MANUAL

APPLICATION INTERFACING WITH CUA32 CONTROL UNITS

NEWSON NV

Table of Contents

1. first steps

This document describes how an application can control CUA32 devices. A complete license free software package, downloadable from the website www.newson.be, is available to bridge your application with CUA32 deflection technology.

software package

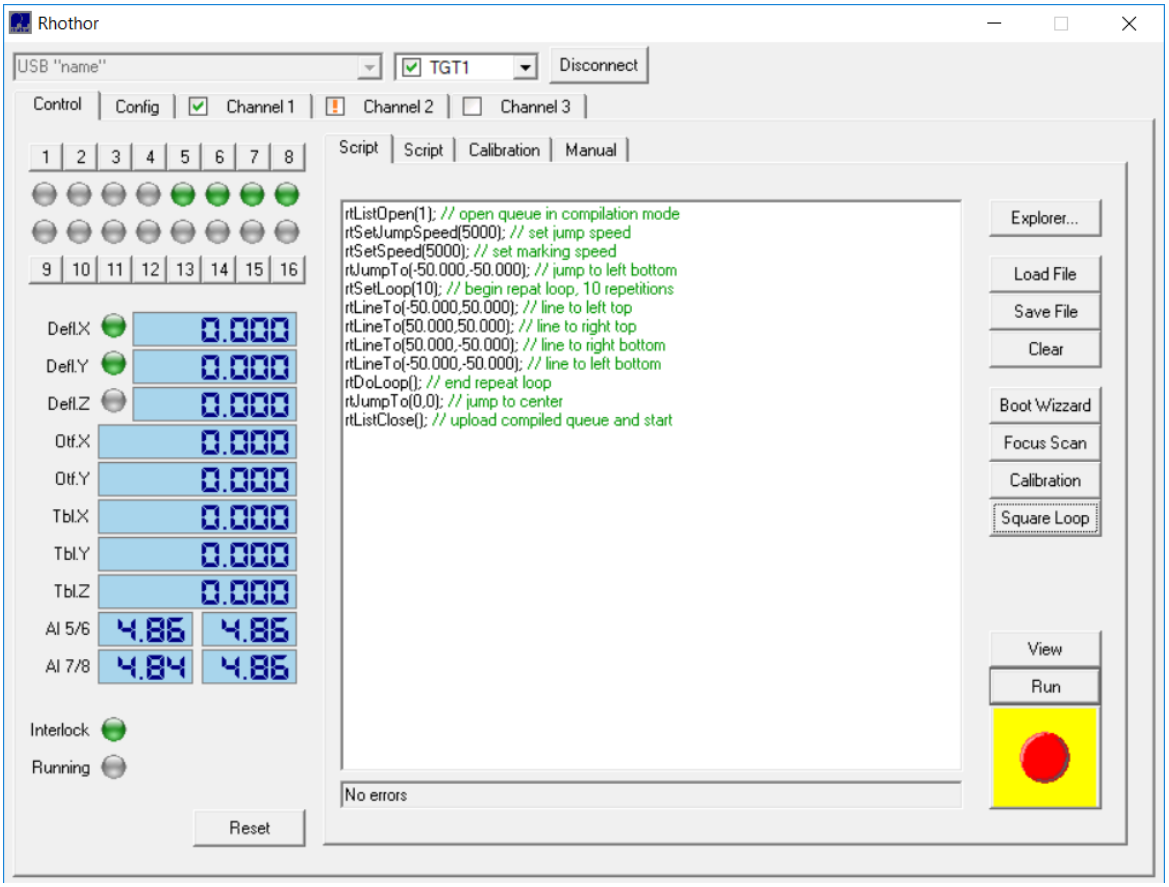
Library files " rhothor.dll","rhothorDLL.h"
Any windows application can easily setup a connection by means of this library. Source code of the library is available for customizing or compilation for other operating systems.

Executable "rhothor.exe"
System configuration, IO's, deflectors, table axis's, laser types and much more can be configured using "rhothor.exe". The application also has a text entry box to try out library function calls. As any other application, the executable talks with the hardware through the dynamic link library "rhothor.dll". A good understanding of the library functions and their impact on the hardware is key to build an application. Not intended to be used as a final application interface, the rhothor executable has easy to use features to fast track your application development.

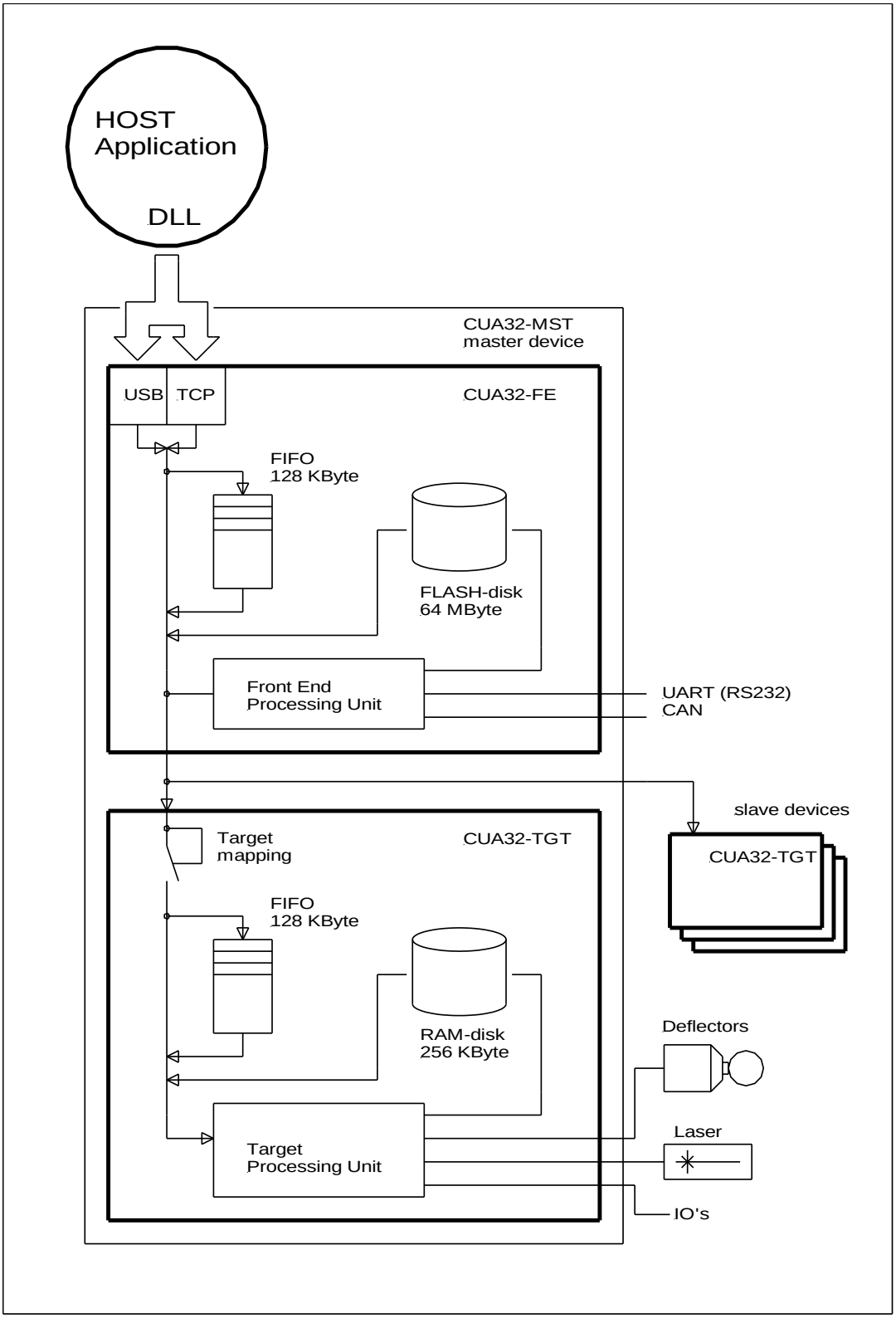
10 steps towards the first marking

1: Mount the deflection head and flat field lens
2: Connect laser and deflection head with the CUA32-MST control card
3: Download and install rhothor software (www.newson.be).
4: Switch on laser and power CUA32-MST
5: Make a USB connection between host and CUA32-MST.
6: Run rhothor.exe and make a connection.
7: Pressing the "Config" tab gives access to configuration windows.
8: Set field sizes, activate deflectors and select laser type.
9: The system becomes ready to use by pressing the "Control" tab.
10: Pressing "Square Loop" and "Run" marks a square 10 times over.

Function calls needed for this marking are shown by the rhothor executable. When called from within your application, the system will behave the same. Check your application development tool on how to include the rhothor library files. They define the methods to command and query your laser system. In following example, the deflectors field size was set to 100 mm. The screen print shows how the code to mark a full-field-square looks like.


Rhothor
USB "name"
TGT1
Disconnect
Control
Config
Channel 1
Channel 2
Channel 3
1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16
Defl.X 0.000
Defl.Y 0.000
Defl.Z 0.000
Otf.X 0.000
Otf.Y 0.000
Tbl.X 0.000
Tbl.Y 0.000
Tbl.Z 0.000
AI 5/6 4.86 4.86
AI 7/8 4.84 4.86
Interlock
Running
Reset
Script
Script
Calibration
Manual
rtListOpen(1); // open queue in compilation mode
rtSetJumpSpeed(5000); // set jump speed
rtSetSpeed(5000); // set marking speed
rtJumpTo(-50.000,-50.000); // jump to left bottom
rtSetLoop(10); // begin repat loop, 10 repetitions
rtLineTo(-50.000,50.000); // line to left top
rtLineTo(50.000,50.000); // line to right top
rtLineTo(50.000,-50.000); // line to right bottom
rtLineTo(-50.000,-50.000); // line to left bottom
rtDoLoop(); // end repeat loop
rtJumpTo(0,0); // jump to center
rtListClose(); // upload compiled queue and start
No errors
Explorer...
Load File
Save File
Clear
Boot Wizzard
Focus Scan
Calibration
Square Loop
View
Run

2. application interfacing



The CUA32 device is connected to the host computer over USB or ethernet. Data flow over this connection is managed by a DLL (rhothor.dll) comprising a large set of functions to query and command the CUA32 system. Queued functions are executed on a fifo based system while

interrupt functions issued by the application are executed immediately. The fifo is configurable by the application and used during marking to guarantee a constant flow of commands. When a command list is opened in streaming mode the fifo on the CUA32 controller is extended using memory of the host computer. When opened in compile mode the command storage is completely done using the CUA32 systems memory. The latter allows flow commands to be added for implementing loops and conditional processing without the need for host intervention. The execution list remains stored in the queue memory until all iterations are processed. Interrupt functions provide a zero-delay access to system resources allowing IO's and process parameters to be altered without disturbing a running job.

The CUA32-MST master controller comprises a front end and a target controller. The front end is responsible for handling command and data streams from the host application. The target controller holds all the logic needed to control three deflectors, three table stages, a laser and several IO's. The master controller can be extended with slave devices to handle more deflectors, table stages or IO's. Both front end and target controller have command buffers supporting streaming and compilation mode. When correctly executed, an application can start a perpetual command loop in one target while issuing a completely different command stream in another target.

Besides the volatile storage area for the command lists, the masters front end controller also comprises Flash memory formatted to 256 sectors of 256 Kbyte. 250 sectors can be used by the application as a file system to store and index command sequences. The remaining sectors are reserved for the operating system and configuration storage. At power-up the front-end controller initializes all connected target controllers using information stored on these sectors. The first file sector contains a boot start sequence. Commands thereof are executed automatically as a part of the system boot cycle. The dll provide easy to use primitives to create and copy flash files.

The target controllers in the CUA32-SLV slave devices do not have non-volatile memory. Every target controller needs to be initialized by the master's front end. All slave devices must be powered up before or together with the master device to guarantee successful booting. Besides the program list queue-memory the target controller has a (volatile) RAM disk that can be used to store a system font.

3. deflector control

The laser beam deflecting unit is the main means for positioning the laser beam onto the workpiece. The CUA32 controller executes commands like *rtLineTo* by changing the setpoints transmitted to the deflectors. Besides the obvious connections for an X- and Y-deflector, a third channel is provided to be used for dynamic focusing. All three deflector channels are controlled in phase to support full 3D-marking. The CUA32 controller also comprising functionality to process moving parts. In this case, the actual deflector positions are calculated based on both command processing and on-the-fly position measurements (OTF X, OTF Y, OTF Z).

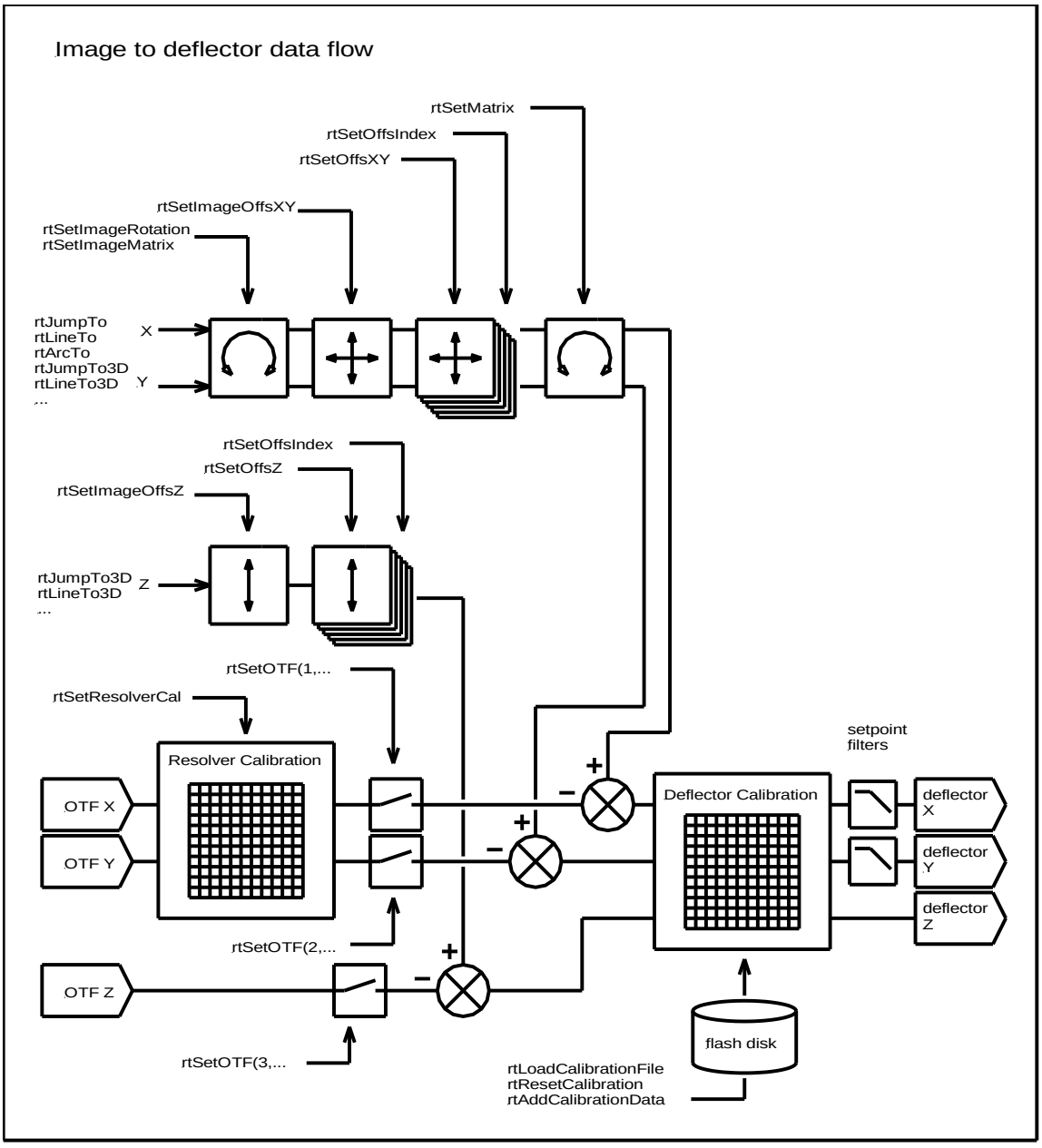


Image coordinates (*rtLineTo, rtArcTo*,…) pass through an image transformation matrix (*rtSetImageRotation,rtSetImageMatrix*) before being offset by image offset data (*rtSetImageOffset*). The interpolation processor uses said transformation and offsetting when generating the stream of setpoints. When an application invokes *rtSetImageMatrix* with a determinant larger than one, coordinate data will be enlarged and execution time automatically prolonged by the same factor. The image transformation matrix maintains the marking speed.

