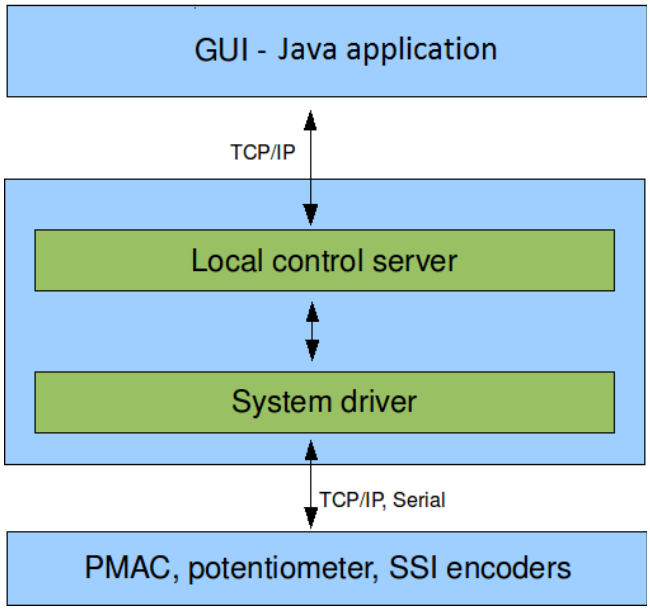

Figure 1: Layout of the local control

4.1 LOCAL CONTROL GUI

Local control GUI is basically divided into two main screens:

- Motor and motor pair configuration screen
- Motor and motor pair control screen

Motor and motor pair configuration screen can be seen on the figure below.

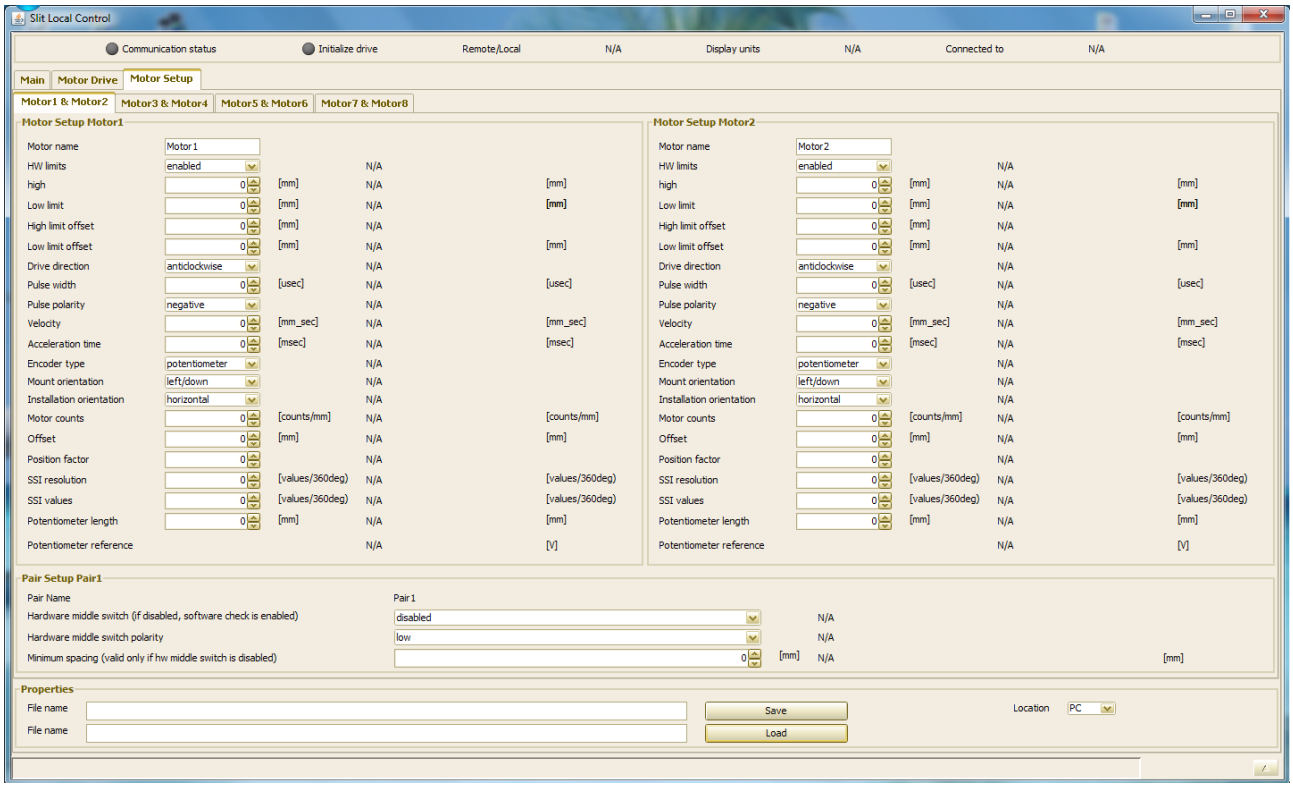

Figure 6: Motor and motor pair configuration screen

Motor and motor pair configuration screen offers modification of the following parameters for each motor :

