Instrumentation Group

BioSAXS Sample Changer

Control Software

Author: Alexandre Gobbo

Revision: 0.1.1

Date: 10.20.2011

EMBL

Table of Contents

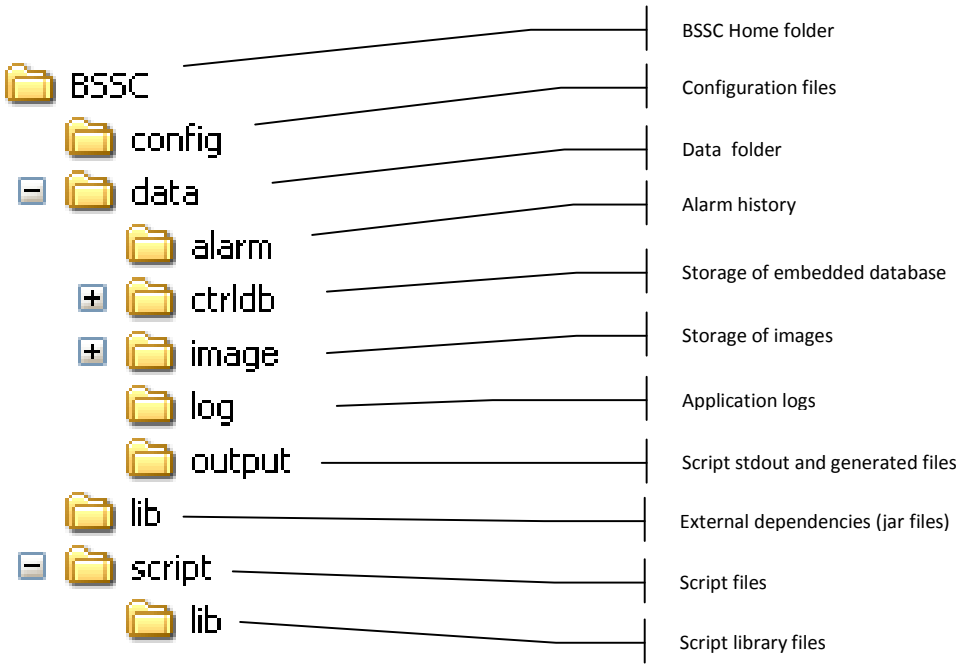
Requirements

1) Hardware Requirements
    - PC (x86-compatible), recommended minimum Intel Core2 Duo or AMD Athlon X2 CPU, 1Gb RAM and 20Gb free disk space.
    - Minimal screen resolution: 1024 x 768.

2) Software Requirements
    - Windows XP SP3.
    - Java 6 (JRE 6 Update 15 or higher).
    - Java Communications API. This can be installed by typing on the application folder:
        - java –jar serialsetup.jar