The output data from the interpolation processor passes through a second coordinate transformation system. The latter (controlled by *rtSetOffsXY,rtSetMatrix,…*) is commonly used to align the deflection system with the axis system of the outside world. It scales the coordinates from the stream on a sequential basis. If the transformation matrix has a determinant larger than 1, the actual speed will be higher by the same amount. The system transformation matrix maintains the execution time. The latter can be important when an application needs to run the same job with different scaling's simultaneously on several targets

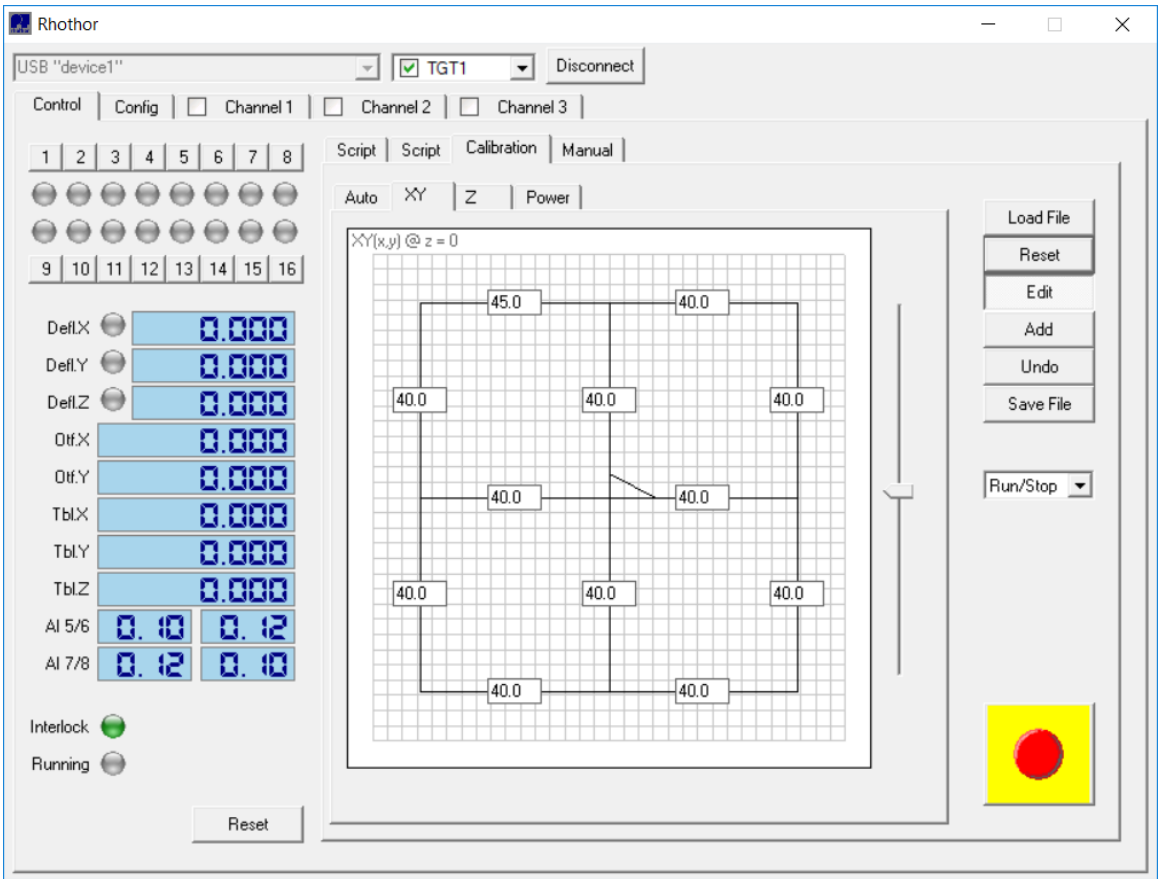
The resulting setpoint streams contain high frequency components that are hard to handle by the deflectors. Limited stiffness of mirror and mechanics can induce a ripple style behavior during a move. When used to drive RLA, RTA and CYA deflectors, the CUA32 controller uses switchable setpoint filters to improve track quality in high frequency applications. By limiting the bandwidth of the setpoint signal, current spikes inside the deflector are reduced without changing its dynamics. Knowledge of this filter and his configuration is important for understanding the values that have to be used in dll-function *rtSetLaserDelay*. When used with a data convertor RTBE-XY2 and third-party scanners, the setpoint low pass filters are disabled. The time constants are set to zero and setpoints are transmitted as is without any bandwidth limitation.

4. deflector calibration

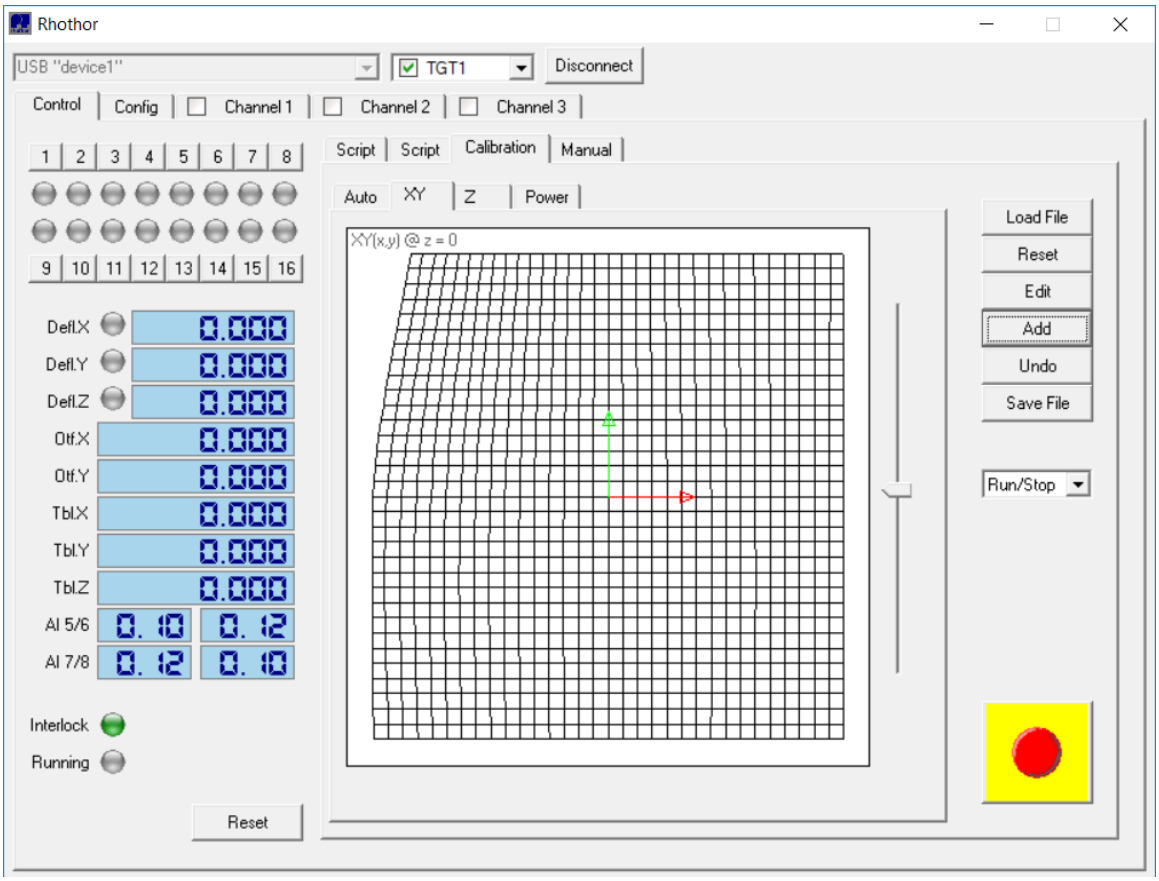
Deflection systems suffer from positional errors. The CUA32 systems handle these errors by counter steering during the movement of the laser beam.

mark square and measure

The rhothor configuration software provides means for easy calibration. First step is setting the calibration size (“Config tab”). This value should be at least equal with the area actually used by the application. A calibration figure can be generated into the rhothor script editor by pressing “Calibration” button (“Control/Script” tab). The figure is a square with vertical and horizontal center lines covering the full calibration size. A small triangle in the center defines the axis system. After marking (press “Run”), it can be measured. The actual sizes can be entered in the “Control/Calibration/XY” tab after pressing the “Edit” button. Pressing “Add” will update the targets calibration. Like any other application, rhothor uses the *rtAddCalibrationData* function for uploading the data. A temporary file “rhothor.tmp” is created for this purpose. When “Add” returns, the file still exists and can be loaded into a text editor for evaluation.



In this example the target's field size was set to 100 and the calibration size to 80. After changing the left top edit box from 40 to 45 and pressing "Add" the calibration was changed and now looks like:



mark fiducials and measure

The CUA32 controller maps its total scan field (field size) with an orthogonal grid of 33 by 33 points to implement the calibration math. Offset X-, Y- and Z-values are stored for every point. The flash of the CUA32-MST master devices contains eight of those vector sets, one for each target device.

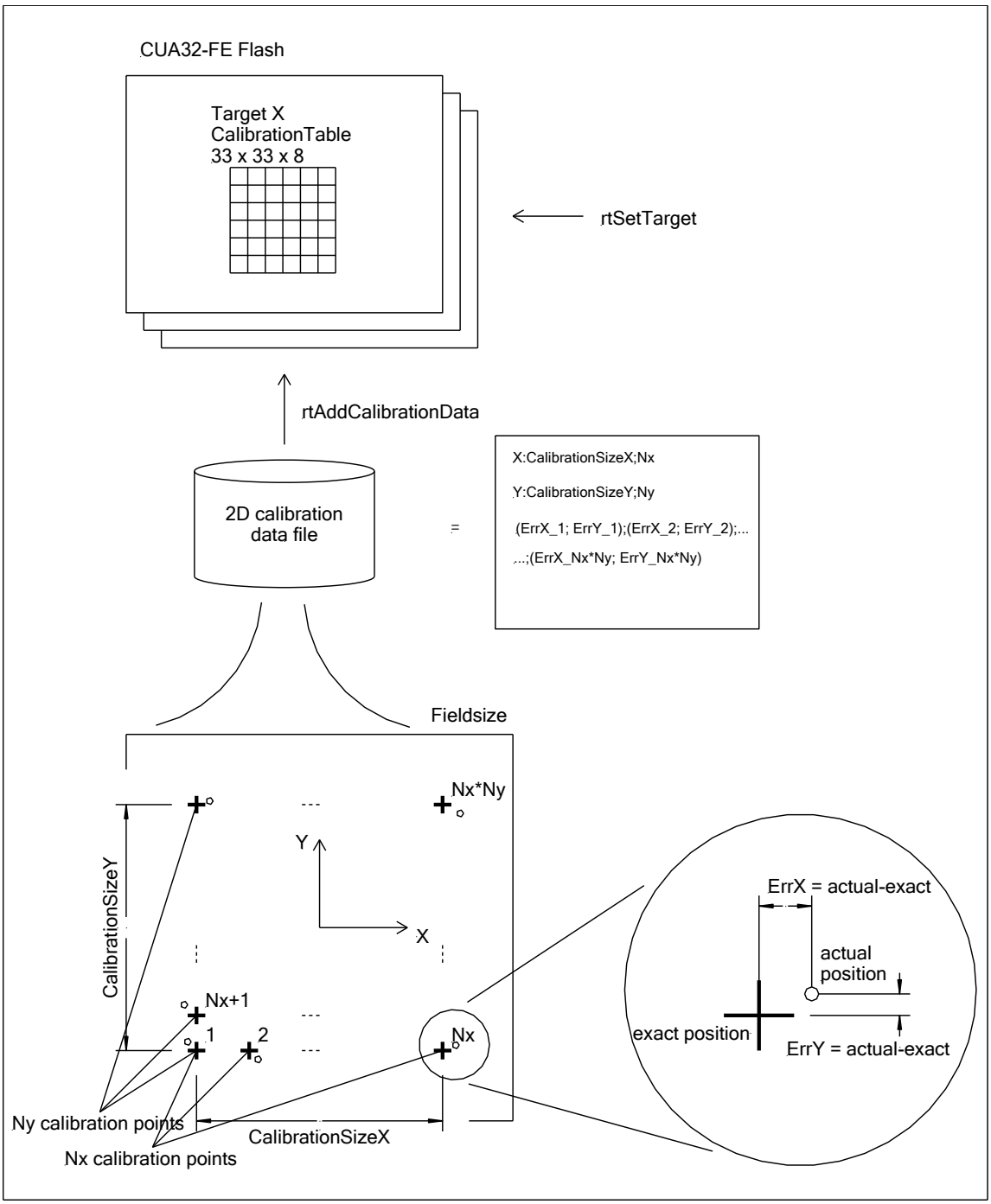
Marking an array of fiducials and measuring their positions is a common way to quantify the image distortion. Any number between 3 by 3 to 33 by 33 fiducials may be chosen. The application needs to compile the measurements into a calibration data text file. The data file must comprise a header declaring calibration size and number of calibration points followed by array data. All dimensions, coordinates and offsets are in mm. The function r*tAddCalibrationData* remaps the data stored in said file and updates the calibration. Remapping is needed because the number of fiducials marked and measured to gather the distortion data is likely to be different than 33 by 33. Furthermore, the calibration size will likely be smaller than the field size. The field size of a marking unit is the result of the angular range of the deflectors combined with the focal length of the optics while the calibration size is the applications actual working area.

Generating a calibration is an iterative process and needs a few iterations. A single shoot and measure sequence is unlikely sufficient to do the job. Not all the runs need to be done with the same number of calibration points. It is a common practice to start the first calibration run with 3*3 calibrations points and increase the number of calibration points as the iteration progresses.

CUA32 devices support full 3D marking. A dynamic beam expander can easily be constructed using an ELA-TR4 actuator. By changing the distance between two lenses, their combined focal length can be altered allowing the deflector system to change its focal plane. Gathering offset data should be done on two Z levels to obtain full 3D calibration.