- nomenclature of the motor
- mounting orientation
- part of motor pair or not
- horizontal or vertical installation
- encoder type selection
- middle switch enabled or disabled
- limit switch enabled/disabled
- set driver direction CW/CCW (linked to PMAC variable I7mn8, see [3])

- driving frequency (velocity)
- pulse width (linked to PMAC variables I7m03 and I7m04, see [3])
- pulse polarity (linked to PMAC variable I7mn7, see [3])
- acceleration time
- engineering unit to steps conversion (EGU/encoder steps)
- offset
- positive/negative limits
- position factor
- minimum spacing if middle switch disabled

Configuration screen also offers properties to be saved/load to/from files on the PC where local control GUI is running or directly to/from M-Box.

Motor and motor pair control screen is shown on the figure below:

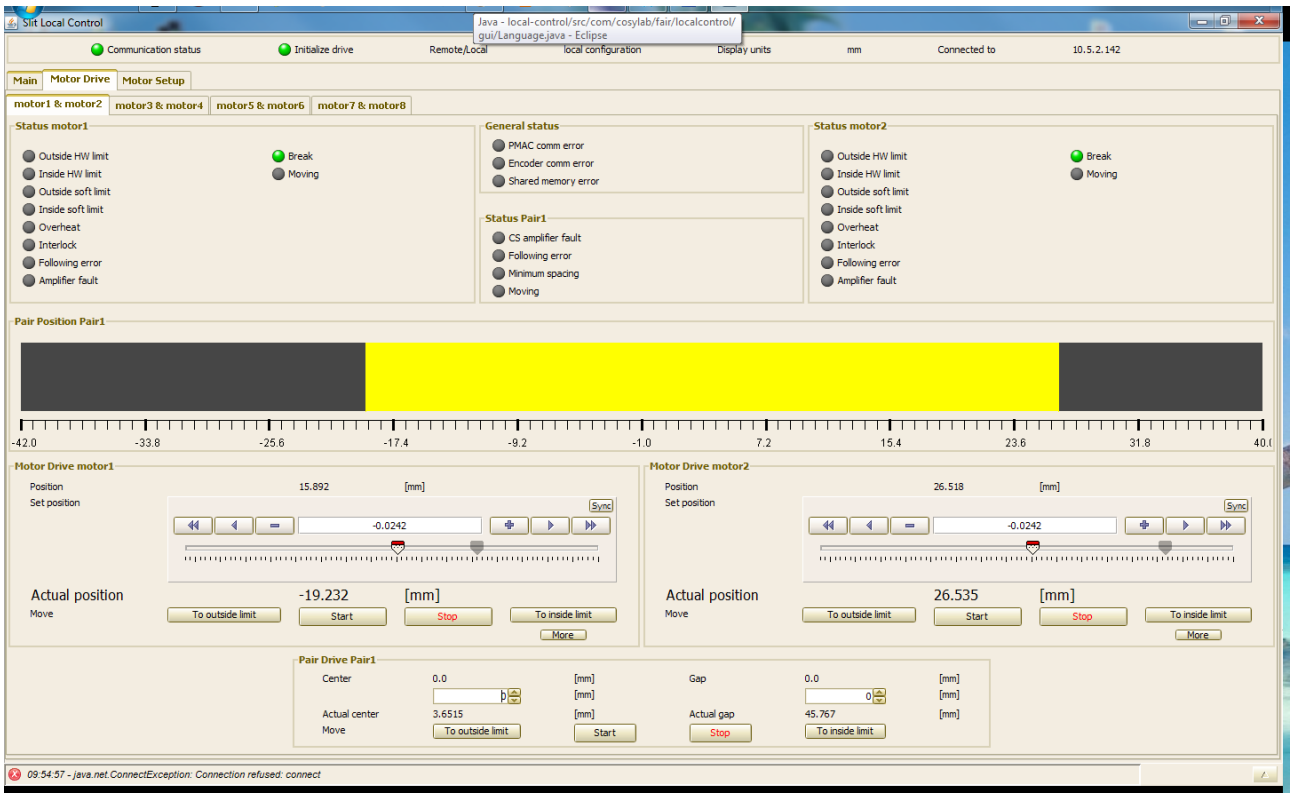

Figure 10: Motor and motor pair control screen

As shown on the figure above motor and motor pair screen is divided into two sections. Upper side of the screen comprises:

- Motor and motor pair status

- General status
- Graphical presentation of motor positions

Lower part comprises motor and motor pair controls.