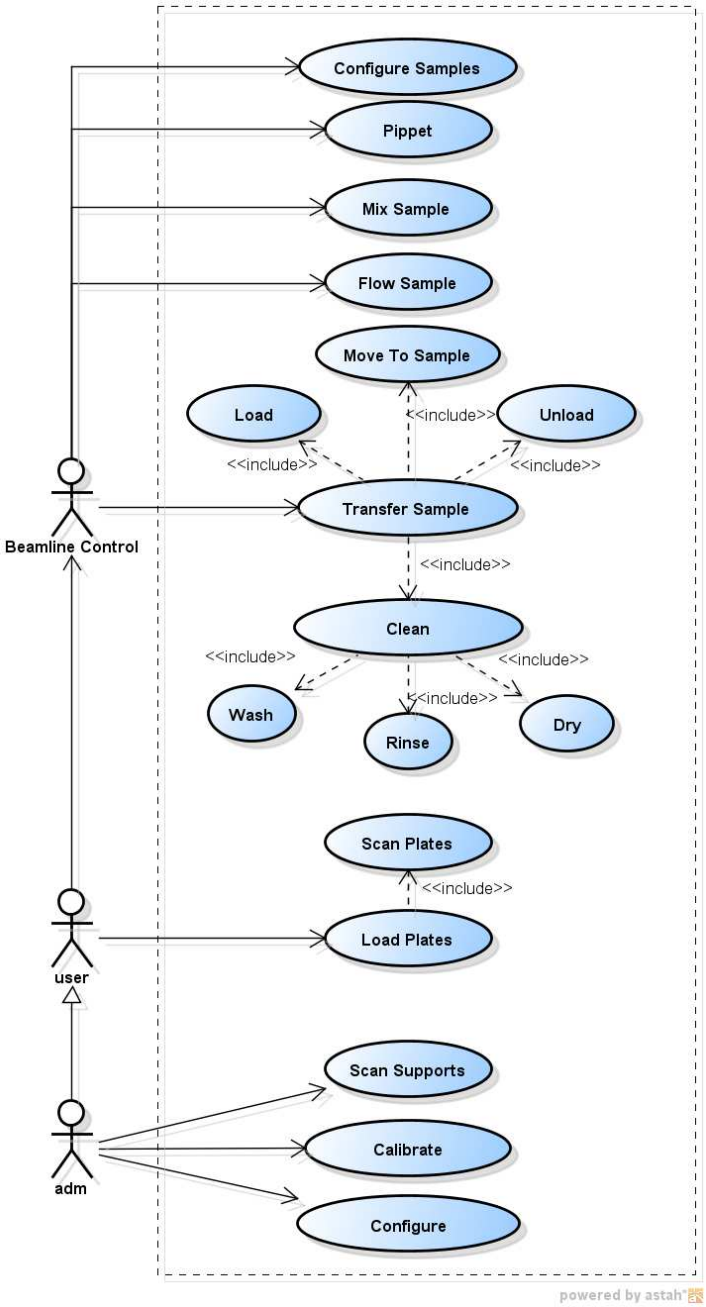
Installation

The distribution package contains the following structure:



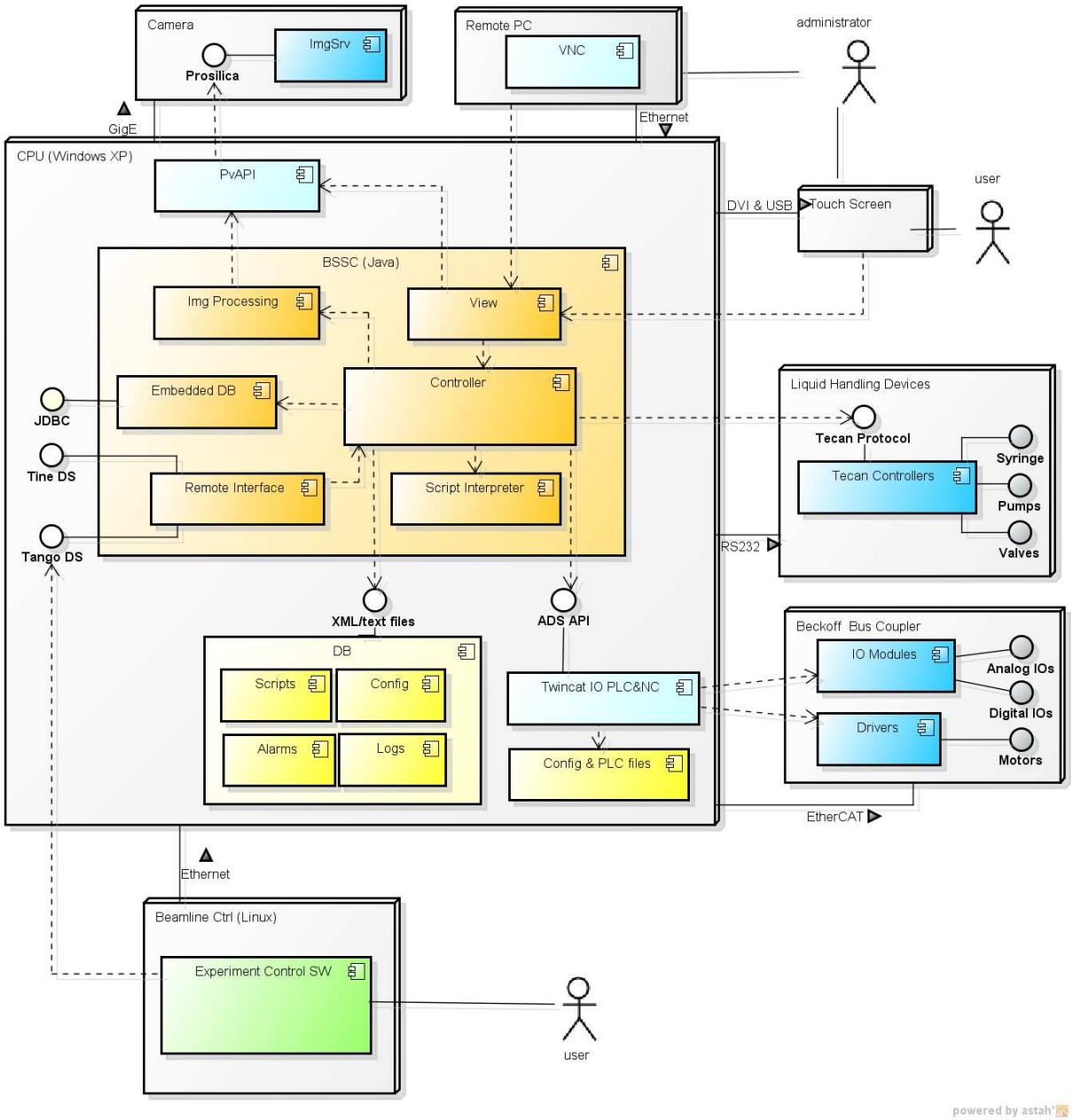
Use Cases

The main use cases of the Sample Changer control software are presented in the following diagram.



Deployment

The Sample Changer software (BSSC.jar) is an application that interacts with devices, users, and experiment control software. The main interactions and protocols are presented in the following deployment diagram.