2D calibration data file

```
// HEADER: sizes in mm
// ARRAY DATA:
// actual position- ideal position (mm) for every fiducial
// starting at left bottom up to right top, X direction first
// z calibration data is optional
X:calibration_size;samples    // header
Y:calibration_size;samples    // header
(dx;dy[;dz]);...;(dx;dy[;dz]) // array data
EOF
```

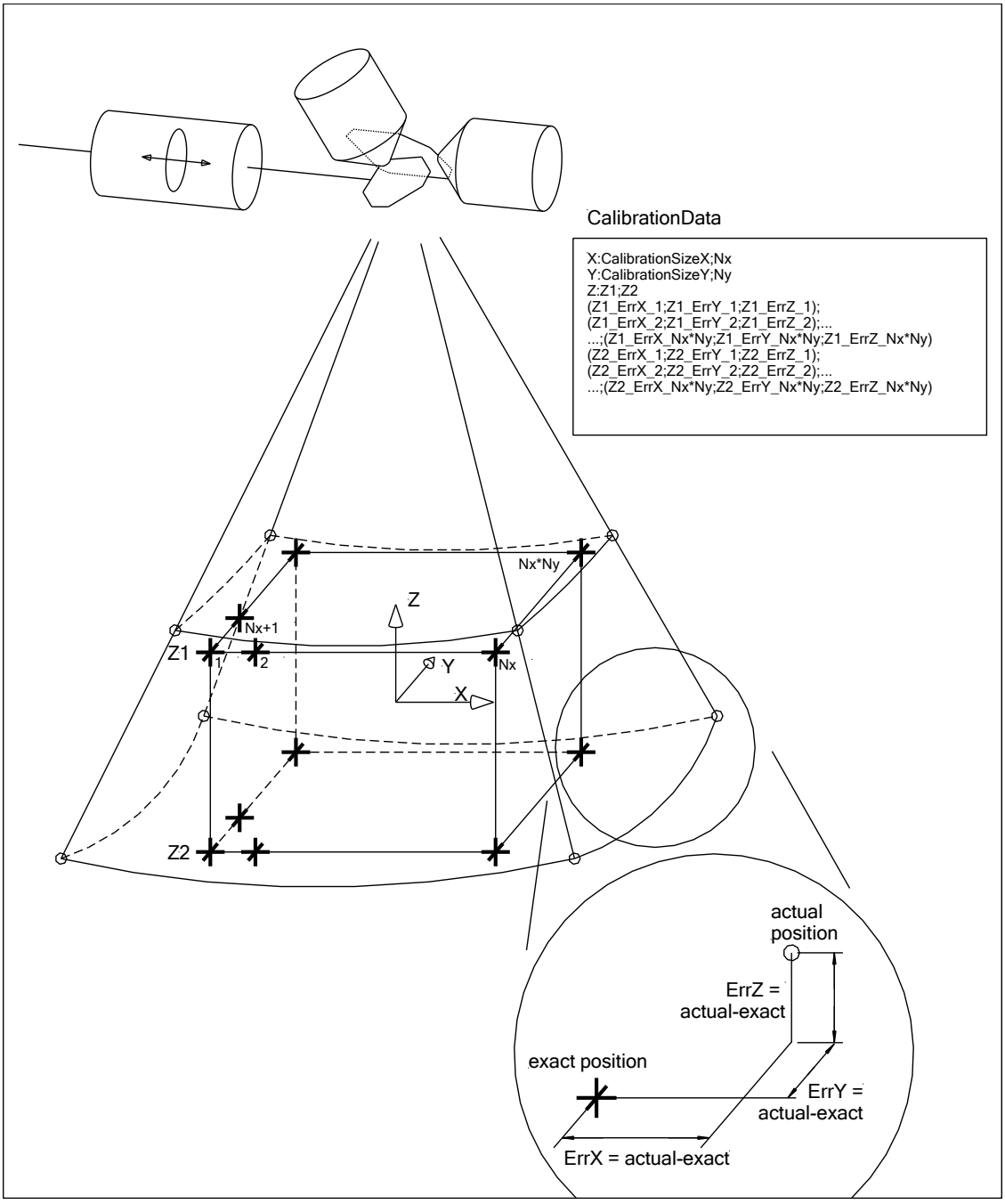


The “rhothor.tmp” calibration data file created in example of “mark square and measure”:

```
X:80.000;3;
Y:80.000;3;
(0.000;0.000);(0.000;0.000);(0.000;0.000);
(0.000;0.000);(0.000;0.000);(0.000;0.000);
(-5.000;0.000);(0.000;0.000);(0.000;0.000);
```

3D calibration data file

```
X:x_calibration_size;x_samples // header
Y:y_calibration_size;y_samples // header
Z:Z1,Z2                        // header
(dx;dy[;dz]);...;(dx;dy[;dz])  // array data at Z=Z1
(dx;dy[;dz]);...;(dx;dy[;dz])  // array data at Z=Z2
EOF
```

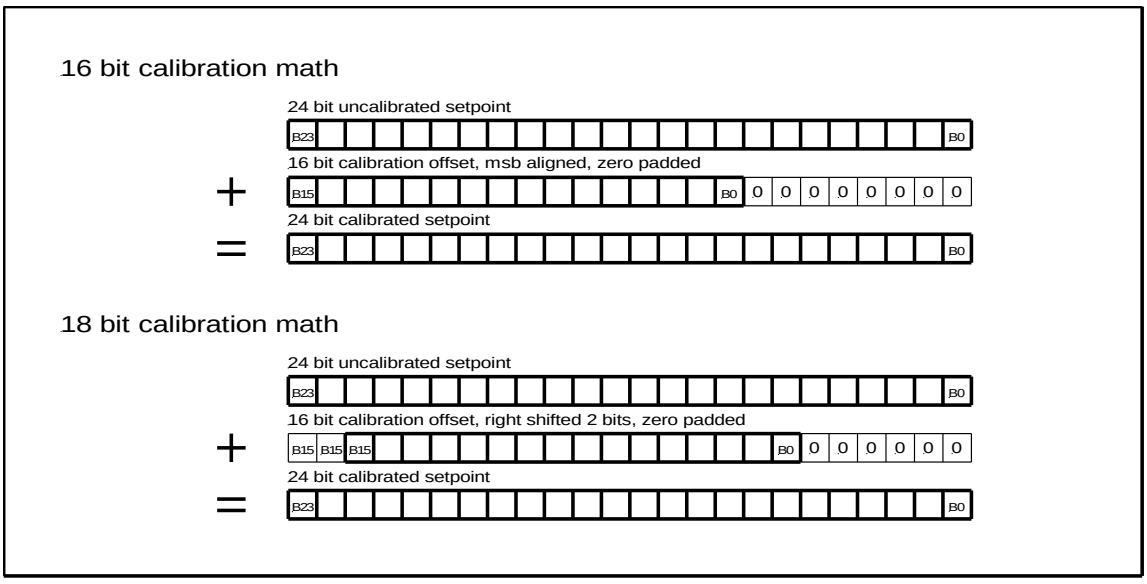


beacon fully automatic calibration

Previous methods involve marking and measuring. While commonly used and known, time and waste materials are needed to achieve calibration. The rhothor software allows connection with an optional beacon calibration system. Said system supports fully automatic 2D calibration using the actual laser light. There is no need to compile files, to measure are even to mark. After the beacon calibration device is placed in the process area of the deflection head, it handles the entire calibration process.

16/18 bit

The calibration is stored on flash as arrays of 16-bit offset vectors. Setpoints send to deflectors are formatted as 24-bit integer values. For compatibility with previous controller versions, CUA32 controllers can be configured to 16-bit-align or to 18-bit-align the calibration data. While the 16-bit alignment allows for dramatic calibrations, mapping the values on 18 bit increases the calibration accuracy. Resolution selection for the calibration data must be done using the rhothor configuration software prior to calibration. This resolution selection is only available for X- and Y-channel. The Z-axis always uses the 16-bit alignment.



5. laser control

During processing, the CUA32 controls deflectors and laser. Operational moves have to be done with laser activated while idle moves are executed with laser switched off.

gate mode

The easiest way to control a laser is by means of a gate signal. Whenever the gate signal is active, the laser is emitting power. While this approach is straight forward additional laser control modes are supported. Selection between said mode is done using the rhothor configuration software while library functions (*rtSetOscillator*, *rtSetLaserFirstPulse*, *rtSetLaserTimes*,…) provide means to set process parameters by the application.

burst mode

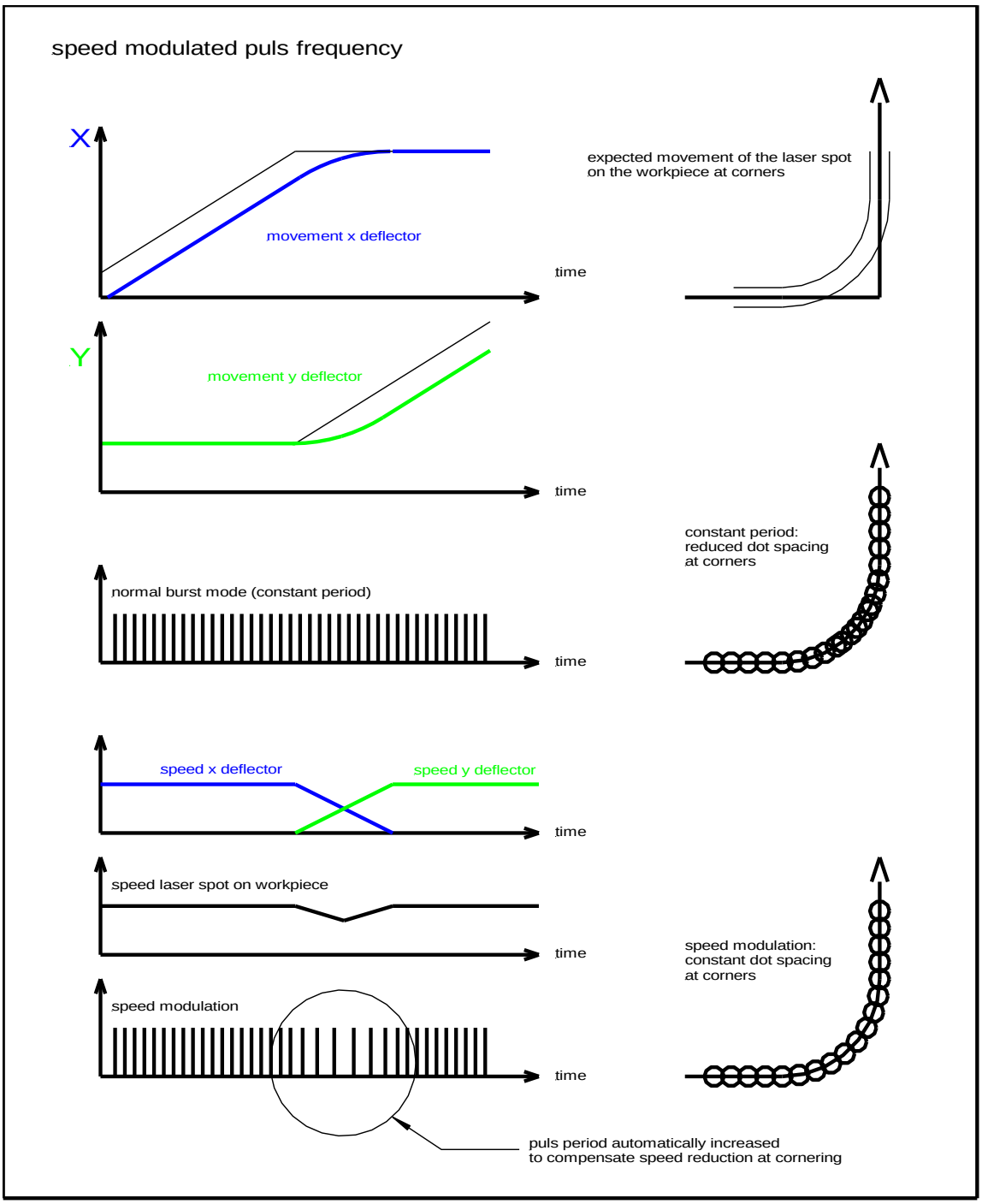
Most lasers do not source a constant optical output but run sequences of high energy optical pulses. The frequency of said pulses can be generated internally or applied by an external signal. For this purpose, the CUA32 has a programmable oscillator who can be configured to generate pulses whenever laser energy is needed. Because this oscillator uses the same clock frequency as the command processor, the phase relation between pulses and deflector movements is constant. This means that, between successive markings, the actual laser dots on the workpiece will always be on the same location. When the pulse frequency is generated by electronics inside the laser this will not be case by lack of synchronization. The laser has its digital logic running on its own clock. Furthermore, the CUA32 controller will resynchronize the oscillator on every rising flank of the gate. When parallel lines are lasered, the laser dots will form a nice square like array. The latter provides better and more homogenous laser hatching results.

CO2 mode

In this mode, the CUA32 controller generates pulses all the time regardless of the laser state. The gate signal is used to change the pulse width. This control mode is commonly used with CO2 lasers. Whenever the laser must be switched on the pulse duty cycle (= width/period) defines the optical output energy. When idle, the laser still gets (tickle) pulses. However, these pulses have a very small pulse width and allow the laser to generate just enough energy to sustain readiness.

burst with speed modulated period

Some processes have a very tight operational window. The optical energy density needs to be as constant as possible over the entire lasered line. Said density is defined by laser output power and track speed. However due to deflector delays the latter can vary. Extending lines with idle up and down ramping is a common way to guarantee that the track speed is constant whenever the laser is activated. This solution comes with a cost of increased processing time and software overhead. The pulse oscillator on the CUA32 controller supports speed modulation as a zero-cost alternative. When cornering, the track speed will go down due to the rounding effect. The CUA32 controller will automatically reduce the pulse frequency to counter act and keep the inter dot distance on the track constant. As in burst mode, the pulse-period and width are set by invoking library function *rtSetOscillator(3,..)*. The selected period will be linked to the active track speed (set by last call *rtSetSpeed*). Whenever the actual speed reduces, the period will automatically be increased to guarantee constant dot pitch on the workpiece.



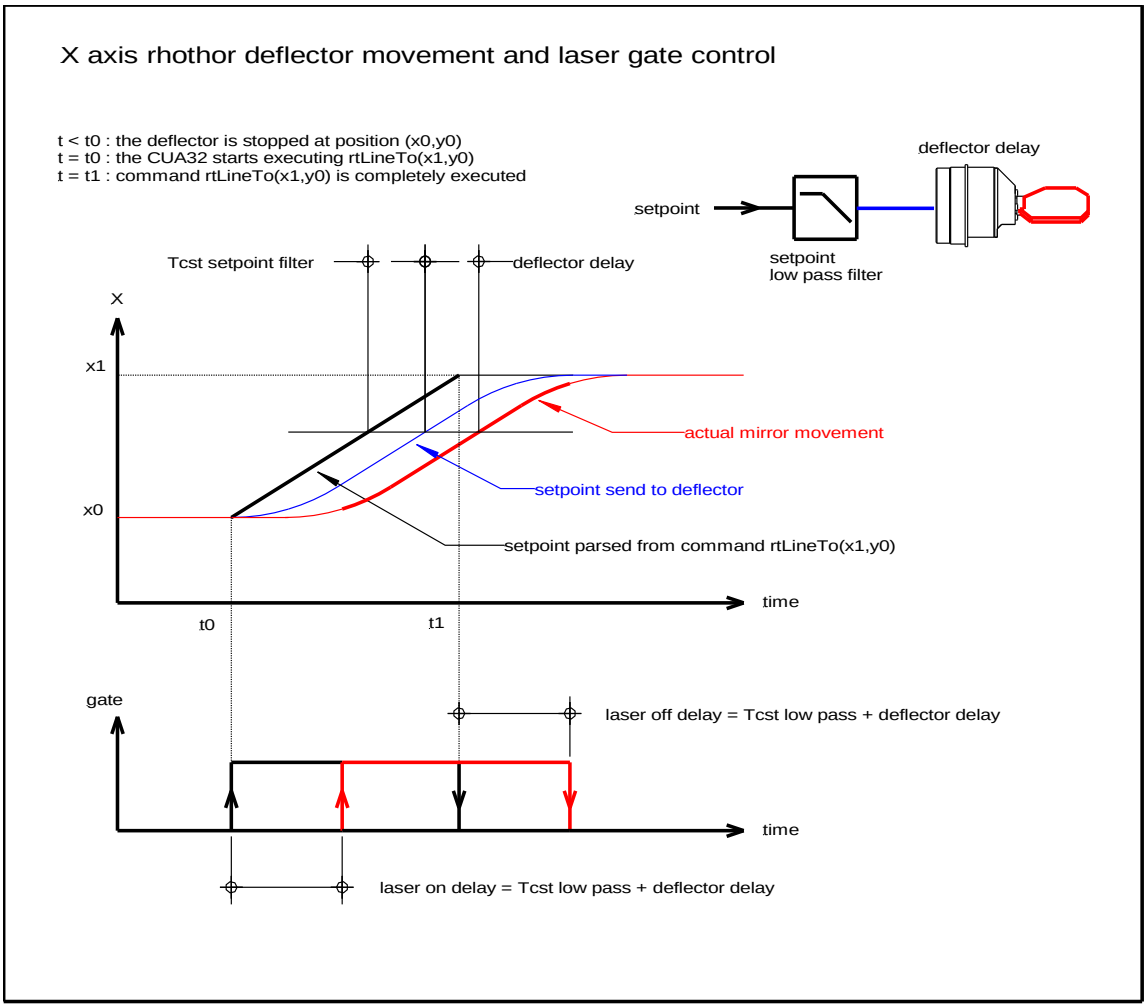
laser link

For some lasers, optional devices are available to ease interfacing. Lasers needing TTL like signals to set power, query alarms, controlling peripherals often end up with high pin count connectors. Laser links are smart connectors comprising electronics to support the laser side interfacing while exchanging laser data with the CUA32 controller over a single twisted pair cable.

on-off delays

Whatever laser mode is selected, in all cases the laser control signals (gate, burst...) need to be synchronized with the actual mirror positioning. Time delay induced by the deflectors and their setpoint filters can be compensated by shifting the laser control signals. Using the library function *rtSetLaserTimes*, laser on delay and laser off delay can be set independently. In most cases both delays can be calculated as follows:

GateOnDelay = *GateOffDelay* = deflector delay +time constant setpoint filter.



power

Besides changing the burst frequency, output power of a laser can often be altered by changing the optical pump energy. How lasers work lies well beyond the scope of this manual. However, some understanding is needed. It's obvious that with a pulsed laser, the output power is a linear function of applied burst frequency. Assuming that pulse energy remains constant, doubling its frequency doubles the output power. Sadly, in most cases, said assumption is not valid. Furthermore, the application parameters may require a specific dot distance on the workpiece limiting the range in which one can set the burst frequency. Most lasers have an additional analog input to control the pump energy. This input alters the energy of a laser pulse not it's frequency. The CUA32-TGT controller features a dedicated analog output for this purpose. Output power of CO2 laser is commonly controlled using duty cycle modulation of the burst signal (PWM), so no additional resources are needed.