5 ACCESS CONTROL MONITOR

Access control monitor is an application which resides on M-Box and monitors the state of the shared memory used for access control. It performs the following tasks:

- monitors the lock counter of shared object. If object is locked for 10 seconds and the counter does not increase in this time (which would indicate that process locking the object crashed) then object is unlocked so that it can be used by other processes.
- Monitors the heart beat counter of local control mode. When local control is active it will increase this counter. If the counter will not increase for a certain time the monitor application will change the mode to remote control. The time-out for this callback to occur is configurable via configuration file.

Code base for this application is part of the system driver described in chapter 3.

6 FESA DEVICE SERVER

FESA class mirrors the functionality supported in the system driver i.e. FESA class offers control of single or pair of motors.

6.1 GENERAL DESIGN

6.1.1 Class composition

Two FESA classes are provided. A FESA class to manipulate a single motor and a FESA class to manipulate pair of motors. FESA composition mechanism is used to deploy them as single deployment unit.

6.1.2 Data Types

Floating point values are of float type. Step unit values are of 32-bit integer type. Information property items are numeric for numerical items and strings for the rest.

6.1.3 Data Synchronization and Notifications

FESA class stores all the set-point data and the read-back data received from system driver.

Since the system driver does not support callbacks for notifications of data change, and FESA properties interface assumes that properties can be monitored for changes, the notification mechanism is implemented in the FESA class.

Notification mechanism uses polling of the data on the system driver at regular intervals. It is assumed that FESA does not handle very frequent updates well, so the polling is expected to be done at a 1s interval. Due to the low update rate, there is no distinction between properties and all are updated at this same interval.

To use an already existing FESA functionality, a custom software event generator is included in the RT part of the FESA class design to perform the notifications. On the event, all data is read from the system driver and compared to stored data. On a change, the relevant properties are notified, and the new values are stored.

On the client property get request, the values are read from the system driver instead of from the stored values. This is done with the assumption that the reads on the system driver are not a bottleneck.

On the client property set request, the value is set on the system driver regardless if the value is the same as old value, and no checking with re-reading is performed. This assumes that the system driver throws an exception in every situation in which the value could not be set exactly.

6.1.4 Property Coupling

A change in a property can trigger an immediate or delayed change of a status property. When a change of status is unconditional due to logic dependency, the status property is always notified. When a change of status happens only in the case of an error, the status property is notified when an exception is thrown. Delayed status changes are ignored as they are handled by periodic polling notifications.

6.1.5 Error Handling

FESA class checks the error state and catches exceptions generated by the system driver, and re-throws them as FESA exceptions.

All writable parameters whose valid range is not checked in the system driver are checked in the FESA class, and appropriate exceptions are thrown.

6.1.6 Initialization

During FESA class initialization, the system driver is first initialized from configuration files and then property values are set from the driver. All changed properties are notified. For motor parameters initialization FESA instantiation XML files are not used. These files are used only for FESA specific initialization.

6.1.7 Access control

As described in chapter Access control access control is implemented on the system driver level. When system is in use by local control FESA class will be denied any manipulation of the motors (e.g. change of setpoints, moving the motors etc.), only reading from the system driver is possible.

6.2 CLASS PROPERTIES

6.2.1 Motor class

Property name	***Property type***	***Value Items name***	***Value Item type***	***unit***	***description***
Status	status				
		mode-item	GSI-mode-field		Always on
		control-item	GSI-control-field	/	REMOTE/LOCAL CONTROL/ LOCAL CONFIG
		status-item	GSI-status-field	/	UNKNOWN/OK/ WARNING/ERRO