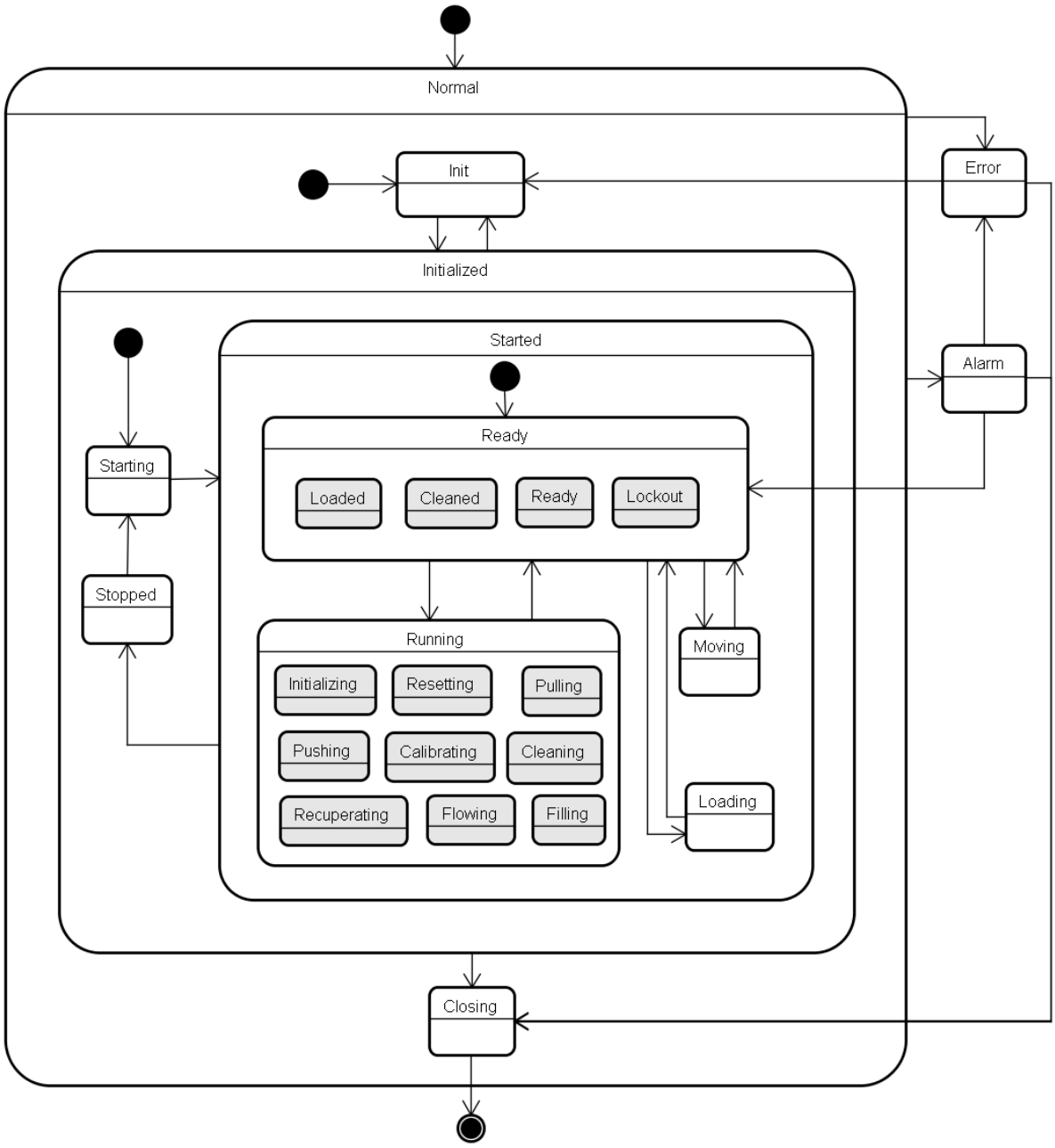
Startup

The application can be launched in different modes:

Mode	Command	Description
Headless	java -Djava.awt.headless=true -jar bssc.jar	No heavyweight graphical widget may be initialized. Console commands that create widgets are not allowed. Depending on the configuration – as the video grabbing library used – this can generate problems, since some used libraries may instantiate graphical components.
Server	java -jar bssc.jar -nogui	No graphical user interface is instantiated. Runs just like headless mode but can create graphical widgets responding to console commands (as "inspector")
Service	java -jar bssc.jar -service -hide	At first sight it is identical to Server mode but in fact the GUI is instantiated and hidden. The GUI is shown by typing (or sending to stdin) "show", and hidden by typing "hide". Intended to run as a Windows service, when a system tray icon can show/hide the application interface.
Normal	java -jar bssc.jar	Normal mode. The GUI instantiated and its reference given to the controller.

State Machine

The overall application state machine is represented by the UML Statechart below. The current application state is exported as the property “State”. The states “Ready” and “Running” can be detailed in sub-states. In order to simplify synchronization, these sub-states do not change the value of the property “State” but they change the “Status” property instead.



Remote Access

The software functionality is fully available remotely.

There are two remote interfaces:

- A socket interface to low-level software functionality. This is the machine interface seen by scripts. The access of this interface can be enabled in the software configuration dialog “Enable low level control”. This interface is used to debug scripts. It can be accessed in a simple way in Python with “sc.py”- file available in scripts/lib path - or by C++ or Java clients.

- An embedded server of the high-level functionality. This server is dynamically created exporting all needed functionality to the beamline control, hiding the sample changer specific details. The interface exported is a set of generic methods (containing a return type, a variable number of parameters and the possibility of raising exceptions). The server can be created using different protocols (or control systems). A strategy is done to map generic methods into each protocol if generic method call is not supported.

The server is configured in the software configuration dialog box. It can be:

- Tango
  Single parameter get/set methods are mapped into Tango Attributes.
  Single parameter methods are mapped into standard tango methods (that support exceptions).
  Since Tango does not support methods with multiple parameters, multiple parameter methods are exported as methods receiving an array of string as parameters and returning a string. The parameters are translated to/from the string using the most appropriate conversion.

- Tine
  Single parameter get-set methods are mapped into Tine Properties.
  Non get/set methods are exported as a string property with write/read access. The written value is a comma-separated textual representation of each parameter. The result is read back as a string from the property.
  Since Tine does not support exceptions, a “protocol” is defined to report exceptions to the client: after the method call, if the Tine last link error is “ask_error” (18) it indicates that an exception has been raised on the server and its textual representation can then be read by the Tine property “LastError”.