Functions like *rtSetPower*, *rtSetPowerLevels*, *rtSetPowerProfile*,… provide means to control said laser pump energy, independent of laser type. So, marking jobs can transferred from one system to another, without the need to adapt commands therein to the applied laser system.
Available means to control power outputs:

- IO5 can configured as a power-output (gate- and burst-mode)
- Oscillator 1 pulse with modulation (CO2-mode)
- Invoking set power command on a Laser-link

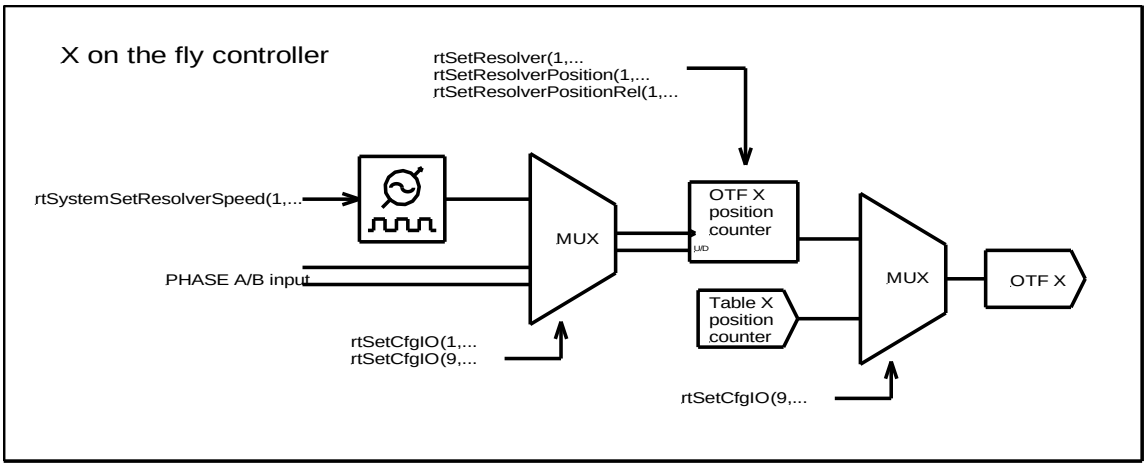
Based on the configuration, an *rtSetPower* command will automatically select between them. On a CO2-laser, the command will alter the pulse width while on a burst laser, the analog output will be used. Whenever IO5 is configured for power-output, pulse width modulation (CO2 mode) or invoking laser link command become suppressed.

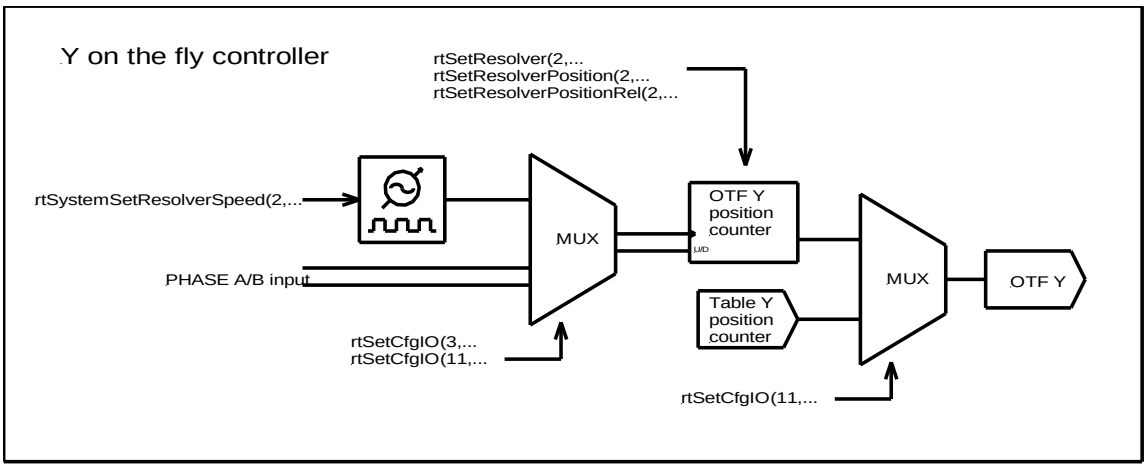
6. on-the-fly control

When a workpiece moves during its marking, the deflection system needs to compensate said movements for correct printing. Any position sensing device outputting quadrature signals can be connected with the CUA32-controller. The CUA32-TGT device comprises on-the-fly state machines able to integrate the applied position signals. Based on their sequence, any flank measured on either A- or B-signal will result in incrementing or decrementing the on-the-fly position counter.

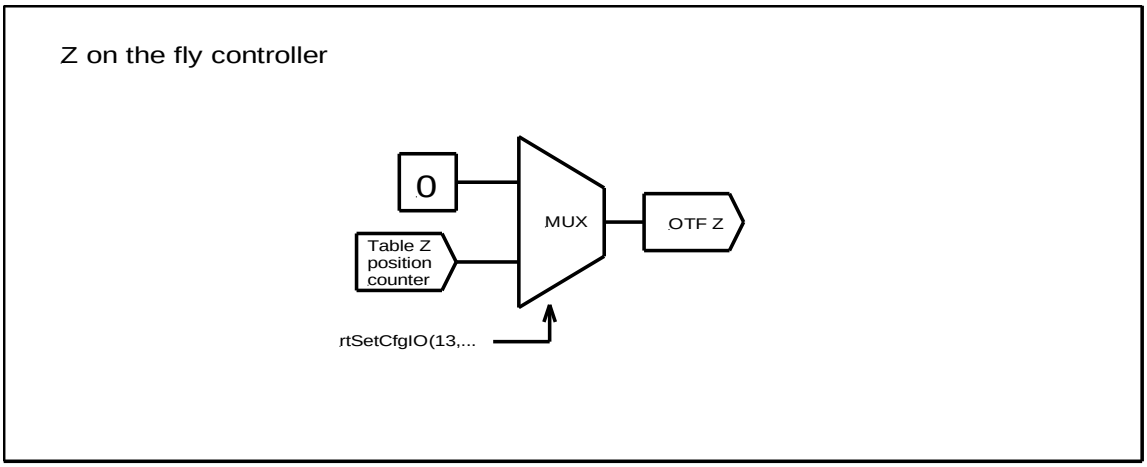
When position sensing is not available, a constant speed mode can serve as a position estimation. This mode is particularly useful when marking components on moving trays. The speed with which the tray moves during the marking is likely to be constant. When needed, an IO hardware trigger can be added for better positioning the marking onto the work piece.

A third source that can serve for position sensing is a write through of the table position. A CUA32-controller is a full 6 axis system able to control 3 table stages. The on-the-fly state machines can be set so their outputs are overwritten by said table setpoints.





Unlike X- and Y-channel, the on-the-fly state machine for the Z-channel only supports the table position write through mode.

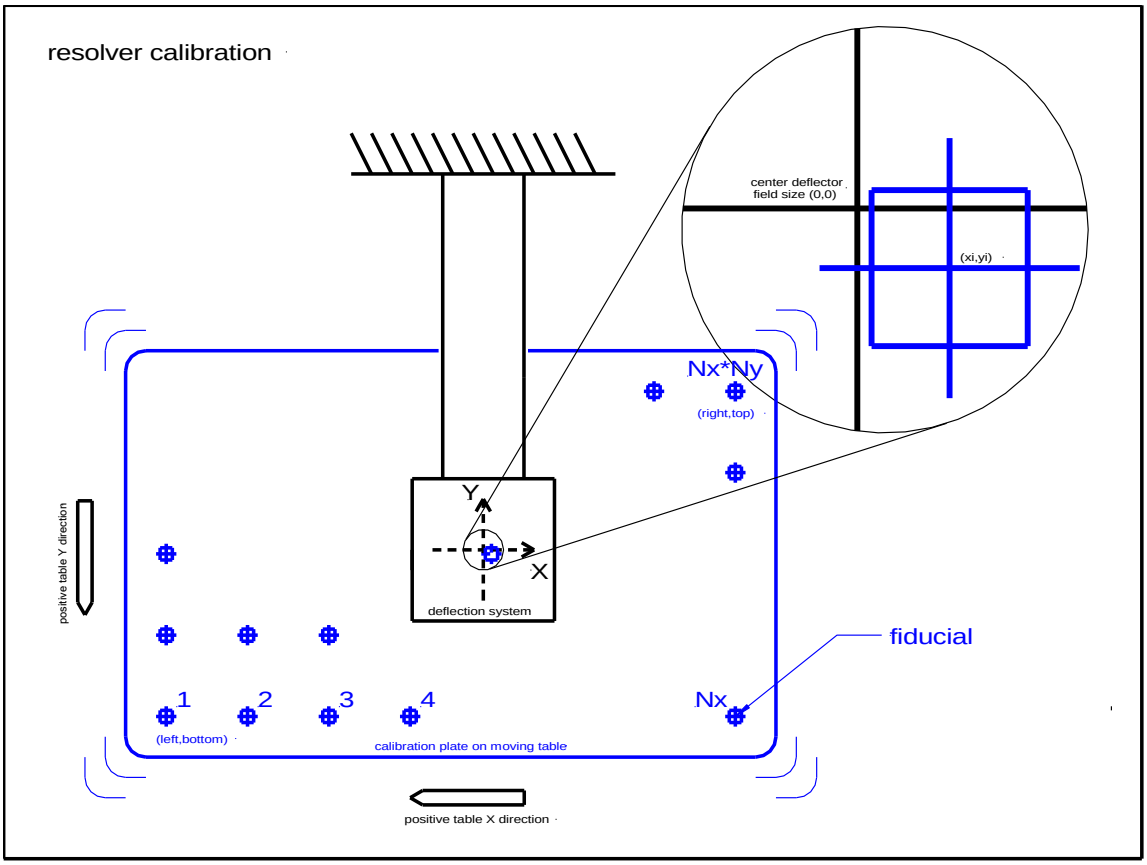


7. on-the-fly resolver calibration

When resolver inputs are used to determine the position of the deflection system in relation to the workpiece, errors can be introduced by sensor linearities and alignments. The CUA32-TGT controller comprises a tool to map out those errors. The method *rtSetResolverCal* loads offset data stored in *FileName* into the selected target controller. Said offset data will be subtracted from the resolver counts during on-the-fly operation. Unlike deflector calibration, resolver calibration is not stored in system flash and needs to be uploaded by the application as part of a boot cycle.

Resolver calibration data file

```
// HEADER: sizes in mm
// ARRAY DATA:
// actual position fiducial as seen by deflector when
// deflection head is positioned right above said fiducial
// starting at left bottom up to right top, X direction first
X:left;right;Xsamples          // header
Y:bottom;top;Ysamples          // header
(x1;y1);...;(xn;yn)            // array data
EOF
```



An example:

Assume that, in previous drawing, the fiducial coordinates beneath the deflector equal (1000,500). The X resolver counted to 1000 and the Y resolver to 500. The controller assumes that the deflector is positioned right above the fiducial. The X count, 1000, appeared to be too high, the correct X count should be 998. The Y count, 500, appeared to be too low, the correct Y count should be 501. Notice that in this the setup, using a fixed deflection system and a moving workpiece, the positive on-the-fly count directions are opposite to the image coordinate system. If the controller uses the OTFC (on-the-fly counts) as is, a rtJumpTo(1000,500) command will position the deflector system dead center. Markings will be error wise 2 mm to the left and 1 mm too high.

By defining “error = OTFC - correct count” and measure and map said errors , the CUA32-TGT controller is able to calculate the correct part position by subtracting the error data from the on-the-fly-counts:
“correct position = OTFC- error = OTFC –(OTFC-correct-count)= correct count”

In this case the on-the-fly count error equals (2,-1) when the workpiece is positioned at (1000,500). After compensation of the on-the-fly-counts the position is measured at (998,501). A rtJumpTo(1000,500) will position the deflector system at (1000-998,500-501).

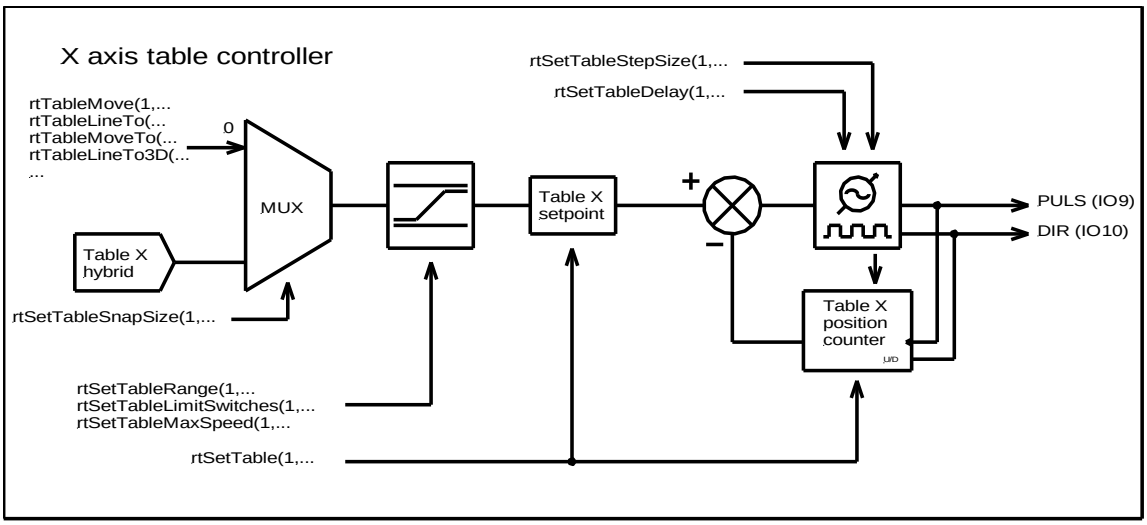
8. table axis control

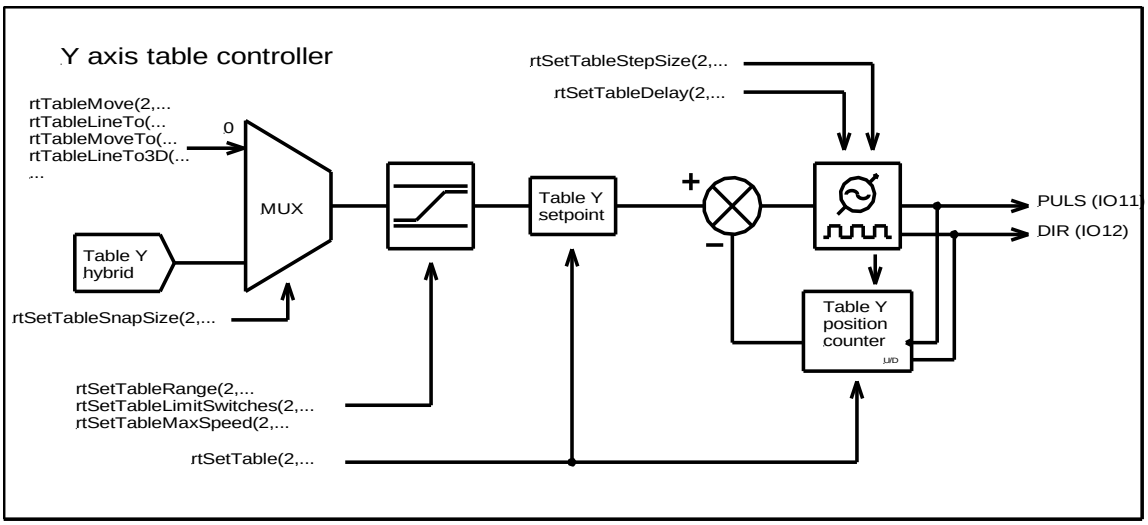
The CUA32 is a six-axis numerical control system. Besides three deflectors, three table stages can be controlled. Laser beam deflection systems are very fast but have a limited field size. When larger areas are needed, a mechanical table is used to change the position of the deflection system in relation with the workpiece. A machine comprising a XYZ table and a 3D-deflection system can process parts as large as the travel range of its table at a speed not limited by it. To control such a machine, the application needs to handle six axes.