					R
		detailed-status-field	GSI-detailed-status-field	/	0 - Pmac comm error 1 - encoder comm error 2 – shared memory error 3 – internal error
		motorStatus	bitset	/	0 – inner hw end limit set 1 – outer hw end limit set 2 – inner sw end limit set 3 – outer sw end limit set 4 – moving 5 – fatal following error 6 - amplifier fault 7 - overheat 8 – axis interlock 9 – break 10 - potentiometer reference error 11- middle switch (only used if motor is paired)
Power	power				Always on. Returns error if try to switch it off.
Init	init				Initialize (from XML and configuration files) Motor is not moved. Set position and readback are consistent.
Reset	reset				Same as init.
Version	version				Version of hw, firmware, driver, fesa class, fesa framework
Setting	setting				

		position	float	meter	Absolute position setpoint
Acquisition	acquisition				
		position	float	meter	Absolute position readback
		position_status	AQN_STATUS	/	Set to DIFFERENT_FROM_SETTING when deviation from set value greater position_tolAbs
		position_tolAbs	float	meter	Absolute tolerance of deviation
		setPosition	float	meter	Absolute position setpoint
		resolution	float	meter	Resolution precision of readback
		motorStatus	bitset	/	0 – inner hw end limit set 1 – outer hw end limit set 2 – inner sw end limit set 3 – outer sw end limit set 4 – moving 5 – fatal following error 6 - amplifier fault 7 - overheat 8 – axis interlock 9 – break 10 - potentiometer reference error 11- middle switch (only used if motor is paired)
PositionRelative	setting	positionVariation	float	meter	Move relative to current value.
MoveSteps	setting	steps	int	/	Stepwise relative movement
ToEndPosition	setting	endPosition	int	0/1	move to inner/outer end position (0: out,

					1: in)
StopMotor	command	/	/	/	/
Configuration	acquisition				
		encoderType	string list	potentiometer ssi	
		mounting	string list	left/down right/up	
		installation	string list	vertical horizontal	
		partOfPair	string list	yes no	
		settingResolution	float	meter	minimum possible movement (1 step)
		maxPosition	float	meter	
		minPosition	float	meter	
		offset	float	meter	
		positionFactor	float	/	
		driveDirection	string list	clockwise anticlockwise	
		pulseWidth	float	microseconds	
		pulsePolarity	string list	positive negative	
		accelerationTime	float	seconds	
		limitsEnabled	string list	yes no	
		potentiometerLength	float	meter	
		velocity	float	meter/s	
		ssiTurnResolution	integer	counts	Number of counts per one encoder turn.
		ssiResolution	float	meter	Resolution of ssi encoder
		referenceVoltageLimit	float	Volt	Maximum allowed difference for referenceVoltage

Diagnostics	acquisition				
		referenceVoltage	float	Volt	Reference voltage
		positionFactor	float	/	
		offset	float	meter	
		positionVoltage	float	Volt	Raw position value received from potentiometer
		positionSSI	integer	counts	Raw position value received from potentiometer

6.2.2 Slit class

Property name	***Property type***	***Field name***	***Field type***	***unit***	***description***
Status	status				
		mode-item	GSI-mode-field		Always on
		control-item	GSI-control-field	/	REMOTE/LOCAL CONTROL/ LOCAL CONFIG
		status-item	GSI-status-field	/	UNKNOWN/OK/WARNING/ERROR
		detailed-status-field	GSI-detailed-status-field	/	0 - Pmac comm error 1 - encoder comm error 2 – shared memory error 3 – internal error
		motor1Status	bitset	/	0 – inner hw end limit set 1 – outer hw end limit set 2 – inner sw end limit set 3 – outer sw end limit set 4 – moving 5 – fatal following error

					6 - amplifier fault 7 - overheat 8 – axis interlock 9 – break 10 - potentiometer reference error 11- middle switch (only used if motor is paired)
		motor2Status	bitset	/	0 – inner hw end limit set 1 – outer hw end limit set 2 – inner sw end limit set 3 – outer sw end limit set 4 – moving 5 – fatal following error 6 - amplifier fault 7 - overheat 8 – axis interlock 9 – break 10 - potentiometer reference error 11- middle switch (only used if motor is paired)
		slitsStatus	bitSet	/	0 – middle switch status
Power	power				Always on. Error if try to switch it off.
Init	init				Initialize (from XML and configuration files) Motor is not moved. Set position and readback are consistent.
Reset	reset				Same as init.
Version	version				Version of hw, firmware, driver,

					fesa class, fesa framework
Setting	setting				
		center	float	meter	Slit center
		width	float	meter	Slit width
Acquisition	acquisition				
		center	float	meter	Actual center
		width	float	meter	Actual width
		center_status	AQN_STATUS	/	Set to DIFFERENT_FROM_SETTING when deviation from set value greater position_tolAbs
		center_tolAbs	float	meter	Absolute tolerance of deviation
		width_status	AQN_STATUS	/	Set to DIFFERENT_FROM_SETTING when deviation from set value greater position_tolAbs
		width_tolAbs	float	meter	Absolute tolerance of deviation
		setCenter	float	meter	Center setpoint
		setWidth	float	meter	Width setpoint
		resolution1	float	meter	Resolution precision of readback
		resolution2	float	meter	Resolution precision of readback
		motor1Status	bitset	/	0 – inner hw end limit set 1 – outer hw end limit set 2 – inner sw end limit set 3 – outer sw end limit set 4 – moving 5 – fatal following