- Epics
  Single parameter get/set methods are mapped into PVs of the corresponding type.
  Non get-set methods are mapped as PV of type array of strings. The array contains textual representation of the parameters. The reading of the PV following writing the parameters contains the return value or, if the return signals an alarm, the textual representation of the exception.

- Exporter
  This is the EMBL protocol. All the server methods are available through client libraries in Python, Java or C++.

- Web Service
  The server can be generated as a web service. A full-feature interface that supports generic parameters and exceptions so no mapping is required.

The server has three types of exports:

- Synchronous methods:

  Short operations, executed synchronously (blocks the client). Execution should never last more than 3 seconds. If any exception is raised during the execution of the method it is raise directly to the client within the same call.

- Get/set methods (properties):

  Same as Synchronous methods:

- Asynchronous methods (tasks):

  Long operations executed asynchronously. Since execution can take a long time, these method calls cannot block the client. The method returns immediately (unless there is an exception when starting the task process). The return value of the task (or its exception) should be checked later according to the task synchronization method used.

Task Synchronization

There are two methods to retrieve the result of an asynchronous method:

1. Simple (optimistic, single-client)

   All asynchronous methods change the overall application state: tasks can only be started if the state is “Ready”, and, after starting, they change the state to “Running”.

   After starting the task, the client application should monitor the application state (on a timer, specific thread, or as an event handler). If the state is different from “Running” it means that the task has finished. Then the client should check the property “Command Exception”. If it is not empty, it holds the textual representation of an exception raised within the task. Otherwise the return value can be read from the property “CommandOutput”.

   Optionally the client can monitor the property LastTaskInfo to identify the end of execution and at the same time retrieve the task output.

2. Advanced (reliable, multi-client)

   The method above is very straightforward but it has limitations: a new task can be started before the previous output has been read. This is very unlikely in single-client scenarios, but in more complex cases one may need a more robust synchronization method. Furthermore, today all asynchronous methods are exclusive (only one is executed at a time) but in the future we may have multiple tasks executing in parallel, what would require them to be referenced individually.

   All asynchronous methods return one unique task id. With this task ID one can get the information of a specific task. The synchronization is independent of the overall application state. After starting the task, the client application should monitor the task completion:

   - One can call the method “isTaskRunning(int id)” till it is equals to false and the call the method “checkTaskResult(int id)” that returns the task output or throw an exception if one has taken place.
   - Optionally at any time one can call getTaskInfo(int id) that returns an array containing: task name, task flags, start timestamp, end timestamp (empty if still running), task output and task exception.

Server Interface

Overall Application State Properties

Property Name	Type	RW	Info
State	State	RO	Overall application state.
Status	String	RO	Application status string. Textual representation of state.
CommandOutput	String	RO	Holds the return value of the last asynchronous task executed.
CommandException	String	RO	Exception occurred during the last asynchronous task (if any).
LastTaskInfo	String	RO	Information about the last task executed (Same as getTaskInfo(-1))

Asynchronous Methods (Return values are the task id):

Method Signature	Info
void restart(boolean init_hardware)	Reinitialize the machine controller. init_hardware defines if a hardware initialization (or "cold start") should be performed.
int fill(int plate, int row, int col,double volume)	Fill a given volume of sample from a given well to the exposure cell.
int recuperate(int plate, int row, int col)	Unload the sample in the exposure cell to a given well.
int transfer(int from_plate, int from_row, int from_col, int to_plate, int to_row, int to_col,double volume)	Pipetting of a given volume from a well to another.
int mix(int plate, int row, int col,double vol, int cycles)	Mix the contents of the well a given number of cycles.
int measureConcentration(int plate, int row, int col)	Get the spectrometer readings for the sample in a give well.
int clean()	Complete system cleaning.
int dry(double dry_time)	Dry the sample path for a given time.
int flow(double volume,double time)	Flow a given volume of sample in a given time.
int flowAll(double time)	Flow the loaded volume of sample in a given time.
int push(double volume, double speed)	Push the loaded sample by a given volume in a given speed (ul/s)
int pull(double volume, double speed)	Pull the loaded sample by a given volume in a given speed (ul/s)
int calibrate()	Perform a full calibration of the sample path
int loadPlates()	Move the plate table to plate load position.
int scanAndPark()	Check the needle, sample overlap, plate codebars and park the table
int waitTemperatureSEU(double value)	Set the exposure cell temperature setpoint and wait for the readout
int waitTemperatureSample(double value)	Set the storage temperature setpoint and wait for the readout