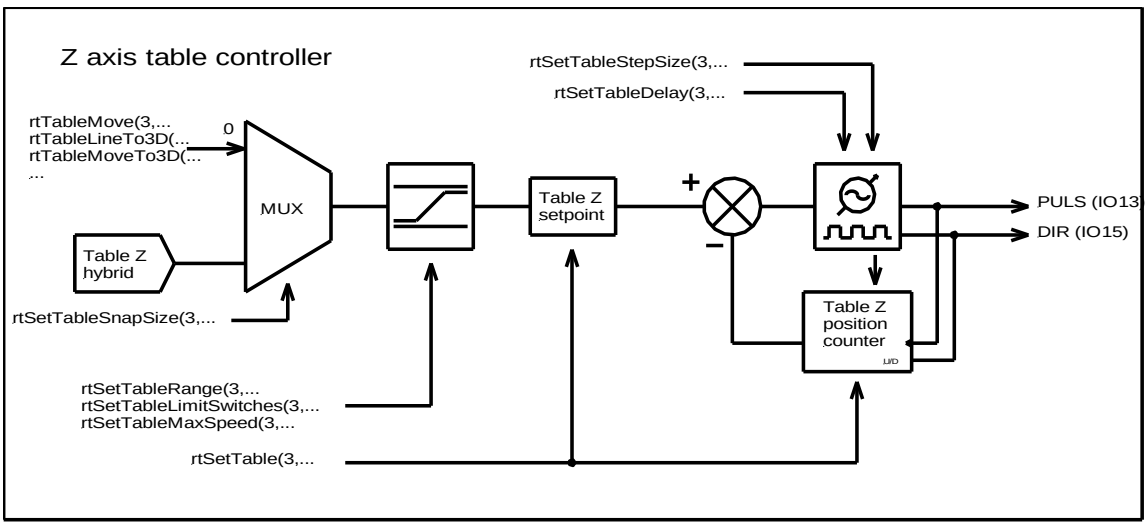
In normal mode, table and deflector are controlled by their own commands. Each having their own coordinate system. Image functions (*rtLineTo, rtArcTo...*) will invoke a deflector movement while table functions (*rtTableMove, rtTableLineTo...*) will position the table. Image coordinates pass through several data transformation steps before being transferred to the deflectors. Their values are shifted and rotated as requested by preceding functions (*rtSetImageMatrix, rtSetOffsXY, rtSetMatrix,...*). Coordinates used in table functions are send to the stepper controllers as is, without any offsetting nor rotation. Calling *"rtSetOffsXY"* will offset the coordinates of a *"rtMoveTo"* function but will have no effect on a *"rtTableMoveTo"* function. When controlled separately, the application needs to make sure that table is put in position before starting the deflectors. To avoid clipping, the graphical content processed by the deflection head must fit within its limited working area. The latter is straight forward when only small isolated images have to be marked. Processing images larger than the field size of the deflection head imposes challenges.

To reduce application overhead, the CUA32 device supports a hybrid control mode. In this mode, the table and deflectors share the image coordinate system. A *rtLineTo* will mark using both deflectors and steppers at the same time. The long straights will be done using the table, while the smaller graphical data will be handled by the deflectors. A hysteresis scheme avoids unnecessary table movement. Because the image coordinate system is used, the positions are rotated and shifted before being send to the hybrid axes. The CUA32 control board uses the available on-the-fly hardware to adjust the deflector setpoints. Both open loop and closed loop style operations are supported. In open loop, the on-the-fly position values reflect their respective table setpoints. When a position feedback mechanism is available, its quadrature signals (PHASE A, PHASE B) can be used to increase overall accuracy. The application can activate/deactivate this hybrid mode at any time (*rtSetTableSnapSize*, *rtSetTableSnapSizeEx*).

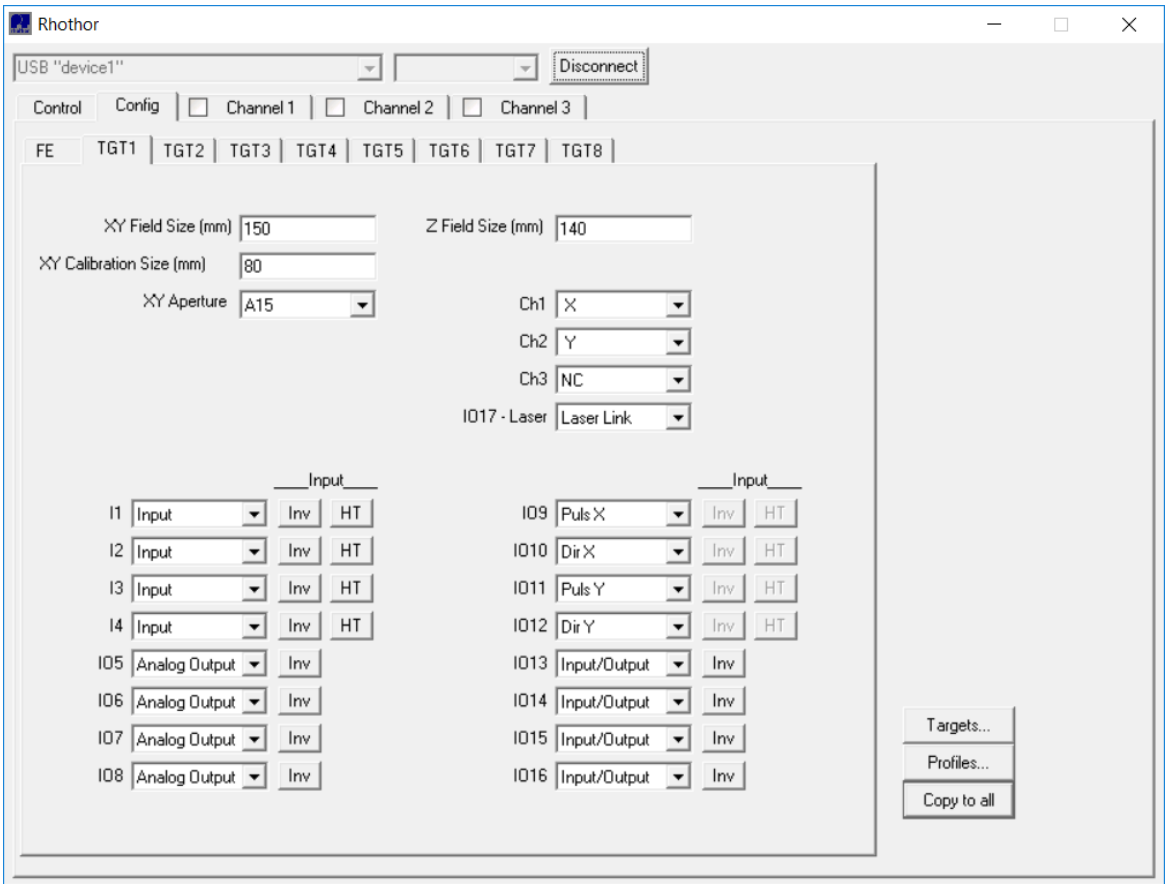
The CUA32-TGT device comprises three table axis controllers. Third party axis drivers can be controlled through *PULS* and *DIR* signals. The CUA32 controller comprises all the logic needed for smooth ramping by limiting frequency changes on said signals.


X axis table controller
rtSetTableStepSize(1,...
rtSetTableDelay(1,...
rtTableMove(1,...
rtTableLineTo(...
rtTableMoveTo(...
rtTableLineTo3D(...
...
0
MUX
Table X hybrid
rtSetTableSnapSize(1,...
Table X setpoint
+
−
PULS (IO9)
DIR (IO10)
Table X position counter
CNT
rtSetTableRange(1,...
rtSetTableLimitSwitches(1,...
rtSetTableMaxSpeed(1,...
rtSetTable(1,...


Y axis table controller
rtSetTableStepSize(2,...
rtSetTableDelay(2,...
rtTableMove(2,...
rtTableLineTo(...
rtTableMoveTo(...
rtTableLineTo3D(...
...
0
MUX
Table Y hybrid
rtSetTableSnapSize(2,...
Table Y setpoint
+
−
PULS (IO11)
DIR (IO12)
Table Y position counter
CNT
rtSetTableRange(2,...
rtSetTableLimitSwitches(2,...
rtSetTableMaxSpeed(2,...
rtSetTable(2,...


Z axis table controller
rtSetTableStepSize(3,...
rtSetTableDelay(3,...
rtTableMove(3,...
rtTableLineTo3D(...
rtTableMoveTo3D(...
...
0
MUX
Table Z hybrid
rtSetTableSnapSize(3,...
Table Z setpoint
+
−
PULS (IO13)
DIR (IO15)
Table Z position counter
CNT
rtSetTableRange(3,...
rtSetTableLimitSwitches(3,...
rtSetTableMaxSpeed(3,...
rtSetTable(3,...

When a CUA32-TGT device controls a table axis, the respective IO pins must be allocated. The following screen-print from the rhothor executable shows a configuration wherein the CUA32-TGT is set to control the X axis through IO9 and10 and the Y axis through IO11 and 12.

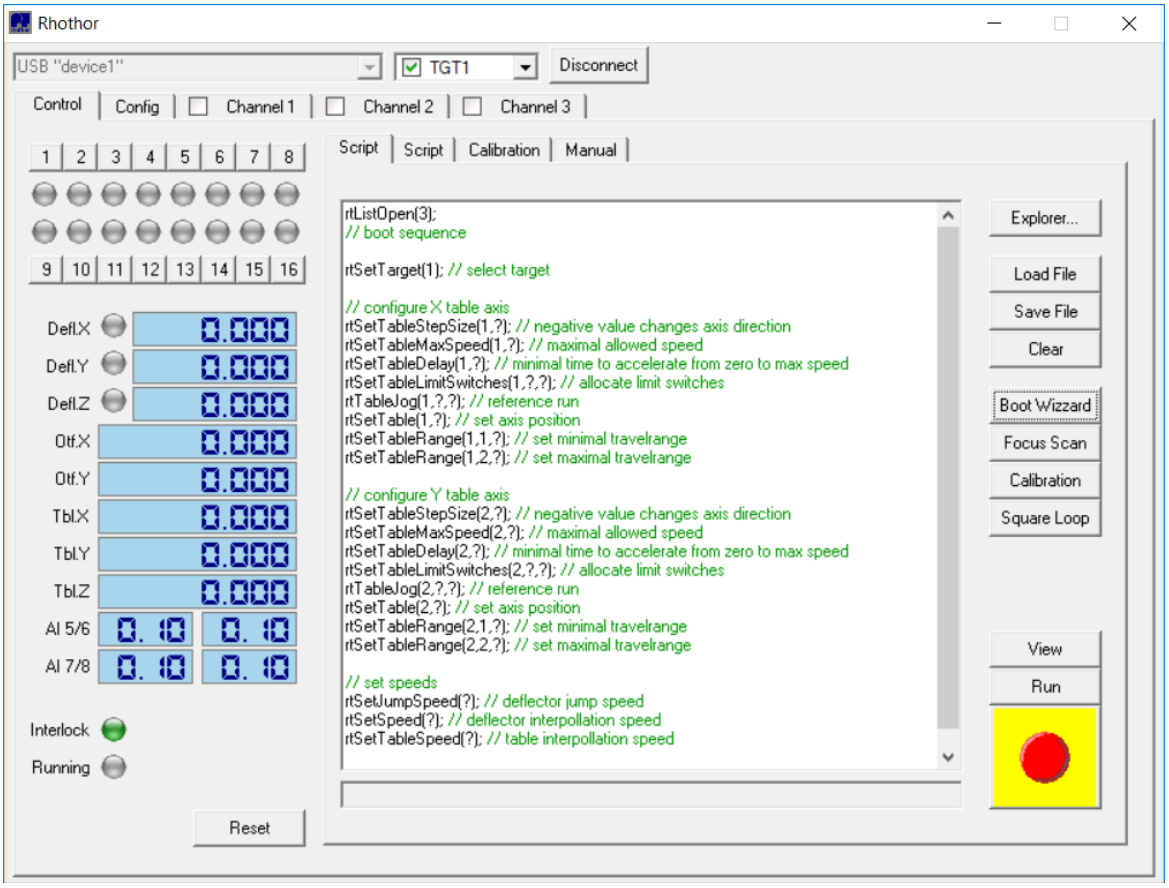


This configuration only allocates the IO pins. Before a table axis can be used, following data needs to initialized:

data	description	available function
step size	How much does the table move when one pulse is applied?	rtSetTableStepSize
direction	In which direction does the table move when the “DIR” pin is low and pulses are applied?	rtSetTableStepSize
maximal speed	What is the maximal pulse frequency that can be applied?	rtSetTableMaxSpeed
maximal acceleration	How fast can we accelerate from stand still to maximal speed	rtSetTableDelay
limit switches	Does the table have limit or reference switches?	rtSetTableLimitSwitches
table ranges	Defines the field size of the table axis.	rtSetTableRange

The table axis initialization can be done by the application or at power up using the boot start executable. When the “Boot Wizard” button is pressed, the rhothor executable loads the “Script” edit box with the commands needed to initialize the axes. Depending on the axis setup some commands can be left out or additional commands must be added. In any case, configuring an axis starts with

defining its step size and direction. As long as *rtSetTableStepSize* for the axis isn't invoked all other table commands are ignored. Fill in the question marks in the command sequences and press the "Run" button to program the boot start file on flash. Press the "Reset" button to actually run the script. Even when using the boot start file is not desired, this tool can be used as a source code generator for the application. The code lines in the "Script" edit box can be copied through the clip board into any development environment.