					error 6 - amplifier fault 7 - overheat 8 – axis interlock 9 – break 10 - potentiometer reference error 11- middle switch
		motor2Status	bitset	/	0 – inner hw end limit set 1 – outer hw end limit set 2 – inner sw end limit set 3 – outer sw end limit set 4 – moving 5 – fatal following error 6 - amplifier fault 7 - overheat 8 – axis interlock 9 – break 10 - potentiometer reference error 11- middle switch
		slitsStatus	bitSet	/	0 – middle switch status
CenterRelative	setting	centerVariation	float	meter	Move center relative to current set value
WidthRelative	setting	widthVariation	float	meter	Move width relative to current set value
ToEndPosition	setting	endPosition	int	0/1	move to inner/outer end position (0: out, 1: in)
StopMotor	command	/	/	/	Stops both motors
Configuration	read-only				
		encoderType1	string list	potentiometer ssi	
		mounting1	string list	left/down right/up	

		installation1	string list	vertical horizontal	
		partOfPair1	string list	yes no	
		settingResolution1	float	meter	minimum possible movement (1 step)
		maxPosition1	float	meter	
		minPosition1	float	meter	
		offset1	float	meter	
		positionFactor1	float	/	
		driveDirection1	string list	clockwise anticlockwise	
		pulseWidth1	float	microseconds	
		pulsePolarity1	string list	positive negative	
		accelerationTime1	float	seconds	
		limitsEnabled1	string list	yes no	
		potentiometerLength1	float	meter	
		velocity1	float	meter/s	
		ssiTurnResolution1	integer	counts	Number of counts per one encoder turn.
		ssiResolution1	float	meter	Resolution of ssi encoder
		referenceVoltageLimit1	float	Volt	Maximum allowed difference for referenceVoltage
		encoderType2	string list	potentiometer ssi	
		mounting2	string list	left/down right/up	
		installation2	string list	vertical horizontal	
		partOfPair2	string list	yes no	

		settingResolution2	float	meter	minimum possible movement (1 step)
		maxPosition2	float	meter	
		minPosition2	float	meter	
		offset2	float	meter	
		positionFactor2	float	/	
		driveDirection2	string list	clockwise anticlockwise	
		pulseWidth2	float	microseconds	
		pulsePolarity2	string list	positive negative	
		accelerationTime2	float	seconds	
		limitsEnabled2	string list	yes no	
		potentiometerLength2	float	meter	
		velocity2	float	meter/s	
		ssiTurnResolution2	integer	counts	Number of counts per one encoder turn.
		ssiResolution2	float	meter	Resolution of ssi encoder
		referenceVoltageLimit2	float	Volt	Maximum allowed difference for referenceVoltage
		middleSwitch	string list	enabled/disabled	If disabled then sw check is enabled
		minimumSpacing	float	meter	Minimum distance if sw middle switch enabled
		middleSwitchPolarity	string list	meter	Polarity for hw middle switch.
Diagnostics	acquisition				
		referenceVoltage1	float	Volt	Reference voltage
		positionFactor1	float	/	
		offset1	float	meter	

		positionSSI1	integer	counts	Raw position value received from potentiometer
		referenceVoltage 2	float	Volt	Reference voltage
		positionVoltage2	float	Volt	Raw position value received from potentiometer
		positionSSI2	integer	counts	Raw position value received from potentiometer
		positionFactor2	float	/	
		offset2	float	meter	

7 CONFIGURATION FILES

Systems uses several configuration files during initialization and for persistence. The following configuration files are used in the system:

- PMAC properties file
  - This file contains PMAC configuration setup and commands that needs to be executed to drive single motor or pair of motors, to stop single motor or pair of motors etc. This file needs to be changed only if basic system reconfiguration is done e.g. reconfiguration of motor channels. It is used by the system driver.
- Motor properties file
  - This file contains all motor settings that can be changed by the user (e.g. drive direction, default velocity, default acceleration, …) using local control GUI. This file is used by system driver during initialization or on user request via local control GUI.
- FESA device server instantiation file
  - This is standard FESA instantiation XML configuration file. It includes all initial values for parameters that are of specific use by FESA device server only.
- Startup files
  - These files are used for startup options (e.g. location of log files etc) for the local control server, LCD control and access control monitor.