Synchronous Methods:

Method Signature	Info
void abort()	Stop all devices and ongoing task.
boolean detectCapillary()	Find the capillary position on the image and change the image's ROI
void moveSyringeForward(double speed)	Start a continuous move of the syringe forward in a given speed.
void moveSyringeBackward(double speed)	Start a continuous move of the syringe backward in a given speed.
void stopSyringe()	Stop a syringe move
double[] getPlateInfo(int index)	Returns an array containing geometry information of the plate loaded in the position defined by "index": [rows, columns, deep columns]
double[] getPlateTypeInfo(String type)	Geometry information for a given plate name.

Atributes (get/set Methods)

Property Name	Type	RW	Info
PlatesIDs	String[]	RO	The scanned code bar for the three positions.
TemperatureSEU	double	RW	Exposure cell temperature.
TemperatureSampleStorage	double	RW	Sample storage temperature.
SpectrometerReadout	double	RO	Return the spectrometer reading for the last transferred sample.
SpectrometerDarkReadout	double	RO	Return the last spectrometer dark readout.
SpectrometerRealPathLenght	double	RO	The tube width in the spectrometer measurement location.
BeamLocation	String	RW	Beam location in the format “x1 y1 x2 y2”.
LiquidPositionFixed	boolean	RW	Automatic regulation of syringe to fix the liquid position on screen.
BeamShapeEllipse	boolean	WO	Beam shape format: ellipse or rectangle.
BeamMarkVolume	double	RO	The sample volume contained in the beam mark.
SampleType	String (enum)	RW	“red”, “yellow”, or “green” (change clean/transfer behavior).
ViscosityLevel	String (enum)	RW	“low”, “medium” or “high” (change clean/transfer behavior).
EnablePlateBarcodeScan	boolean	RW	Enable/disable the scan of plate codebars.
EnableSpectrometer	boolean	RW	Enable/disable reading the spectrometer in sample transfer.
EnableVolumeDetectionInWell	boolean	RW	Enable/disable liquid volume detection in wells (Alidum).
SamplePathDeadVolume	double	RW	The pulled volume to reach the exposure cell.
LocalLockout	boolean	RW	Blocks the local access to the application interface.
AlarmList	String[]	RO	String contained the current alarms separated by ‘\n’.
HardwareInitPending	boolean	RO	A cold start is needed in next restart.
CoverOpen	boolean	RO	The state of the cover door.
CollisionDetected	boolean	RO	A collision has been detected . Error state: a restart is needed.
Power12OK	boolean	RO	State of 12V power.
WaterEmpty	boolean	RO	Water container is empty.
DetergentEmpty	boolean	RO	Detergent container is empty.
WasteFull	boolean	RO	Waste container is full.
WaterLevel	int	RO	Level of the water container (0 to 100).
DetergentLevel()	int	RO	Level of the detergent container (0 to 100).
WasteLevel()	int	RO	Level of the waste container (0 to 100).
OverflowVenturiOK	boolean	RO	Overflow venturi check.
CleanVenturiOK	boolean	RO	Clean venturi check.
Flooding	boolean	RO	Flooding has been detected.
VacuumOK	boolean	RO	Vacuum status.
PLCState	State	RO	State of the Twincat PLC.
CurrentLiquidPosition	int[]	RO	Array containing the currently detected liquid borders (if any).
FocusPosition	String (enum)	RW	“near”, ”middle” or “far”.
LightLevel	int	RW	Level of the light (0 to 10).
SampleVolumeWell	double	RO	The volume of liquid detected inside the well in the last operation.
ImageJPG	byte[]	RO	Byte array containing a snapshot of the image in JPG format.