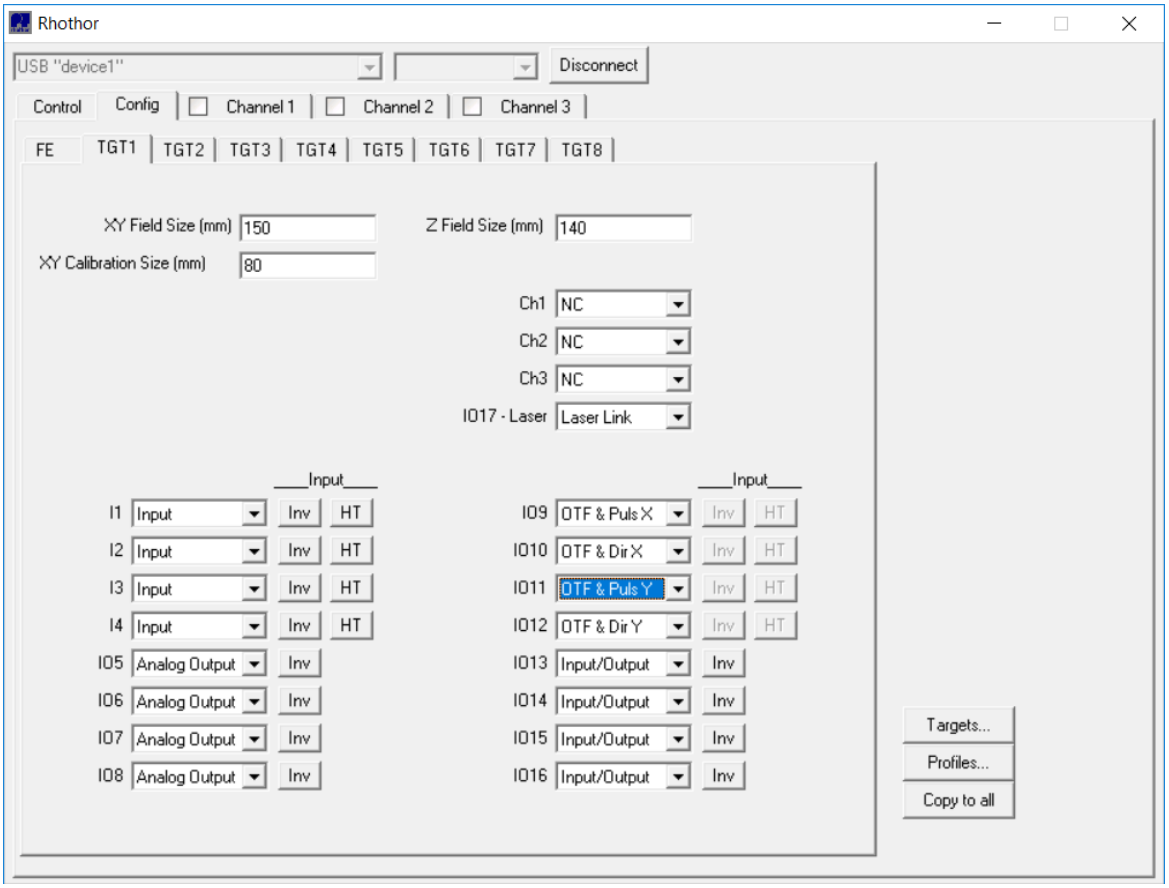
The CUA32 controls XYZ stages in a fashion that is similar than deflectors. When table instructions (*rtTableMoveTo…*) are parsed, the table setpoints change and step pulses are invoked. As with deflectors, low pass filtering is used to limit frequency changes. The applied step frequency needs to rise slowly so the steppers can keep up. As a deflector, they need time to accelerate the load during the movement. By low pass filtering the steppers setpoint explicit ramping math can be avoided. The resulting straight forward command sequencing is a huge advantage. Command chains could stall in streaming mode. The low pass filtering avoids table hard stops when this happens. The application needs to set several table parameters. Table speed and table delay are correlated. Table delay (*rtSetTableDelay*) is the time the steppers get to achieve their maximal speed (*rtSetTableMaxSpeed*).

When the CUA32 controller is used exclusive to control table stages, understanding position errors induced by the low pass filter becomes important. When a command like *rtTableMoveTo* terminates, the table isn't at the destination location. The low pass filter delays the axis setpoint signal. The table behaves like a deflector, but the time constant (*rtSetTableDelay*) is much higher. When accurate tracking of setpoints is required, the application can use similar tools as with deflectors (*rtSleep*, *rtBurst*) to improve cornering. When the XY-stage is used for circle tracking, the diameter of the setpoint circle can be slightly increased to compensate for the attenuation effect. Circle markings

could also be preceded by an arc move to improve the roundness. The need of all this depends on the speed and the bandwidth at which the application wants to control the table. When the system has deflectors all said errors can be autocorrected using the on board on-the-fly support.

open loop tracking

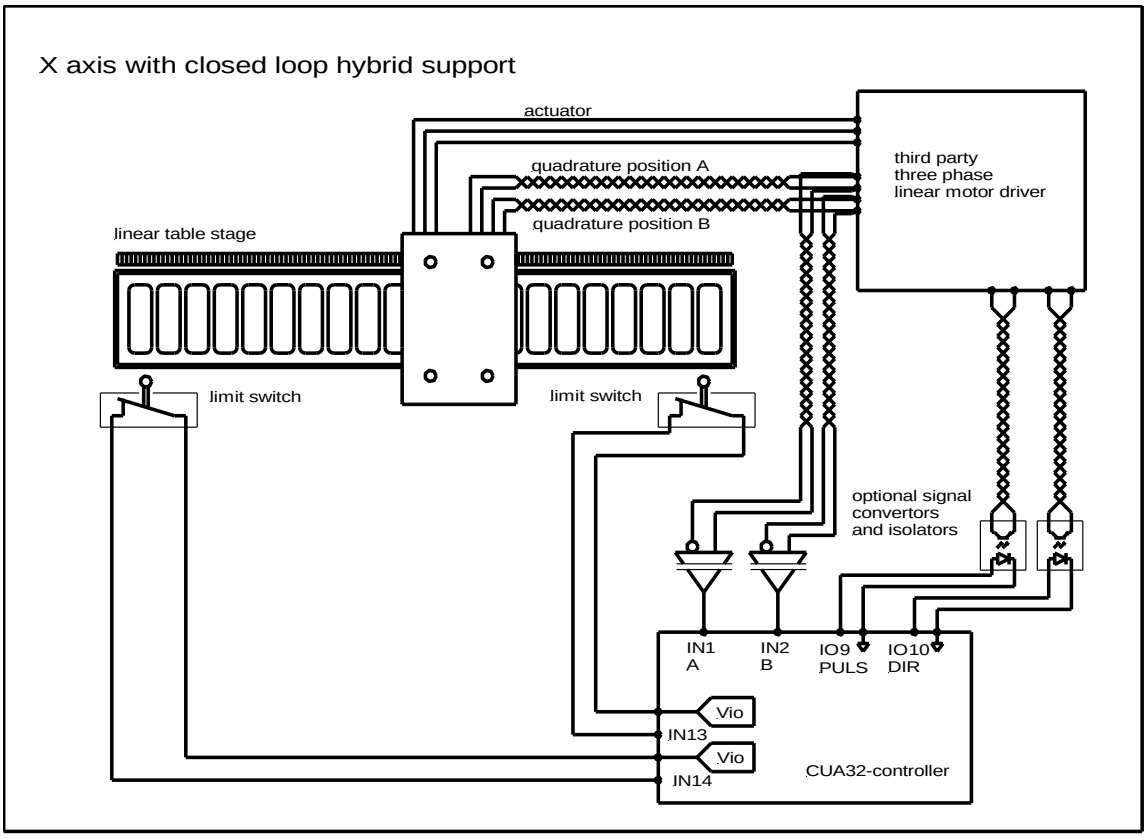
Any XY-stage driven by steppers can be seen as a zero-delay system. Any applied PULS signal is executed by the stepper in theory without delay. Said pulses also need to be applied timely by the controller. When the step frequency needs to be altered, this alteration needs to be smooth allowing the motor to keep up. The table actual setpoint can be considered equal to its setpoint which can be used as on the fly inputs during marking. The Following screen-print shows a configuration wherein the on-the-fly positions will be overwritten by table positions.



When image processing should be done using both deflectors and table stages, hybrid marking can be switched on and parameterized. This implementation of hybrid marking is called open loop because there is no actual feedback. The on-the-fly positions are overwritten with table setpoint positions. There is no fail safe in case the stepper loses steps neither is there a compensation for friction induced backlash.

closed loop tracking

Servo drives will likely add an additional delay. As such the generated pulses cannot be used to estimate the on-the-fly position and actual measurements are needed. Most servos provide access to phase shifted signals for position tracking. When not, additional measuring strips should be mounted. The quadrature signals generated by said systems can be connected the CUA32 device to serve as on-the-fly inputs.



When image processing should be done using both deflectors and stages the resolver input and hybrid marking can be switched on and parameterized. This implementation is called closed loop hybrid marking because the CUA32-controller uses the measured actual position as input. This closed loop hybrid solution comprises more wiring but is likely to have a higher accuracy.

processing on-the-fly resolver calibration

When combined with on-the-fly, the CUA32-TGT is hardware wired to see the on-the-fly input and table axis as one coordinate system. They may have different step sizes and reference points but the controller expects that positions shown on both table position counter and on-the-fly input counter should be about the same. When resolver calibration is loaded, its mapped error data will be added to the setpoints in all table commands (*rtTableLineTo, rtTableArcTo,..*) before execution to counter offset the resolver errors.

Assume the example as illustrated in chapter 7 (on-the-fly resolver calibration) comprising X-Y table stages in open loop tracking (overwriting on-the-fly positions with table positions). The application wants to set the laser spot at (1000,500). For said position, the loaded resolver calibration map returns offset vector (2,-1).
Command sequence:
…
rtSetOTF(1,0); // switch off on-the-fly tracking X-axis
rtSetOTF(2,0); // switch off on-the-fly tracking Y-axis
rtTableMoveTo(1000,500), // table and on-the-fly resolver count will go to (1002,499)
rtSetOTF(1,1); // switch on on-the-fly-tracking X-axis
rtSetOTF(2,1);); // switch on on-the-fly tracking Y-axis
rtJumpTo(1000,500); // scanner will jump to its center position
…
Why the latter:
As illustrated by the schematic in chapter “deflector control”, the resolver positions including their errors are compensated by the deflectors. Tanks to counter steering the table with the resolver error data, the calibrated resolver count equals the parameters of *rtTableMoveTo* when positioned.
deflector = setpoint command – (on the fly offset count - resolver calibration)
Xdeflector = 1000 - (1002-2) = 0
Ydeflector = 500 - (499+1) = 0

9. interlock

THE INTERLOCK FUNCTIOANLITY PROVIDED BY THE CUA32 DEVICES MAY NOT BE USED AS A SAFETY SYSTEM.

It may be desirable to stop a marking when a table axis hits a limit switch, a deflector reports an error or some other signal becomes invalid. The application can define a system interlock through the functions rtSetWhileIO and rtSystemSetWhileIO. When the go directives or no longer met, the system automatically stops. After solving the issue, the interlock must be rearmed by calling rtAbort or rtReset. Only persistent function calls can be used during an interlock error.

10. library functions

The applicability of the DLL functions depends on the state of the CUA32 device. Some functions are queued and must be embedded in a *rtListOpen*/*rtListClose* sequence while others can be called at any time. Commands are binary formatted for exchange with the hardware. While most are 8 bytes long, detailed knowledge of their sizes could be of interest when queued sequences are to be stored locally. The CUA32-MST device can be extended with several slave devices. The DLL uses *rtSetTarget* and *rtSetQueryTarget* as route directives for the functions. Some are dedicated to the CUA32-FE controller and some are routed to all CUA32-TGT devices regardless of target setting. Following table gives an overview of applicability, size, target aiming and scope for each library function.

applicability

Powered	The function is applicable whenever the system is switched on.
Connected	The function is applicable whenever the system is connected with the application.
Idle	The function is applicable when the system is connected and idle.
Closed	The function is applicable when the system is connected and the command queue is closed. This differs from the idle state because the system could still be executing the previous loaded commands.
compile or stream	The function is applicable when the system is connected and a command queue was opened using *rtListOpen(x)* or *rtFileOpen(x).*
Compile	The function is applicable when the system is connected and a command queue was opened in compile mode *rtListOpen(1), rtListOpen(3)* , *rtListOpen(5)* or *rtFileOpen(x)*

target aiming

FE	The function is being processed by the CUA32-FE controller.
all	The function is forwarded to all target controllers.
mask	The function is forwarded as specified by the last *call rtSetTarget(mask)*. Target indexes that have a 0 bit in mask will ignore the command. Bit 0 in mask corresponds with target 1, bit 1 with target 2 and so on.
single	The function is forwarded as specified by the last *call rtSetTarget(mask)*. However only the least significant non-zero bit in mask will select a target. All other one bits will be ignored.
FE/mask	Execution is allocated based on the used list mode. When called within mode 1 or 3, execution is handled by the CUA32-FE controller. When called within mode 5 the function is allocated as specified by the last *call rtSetTarget(mask)*.
query	The function queries the target selected by the last call *rtSetQueryTarget.*

scope

abortable	The function stops on interlock errors and after calling *rtAbort().*
persistence	The function tries to execute regardless the interlock state.

function table

Name	applicability	size	target	scope
rtAbort	connected		all	persistent
rtAcceptData	compile or stream	8	mask	abortable
rtAddCalibrationData	idle		single	persistent
rtArcMoveTo	compile or stream	16	mask	abortable
rtArcTo	compile or stream	16	mask	abortable
rtBurst	compile or stream	8	mask	abortable
rtCharDef	compile	16	mask	abortable
rtCircle	compile or stream	16	mask	abortable
rtCircleMove	compile or stream	16	mask	abortable
rtDoLoop	compile	8	FE/mask	abortable
rtDoWhile	compile	16	FE/mask	abortable
rtElse	compile	8	FE/mask	abortable
rtElseIfIO	compile	24	FE/mask	abortable
rtEndIf	compile	0	FE/mask	abortable
rtEraseFromFlash	idle		FE	persistent
rtFileClose	compile		FE	persistent
rtFileCloseAtHost	compile		FE	persistent
rtFileCloseAtIndex	compile		FE	persistent
rtFileDownload	Idle		FE	persistent
rtFileFetch	compile or stream		FE	persistent
rtFileOpen	closed		FE	persistent
rtFileUpload	idle		FE	persistent
rtFileUploadAtIndex	idle		FE	persistent
rtFontDef	compile	2056	mask	abortable
rtFormatFlash	idle		FE	persistent
rtGetAnalog	connected		query	persistent
rtGetCanLink	connected		FE	persistent
rtGetCfgIO	connected		query	persistent
rtGetCounter	connected		query	persistent
rtGetDeflReplies	connected		query	persistent
rtGetFieldSize	connected		query	persistent
rtGetFieldSizeZ	connected		query	persistent
rtGetFileIndex	connected		FE	persistent
rtGetFirstFreeUSBDevice	powered		FE	persistent
rtGetFlashFirstFileEntry	connected		FE	persistent
rtGetFlashMemorySizes	connected		FE	persistent
rtGetFlashNextFileEntry	connected		FE	persistent
rtGetID	connected		FE	persistent
rtGetIO	connected		query	persistent
rtGetIP	connected		FE	persistent
rtGetLaserLink	idle		query	persistent
rtGetMaxSpeed	connected		query	persistent
rtGetNextFreeUSBDevice	powered		FE	persistent
rtGetQueryTarget	connected		FE	persistent

Name	applicability	size	target	scope
rtGetResolvers	connected		query	persistent
rtGetScannerDelay	connected		query	persistent
rtGetSerial	connected		FE	persistent
rtGetSetpointFilter	connected		query	persistent
rtGetStatus	connected		FE	persistent
rtGetTablePositions	connected		query	persistent
rtGetTarget	connected		FE	persistent
rtGetVersion	connected		FE	persistent
rtIncrementCounter	compile or stream	8	mask	abortable
rtIndexFetch	compile or stream	8	FE	abortable
rtIfIO	compile	16	FE/mask	abortable
rtJumpTo	compile or stream	8	mask	abortable
rtLineTo	compile or stream	8	mask	abortable
rtLineTo3D	compile or stream	16	mask	abortable
rtLineToXD	compile or stream	24	mask	abortable
rtListClose	compile or stream		all	abortable
rtListOpen	closed		all	persistent
rtLoadCalibrationFile	idle		single	persistent
rtMoveTo	compile or stream	8	mask	abortable
rtMoveTo3D	compile or stream	16	mask	abortable
rtMoveToXD	compile or stream	24	mask	abortable
rtOpenCanLink	compile or stream	48	FE	abortable
rtParse	x		x	x
rtPowerProfileTo	compile or stream	8+n	mask	abortable
rtPrint	compile or stream	8+n	mask	abortable
rtPulse	compile or stream	8	mask	abortable
rtReset	connected		all	persistent
rtResetCalibration	idle		single	persistent
rtResetCounter	compile or stream	8	mask	abortable
rtResetResolver	compile or stream	8	mask	abortable
rtRunServer	x		x	x
rtScanCanLink	compile or stream	8	FE	abortable
rtSelectDevice	powered		FE	persistent
rtSetAnalog	compile or stream	8	mask	abortable
rtSetCanLink	compile or stream	8+n	FE	abortable
rtSetCfgIO	compile or stream	8	mask	abortable
rtSetCounter	compile or stream	8	mask	abortable
rtSetFieldSize	compile or stream	8	mask	abortable
rtSetImageMatrix	compile or stream	32	mask	abortable
rtSetImageOffsRelXY	compile or stream	8	mask	abortable
rtSetImageOffsXY	compile or stream	8	mask	abortable
rtSetImageOffsZ	compile or stream	8	mask	abortable
rtSetImageRotation	compile or stream	8	mask	abortable
rtSetIO	compile or stream	8	mask	abortable
rtSetJumpSpeed	compile or stream	8	mask	abortable
rtSetLaser	compile or stream	8	mask	abortable
rtSetLaserFirstPulse	compile or stream	8	mask	abortable
rtSetLaserLink	compile or stream	8	mask	abortable
rtSetLaserTimes	compile or stream	8	mask	abortable

Name	applicability	size	target	scope
rtSetLoop	compile	8	FE/mask	abortable
rtSetMatrix	compile or stream	32	mask	abortable
rtSetMinGatePeriod	compile or stream	8	mask	abortable
rtSetOffsIndex	compile or stream	8	mask	abortable
rtSetOffsXY	compile or stream	8	mask	abortable
rtSetOffsZ	compile or stream	8	mask	abortable
rtSetOscillator	compile or stream	8	mask	abortable
rtSetOTF	compile or stream	8	mask	abortable
rtSetPower	compile or stream	8	mask	abortable
rtSetPowerLevels	compile or stream	8	mask	abortable
rtSetPowerProfile	compile or stream	8	mask	abortable
rtSetPulseBulge	compile or stream	8	mask	abortable
rtSetQueryTarget	connected		FE	persistent
rtSetResolver	compile or stream	8	mask	abortable
rtSetResolverCal	idle		single	persistent
rtSetResolverPosition	compile or stream	8	mask	abortable
rtSetResolverRange	compile or stream	8	mask	abortable
rtSetRotation	compile or stream	8	mask	abortable
rtSetScale	compile or stream	8	mask	abortable
rtSetSpeed	compile or stream	8	mask	abortable
rtSetTable	compile or stream	8	mask	abortable
rtSetTableDelay	compile or stream	8	mask	abortable
rtSetTableLimitSwitches	compile or stream	8	mask	abortable
rtSetTableMaxSpeed	compile or stream	8	mask	abortable
rtSetTableRange	compile or stream	8	mask	abortable
rtSetTableSnapSize	compile or stream	16	mask	abortable
rtSetTableSnapSizeEx	compile or stream	16	mask	abortable
rtSetTableSpeed	compile or stream	8	mask	abortable
rtSetTableStepSize	compile or stream	8	mask	abortable
rtSetTarget	connected		FE	persistent
rtSetVarBlock	compile or stream	8	mask	abortable
rtSetWhileIO	compile or stream	8	mask	abortable
rtSetWobble	compile or stream	24	mask	abortable
rtSetWobbleEx	compile or stream	24	mask	abortable
rtSetWobbleMode	compile or stream	8	mask	abortable
rtSleep	compile or stream	8	mask	abortable
rtStoreCalibrationFile	idle		single	persistent
rtSuspend	compile or stream	8	mask	abortable
rtSynchronise	compile or stream	64	all	abortable
rtSystemResume	connected		all	persistent
rtSystemSetIO	connected		mask	persistent
rtSystemSuspend	connected		all	persistent
rtSystemTableMove	connected		all	abortable
rtSystemTableMoveRel	connected		all	abortable
rtSystemTableStop	connected		all	persistent
rtSystemUartOpen	connected		FE	persistent
rtSystemUartWrite	connected		FE	persistent
rtSystemUDPsend	connected		FE	persistent
rtTableArcTo	compile or stream	16	mask	abortable

Name	applicability	size	target	scope
rtTableJog	compile or stream	8	mask	abortable
rtTableLineTo	compile or stream	8	mask	abortable
rtTableMove	compile or stream	8	mask	abortable
rtTableMoveTo	compile or stream	8	mask	abortable
rtUartRead	connected		FE	persistent
rtVarBlockFetch	compile or stream	8	FE	abortable
rtWaitCanLink	compile or stream	8	FE	abortable
rtWaitIdle	compile or stream	8	single	abortable
rtWaitIO	compile or stream	8	FE/mask	abortable
rtWaitResolver	compile or stream	8	mask	abortable
rtWaitStall	compile or stream	8	single	abortable
rtWhileIO	compile	8	FE/mask	abortable

rtAbort();

This function closes the current command list, purges all commands already stored in hardware and forces the system and laser in idle state. The function also restores the system interlock.

rtAcceptData(long DataType);

The functions *rtSetVarBlock*, *rtAcceptData* and *rtVarBlockFetch* combined with font definition functions provide a toolset to implement a marking solution for variable text. The character set (font) is constant and downloaded on the RAM disk of the target controller prior to the text marking. A ping-pong storage system for the variable text allows concurrently reading and writing. *rtAcceptData* adds an accept data event to the command list. When executed the ping-pong buffers are exchanged.

parameters:
DataType : reserved for future use

rtAddCalibrationData(const char* FileName);

This function call adds offset data to the target's deflector calibration stored in flash.

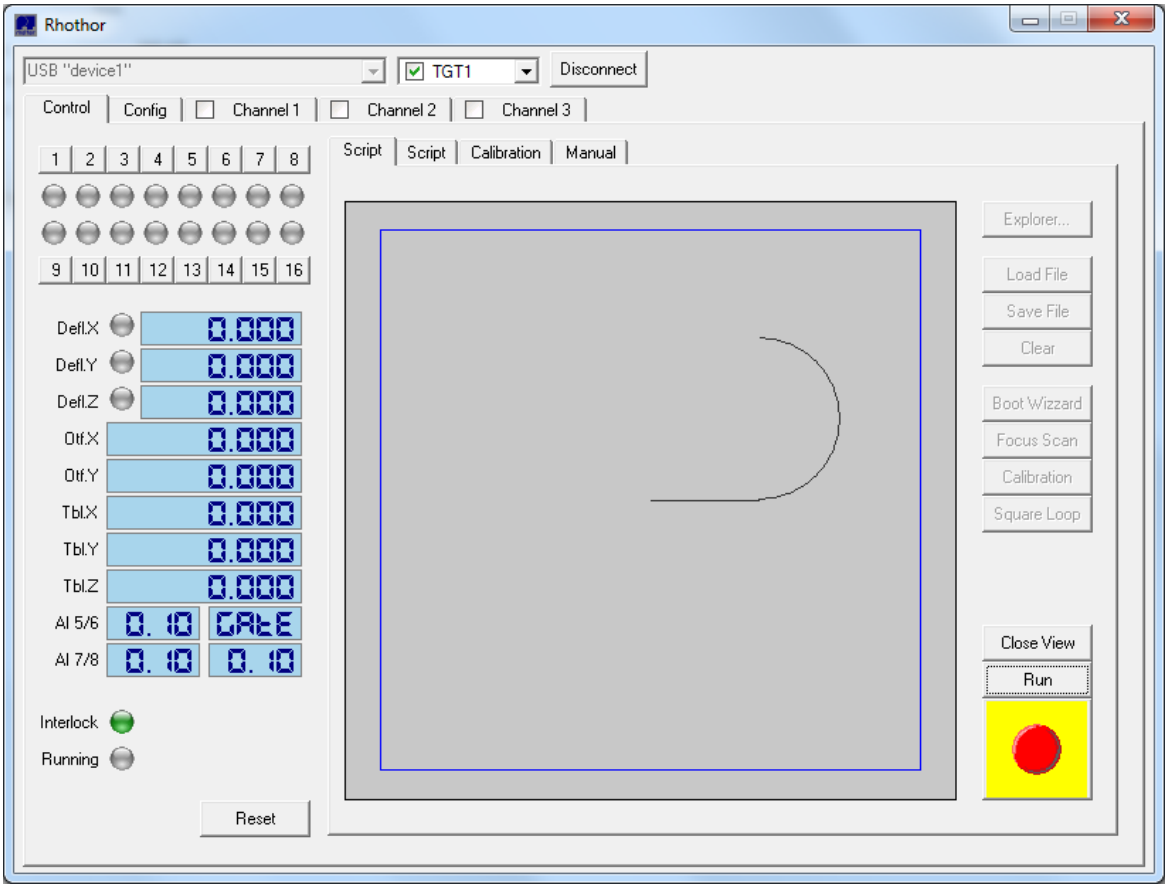
parameters:
FileName : name of the text file holding the offset data

rtArcMoveTo(double X, double Y, double BF);
rtArcTo(double X, double Y, double BF);

The function *rtArcTo* queues an arc wise marking while rtArcMoveTo queues an arc wise move (laser off). The arc starts at the current position and goes to coordinate (*X, Y*) using the BF as bulge factor. The bulge is the tangent of 1/4 the included angle for an arc segment, made negative if the arc goes clockwise from the start point to the end.

parameters:
X,Y: target position, range -8388.608...8388.607 mm
BF: bulge factor

Example:
```
rtListOpen(1)
rtJumpTo(0,0)
rtLineTo(20,0)
rtArcTo(20,30,1)
rtListClose()
```



After the example code was loaded into the script editor, the rhothor executable simulates the marking by pressing the View button. The arcs center point lies at (20,15) and runs counter clockwise over an angle of 180 degrees. (*BF* = tangent(45°))

rtBurst(long Time);

This function adds a burst command to the list. During execution of a burst, the gate signal is activated while the motors are standing still. The rtBurst function can be used to increase corner sharpness. Hovering the setpoint at the corner buys the deflectors more time to reach the corners position.

parameters:
Time : hover time, range 0...2147483647 µsec

rtCircle(double X, double Y, double Angle);
rtCircleMove(double X, double Y, double Angle);

The function *rtCircle* queues a circle marking while *rtCircleMove* queues a circular movement. The parameters *X* and *Y* define the center point of the circle. The Radius is defined as the distance between the center point and the current position. The angle is positive for counter clockwise and negative for clockwise marking.

parameters:
X,Y : center position, range -8388.608...8388.607 mm
Angle : degrees

Example:

```
rtListOpen(1)
rtJumpTo(0,0)
rtLineTo(20,0)
rtCircle(20,15,270)
rtListClose()
```

rtEraseFromFlash(const char* FileName);

Function erases a file from flash. All file names used on the CUA32 are case sensitive. When file is not found, ERR_OK is returned.

rtFileClose();
rtFileCloseAtHost();
rtFileCloseAtIndex(long Index);

The function *rtFileCloseAtIndex* closes the command list and stores the complete list to flash memory at the designated sector. The CUA32 flash is formatted to 250 sectors. 249 of them are available for saving files. The first sector, with index 0, is reserved to hold boot start code. Each sector is 256 Kbyte in size. When a command list doesn't fit in a single sector, it continues in the next. The application should verify that the complete command list can be stored in consecutive free flash memory. CUA32-FE flash memory management is left to the library when invoking *rtFileClose*. The function *rtFileCloseAtHost* compiles and saves command list as a binary file on the host computer. The file can be uploaded afterwards using *rtFileUpload* or *rtFileUploadAtIndex*.

parameters:
Index : sector number

rtFileDownload(const char* FileName, const char* DestFile);

This function copies a file stored on flash to the host.

parameters:
FileName : flash file name (zero terminated string)
DestFile : destination file name flash file name (zero terminated string)

rtFileFetch(const char* FileName);

This command adds all commands comprised by the file on the system's flash to the command list.

parameters:
FileName : zero terminated string

rtFileOpen(const char* FileName);

rtFileOpen allocates memory on the host computer and opens the list for **compilation**. The host maintains the complete command chain during its construction. No list commands are sent to the device. In this mode the instruction set becomes extended with control flow commands. When *rtFileClose* is called, the list is compiled and saved on the CUA32 flash. When *rtFileCloseAtHost* is called, host memory is used.

parameters:
FileName: file name (zero terminated string, length < 244 bytes)

rtFileUpload(const char* SrcFile, const char* FileName);
rtFileUploadAtIndex(const char* SrcFile, const char* FileName, long Index);

The function *rtFileUploadAtIndex* copies a file from host to flash at specified sector. When a file doesn't fit in a single sector, it continues in the next. The application should verify that the all the data from the source file can be stored in consecutive free flash memory. The CUA32 flash is formatted to 250 sectors. 249 of them are available for saving files. The first sector, with index 0, is reserved to hold boot start code. Each sector is 256 Kbyte in size. CUA32-FE flash memory management is left to the library when invoking *rtFileUpload*.

parameters:
SrcFile : source file name (zero terminated string)
FileName : flash file name (zero terminated string, length < 244 bytes)

rtFontDef(const char* Name);
rtCharDef(long Ascii);
rtFontDefEnd();

List commands can be structured into fonts. All commands submitted between *rtFontDef* and *rtFontDefEnd* are not executed but stored on the target's flash drive. An ascii based index scheme can be set up by using *rtCharDef* headers. The CUA32 allows definitions of fonts containing up to 255 characters. When the command rtFontDefEnd is executed, the font is fully downloaded into the target controller and ready to be used. The front end only needs to send ascii strings when executing commands like *rtPrint.* The target fetches the commands while marking a character from its local flash dramatically reducing communication overhead.

parameters:
Name : Font name, ignored because only one font can be stored on the target's RAM-disk.
Ascii : ascii code 1…255
The ascii code 0 is not available because the 0 character is used to mark the end of a string.

rtFormatFlash();

This function deletes all files, including the boot start file, from the flash. Calibration data and system settings, also stored on the flash, are maintained.

rtGetAnalog(long Nr, long* Value);

Function samples the analog voltage on the selected IO pin

parameters:
Nr =5: query voltage on IO5
Nr =6: query voltage on IO6
Nr =7: query voltage on IO7
Nr =8: query voltage on IO8
Value : place holder to store the sample (mV)

rtGetCanLink(long Address, long* Value);

CANopen devices use PDO messages to report changes. Messages of interest, selected using *rtScanCanLink*, are logged by the CUA32-FE controller. The application can read back message data using this function.