Events

Event Callback Name	Info
onExposureCellFilled()	If external software performs cell filling detection, this method should be called to indicate that the cell has been filled.

Advanced Task Synchronization Methods

Method Signature	Info
String[] getTaskInfo(int id)	If id=-1 returns the information about the last executed task. Returns an array of strings containing the information about the given task: - Task Name - Flags - Start timestamp - End timestamp - Output - Exception - Result ID (negative=error, positive=success, 0=aborted)
boolean isTaskRunning(int id)	Returns if the given task is still running (and therefore cannot have the result checked).
String checkTaskResult(int id)	Returns the output of a task or throws an exception raised by the task.

Sample Concentration

If the spectrometer device is configured and enabled then the sample concentration can be calculated during a fill task. Optionally the "measureConcentration" command can be used to to get the spectrometer readouts of a given sample. The concentration is calculated based on the spectrometer readouts for the sample and its buffer in a single wavelength (280 nm).

When a fill task is started the attributes SpectrometerReadout and SpectrometerDarkReadout are cleared (set to NaN). A dark readout is first stored and and then the sample is moved to the spectrometer position. It is then stored the reference readout. Once the reference readout is done the value of the attribute SpectrometerReadout is set.

If the "measureConcentration" is executed, the task output will hold the readout and the dark readout.

The concentration (mg/ml) can be calculated as:

$$C_{(mg/ml)} = A_{@280}(10mm) / EC$$

$$A_{@280}(10mm) = A_{@280}(\text{path lenght}) * 10 / RPL$$

$$A_{@280}(\text{path lenght}) = - LOG[(I-I_{dark}) / (I0-I0_{dark})]$$

Where:

C = Sample concentration.

$A_{@280}(10mm)$ = Intensity for 280nm wavelength in the reference path length of 10mm.

$A_{@280}(\text{path lenght})$ = Intensity for 280nm wavelength in the real path length.

I = Attribute SpectrometerReadout for sample.

I_{dark} = Attribute SpectrometerDarkReadout for sample.

$I0$ = Attribute SpectrometerReadout for buffer.

$I0_{dark}$ = Attribute SpectrometerDarkReadout for buffer.

RPL = Attribute SpectrometerRealPathLenght (configuated during installation).

EC = Extinction coefficient (sample dependent)

Embedded Database

The application can be configured to start an embedded Derby database. This is a full-feature all java relational database that can be queried by SQL.

The database URL is: jdbc:derby://<hostname>:<port>/ctrldb

The port number is configured on the server, typically 1555 or the default Derby port 1527.

This database contains the following main tables:

- LOGS: Stores all logs with level WARNING or SEVERE.
- ALARMS : Store all alarms and corresponding time span.
- TASKS: Store all the executed tasks, start and end timestamps, return values and exceptions.

This embedded database simplifies a lot retrieving information about the use of the machine.

Example queries of the task table:

1. Total count of all tasks ordered by the number of occurrences.

   ```
   SELECT name, count(*) AS total FROM APP.TASKS WHERE STARTTIME>='2010-05-18 00:00:00' GROUP BY name ORDER BY total
   ```

2. Total count of all tasks that failed ordered by the number of occurrences.

   ```
   SELECT name, count(*) AS failures FROM APP.TASKS WHERE STARTTIME>='2010-05-18 00:00:00' AND NOT TASKEXCEPTION IS NULL AND NOT TASKEXCEPTION LIKE '%was stopped' GROUP BY name ORDER BY total
   ```

3. Details all the failed tasks .

   ```
   SELECT * FROM APP.TASKS WHERE STARTTIME>='2010-05-18 00:00:00' AND NOT TASKEXCEPTION IS NULL AND NOT TASKEXCEPTION LIKE '%was stopped'
   ```

4. List all clean operations.

   ```
   SELECT * FROM APP.TASKS WHERE STARTTIME>='2010-05-18 00:00:00' AND NAME LIKE '%clean%'
   ```

5. List all operations in a specific day.

   ```
   SELECT * FROM APP.TASKS WHERE STARTTIME>='2010-05-18 00:00:00' AND ENDTIME<'2010-05-19 00:00:00'
   ```