Parameters:
Address : 0…7, data in first log buffer
Address : 8…15, data in second buffer
Value: place holder to store the byte

rtGetCfgIO(long Nr, long *Value);

The function queries the configuration setting of the selected IO pin. IO functionality and their respective configuration setting can be depicted from the rhothor.exe configuration page. The returned value equals the IO pin's drop-down list index.

parameters:
Nr : 1,2...,17
Value : placeholder to store configuration setting

rtGetCounter(long* Value);

The function queries the target's counter value. This counter is altered when *rtIncrementCounter* or *rtSetCounter* is executed. Querying this counter allows the host to determine which commands are executed and which commands are still in the queue.

parameters:
Value : place holder to store the target's counter.

rtGetDeflReplies(long* CH1, long* CH2, long* CH3);

The function returns the setpoints (Xdefl,Ydefl,Zdefl) of the deflectors. The chapter *six axis numerical control system* from this document explains how these values are calculated.

parameters:
CH1 : place holder for X deflector setpoint (µm)
CH2 : place holder for Y deflector setpoint (µm)
CH3 : place holder for Z deflector setpoint(µm)

rtGetFieldSize(double* Size);
rtGetFieldSizeZ(double* Size);

The function *rtGetFieldSize* returns the field size of the XY deflection system. The function rtGetFieldSizeZ returns the focal range of the Z axis.

parameters:
Size : place holder to store the target's field size (mm)

rtGetFileIndex(const char* FileName, long* Index);

This function searches the flash for a file with the declared *FileName*. When found, its index is returned. When not found, index is set to -1.

parameters
FileName : file name to be searched (zero terminated string)
Index : place holder to receive file index

rtGetFirstFreeUSBDevice(char* Name);
rtGetNextFreeUSBDevice(char* Name);

The CUA32-FE operating system support addressable USB connectivity. Each device can be given a unique name allowing to application to target its connection requests, regardless of the physical connector. The function *rtGetFirstFreeUSBDevice* start searching the USB tree for the first available CUA32 device. When found, its USB ID string is returned. This string can be set by the rhothor executable. By concussively calling *rtGetNextFreeUSBDevice* a complete list of available CUA32 devices can be obtained. A USB ID string is a zero-terminated string with a total length limited to 64 bytes. When no more free devices are found, a zero string is returned. The returned strings serve as an address to select which USB device to open (*rtSelectDevice*).

parameters
Name : 64-byte place holder to receive the USB ID string (zero terminated string)

rtGetFlashFirstFileEntry(char* Name, long* Size);
rtGetFlashNextFileEntry(char* Name, long* Size);

With these functions an application can query the CUA32-FE flash content. Obtaining the directory starts by calling *rtGetFlashFirstFileEntry* followed by successively calling *rtGetFlashNextFileEntry* until the returned *Name* equals a zero string.

parameters
Name : 244-byte place holder to receive file name (zero terminated string)
Size : place holder to receive file size

rtGetFlashMemorySizes(long* Total, long* Allocated);

This function returns both total and allocated flash memory sizes. The CUA32 master device is fitted with a flash disk comprising 250 sectors each 256 Kbyte in size.

parameters
Total, Allocated : placeholders to store the memory sizes (bytes)

rtGetID(char* Name);

This function returns the "USB ID" string as specified in the rhothor configuration program.

parameters
Name : 64-byte place holder for the USB ID string (zero terminated string)

rtGetIO(long* Value);

Function returns the current digital levels sampled on the IO's of the selected target. Bit 0 holds value of IO1, bit 1 of IO2.... The bit values related with the analog IO's are calculated. On those IO's the analog voltage is sampled and compared with midscale value (2.5V). When higher a one bit will be returned.

parameters:
Value : place holder to store the sample

rtGetIP(char* Mac, char* IP);

When host connection is done over USB, the ethernet connector can be used to send UDP commands over the internet. The devices can be directly connected or connected over a switch. To map IP with MAC addresses, the CUA32 master controller maintains an ARP-cache containing four entries. The MAC addresses are device specific and known. The *rtGetIP* function allows the application to obtain their IP address needed for internet communication.

parameters:
Mac : “dd.dd.dd.dd.dd.dd” (zero terminated string, dd: “0”…”255”)
IP : 16-byte place holder to receive IP address (IPV4)

rtGetLaserLink(long Address, long* Value);

Function queries the laser link connected to the target controller.

parameters:
Address : 0x80...0xFF
Value : place holder to store the reply.

rtGetMaxSpeed(double* Speed);

The function returns the maximal speed of the target controller. The maximal speed is set by to 100 times the field size / sec.

parameters:
Speed : placeholder to store the speed (mm/sec)

rtGetQueryTarget(long* Index);

This function returns the index, 1 to 8, of the current query target (*rtSetQueryTarget*).

parameters
Index : place holder for current query target

rtGetResolvers(double* X, double* Y);

System returns the uncalibrated on the fly positions for both X and Y axis in mm. (resolution 1 µm)

parameters:
X,Y: placeholders to store the on-the-fly-offset counters (mm)

rtGetScannerDelay(long* Delay);

Rhothor deflectors can be customer tuned to any delay between 60 and 350 µsec. Shorter delays will increase system bandwidth while fast operational speeds require a longer delay setting for the deflection system. The setting and tuning are done through the rhothor executable. The result is queried using this function. When combined with *rtGetSetpointFilter* the application can calculate the theoretical value for the laser delay (*rtSetLaserTimes*):
laser on delay = setpoint filter + scanner delay – laser rise time
laser off delay = setpoint filter + scanner delay – laser fall time

parameters:
Delay : place holder for the deflector delay (µsec).

rtGetSerial(long* Serial);

This function returns the serial number of the CUA32-FE device.

rtGetSetpointFilter(long* TimeConst);

This function returns the time constant of the setpoint filter. The setpoint filter can be set using the rhothor executable. This function combined with *rtGetScannerDelay* allows the application to calculate the theoretical values for the laser delay (*rtSetLaserTimes*).

parameters:
TimeConst : place holder for the setpoint filter time constant (µsec)

rtGetStatus(long* Memory);

This function returns ERR_BUSY (2) when there are still commands waiting for execution. When the command queue is empty, the command returns ERR_OK (-1) . When the parameter Memory is not NULL, the queue size is returned. In some cases, the application could use this size when adding commands. When the queue becomes too large, the application should suspend command generation to avoid running out of memory.

parameters:
Memory : placeholder to receive the estimated size of the command list (bytes)

rtGetTablePositions(double* X, double* Y, double* Z);

Function returns the actual table setpoint positions.

parameters:
X,Y,Z: place holders to store the stepper current setpoints (mm)

rtGetTarget(long* Mask);

This function returns the current target mask (*rtSetTarget*).

parameters
Mask : place holder for current mask

rtGetVersion(char* Version);

This function returns the version of the dynamic link library.

rtIfIO(long Value, long Mask);
rtElseIfIO(long Value, long Mask);
rtElse();
rtEndIf();

These functions add a control flow to the command list. Processing a *rtIfIO* command starts by waiting until all connected targets are idle. Afterwards the IO state of the least significant target is tested against the declared value. The comparison only considers those IO with their corresponding mask bit set to one. When the result returns equal, the instruction sequencing is continued until *rtElse* or *rtElseIfIO* is encountered. When the comparison returns false, the instruction sequencer continues at the next *rtElse, rtElseIfIO* or *rtEndIf* . These functions can only be used when the queue is opened in compile mode. When used in list mode 5 the waiting for idle is omitted.

parameters:
Value : 0...65535
Mask : 0...65535

rtIncrementCounter();

This function appends an increment counter event to the command list. Every target has a counter that can be controlled and queried through commands. The counter can be used by the application to determine which commands have been processed and which commands are still in the queue.

rtIndexFetch(long Index);

This command adds all commands stored in flash at the selected index to the command list.

parameters:
Index : 1...249

rtJumpTo(double X, double Y);
rtJumpTo3D(double X, double Y, double Z);

This command adds a jump to the command list. Jumps are executed by the target processors as a linear ramp at jump speed. Laser is switched to idle during command execution.

parameters:
X, Y, Z: target position, range -8388.608...8388.607 mm

rtLineTo(double X, double Y);
rtLineTo3D(double X, double Y, double Z);
rtLineToXD(double X, double Y, double Z, double TX, double TY, double TZ, long Mask);

This command adds a line marking to the command list. The line starts at current position and extends towards the target position. Marking speed (*rtSetSpeed*) is used during execution. All axes are synchronized and will reach the destination at the same time. The function *rtLineToXD* has an additional parameter *Mask* to control which axes to move. Target positions of disabled axis's are ignored.

parameters:
X, Y, Z: deflector target position, range -8388.608...8388.607 mm
TX, TY, TZ: table target position, range -8388.608...8388.607 mm
Mask: axis enable mask
Bit 0= true: move X deflector
Bit 1= true: move Y deflector
Bit 2= true: move Z deflector
Bit 3= true: move X table
Bit 4= true: move Y table
Bit 5= true: move Z table

rtListOpen(long Mode);
rtListClose();

With these commands an application controls the use of the CUA32 command queues.

parameters:

Mode =1: *rtListOpen(1)* allocates 256KB memory on the host computer and opens the list for **compilation**. The host maintains the complete command chain during its construction. No list commands are sent to the device. In this mode the instruction set becomes extended with control flow commands. The command *rtListClose* compiles, transfers, and starts execution of the list. In general, a single command occupies 8 bytes, so the size of the command list will be limited to 32000 commands.

Mode =2: *rtListOpen(2)* will open a list for **streaming**. The list operates like a FIFO memory. The application adds commands to the queue, while the CUA32 device retrieves and processes them. To maximize data bandwidth commands are gathered and send in batches to the CUA32 device. Each batch is 1024 bytes in size. When rtListClose is called, the last batch is padded with NOP commands before being send. The CUA device has 32-Kbyte dedicated dual port memory to hold the queue data. When the command list becomes larger, the library will extend the queue capacity using host memory. To avoid running out of memory, the application should regularly query the total size of the command queue (*rtGetStatus*).

Mode =3: *rtListOpen(3)* allocates 256KB memory on the host computer and open the list for **compilation** similar to *rtListOpen(1).* When rtListClose is called, the complete list is compiled and saved on the CUA32 flash as a boot start file. This file will be started whenever the CUA32 device is powered on. Like in list mode 1, the size of the command list is limited to 32000 commands.

Mode =4: *rtListOpen(4)* operates like *rtListOpen(1*) and is kept for compatibility reasons.

Mode =5: *rtListOpen(5)* operates much like *rtListOpen(1).* While the latter uses fifo memory from front end and target devices, lists opened in mode 5 will be exclusively stored on the targets. Changing target (*rtSetTarget*) or invoking queue functions targeting all or the CUA32-FE are not allowed in this mode. The execution of conditional flow commands will run faster because the target can access all IO data locally.

rtLoadCalibrationFile(const char* FileName);

This function copies the file content, containing the calibration, in the systems flash memory.

rtMoveTo(double X, double Y);
rtMoveTo3D(double X, double Y, double Z);
rtMoveToXD(double X, double Y, double Z, double TX, double TY, double TZ, long Mask);

This command adds a linear movement to the command list. The movement starts at current position and extends towards the target position. During the movement laser is set to idle and the marking speed is used. All axes are synchronized and will reach the destination at the same time. The function *rtMoveToXD* has an additional parameter *Mask* to control which axes to move. Target positions of disabled axis's are ignored.

parameters:
X, Y, Z: deflector target position, range -8388.608...8388.607 mm
TX, TY, TZ: table target position, range -8388.608...8388.607 mm
Mask: axis enable mask
Bit 0= true: move X deflector
Bit 1= true: move Y deflector
Bit 2= true: move Z deflector
Bit 3= true: move X table
Bit 4= true: move Y table
Bit 5= true: move Z table

rtOpenCanLink(long Baudrate);

This function queues a configuration event. When executed, the front-end controller initializes its CAN interface . Knowledge of this protocol is needed for proper understanding. The CUA32-FE uses a CANopen as an NMT master device. When executed it sets the baudrate and broadcasts the CANopen wake up message.

parameters:
Baudrate: bit/sec

rtParse(const char* Cmd);

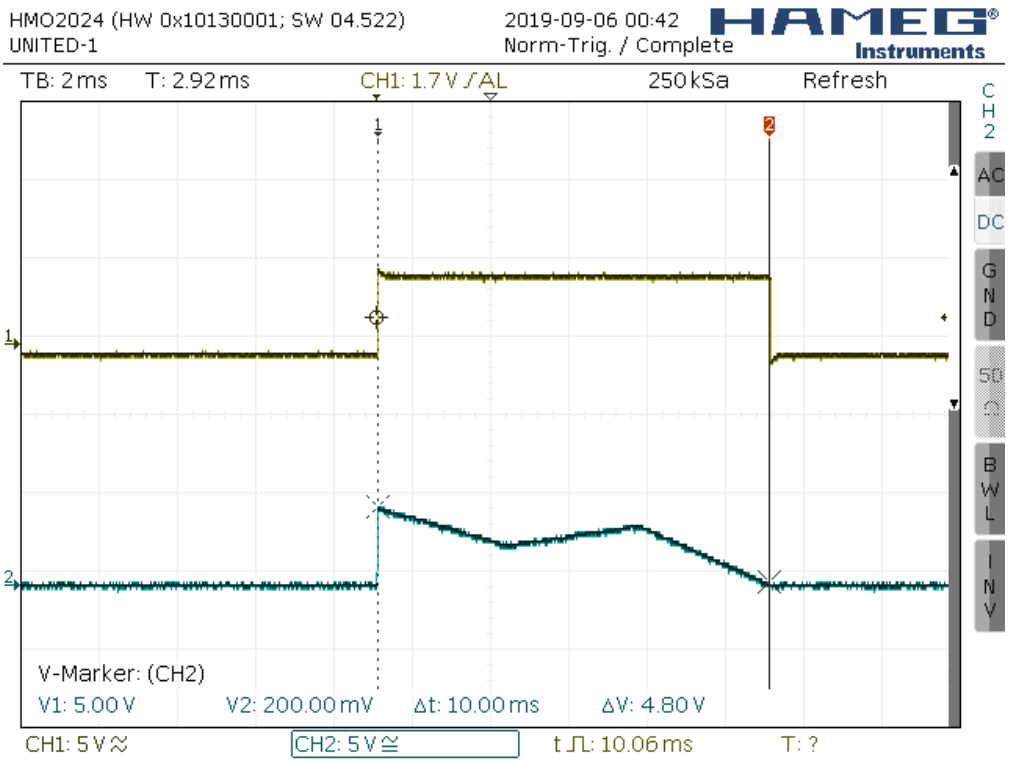
This function provides a string-based interface to the CUA32 controller. The *Cmd* command string is parsed and the resulting DLL functions is invoked. The string must be C-code compatible and may contain "//" sequence to mark comments. The command string should be terminated with a zero-byte preceded by a return and line feed sequence. The use of this function should be avoided because an application gains performance by calling library functions directly. On the other hand, flash files created using *rtParse* also contain the source code. This source can be read back for editing using the rhothor executable.

rtPowerProfileTo(double X, double Y, char* Pixels);

This function queues a line marking to the command list. When executed the target controller marks a line towards point (*X,Y*) starting from current position. The parameter *pixels* points to a hexadecimal string. After conversion to binary its content is used to set the laser power during the mark. A value 255 selects the 100% laser output setting while 0 drives the laser towards its idle setting. The function *rtSetPowerLevels* is provided to map power requirements with actual output data. Like with deflectors, every 10 µsec a new value is set. The CUA32-TGT controller uses a linear interpolation scheme to harmonize the available power data with the execution time of the mark.
Actual power steering output:
IO5 voltage=(*pwr*(*pwr100*-*pwr0*)/256 + *pwr0*)*5mV
PulseWidth=(*pwr*(*pwr100*-*pwr0*)/256 + *pwr0*)*(*Period*/1000) µsec

parameters:
X, Y: deflector target position, range -8388.608...8388.607 mm
Pixels: zero terminate hexadecimal string, maximal length including zero byte is 511 bytes

Example:
…
rtSetSpeed(1000)
rtJumpTo(0,0)
rtSetPowerLevels(1000,0)
rtPowerProfileTo(10,0,"FF80C000")
…



The hexadecimal pixel string in this example contains four binary values dividing the 10 milli second marking into three areas. Top line shows the gate signal and bottom line shows the voltage on IO5. The analog voltage starts at 5V (FF80C000) and goes down to 2.5V (FF80C000)

during the first, rises to 3.75V (FF80C000) during the second and goes to zero (FF80C000) during the last phase of the mark.

rtPrint(const char* data);

This function queues a constant string mark. When executed, the system will mark the string pointed to by parameter *data*. Prior to invocation a font must be uploaded to the CUA32-TGT devices (r*tFontDef, rtCharDef* and *rtFontDefEnd*).

parameters:
data : pointer to a zero terminated string. Its length, including the zero byte, must be smaller than 255 characters.

rtPulse(double X, double Y);
rtPulse3D(double X, double Y, double Z);

This command adds a pulse command to the list. Pulse commands are executed by starting a linear ramp with idling laser towards the target position. The last portion of the movement is done with activated laser signal. Duration of this pulse is determined by the setting of oscillator 3 and the selected laser mode (rhothor.exe). The speed used for ramping is calculated based upon the distance, the selected marking speed (*rtSetSpeed*) and minimal gate period time. (*rtSetMinGatePeriod*). When the distance towards target position is smaller than the minimal gate period, the ramping speed is reduced.

parameters:
X,Y,Z : target position, range -8388.608...8388.607 mm

rtReset();

This function resets the processing units of the CUA32-MST master and all connected CUA32-SLV slave devices. Running commands are aborted, command queues cleared and the system reconfigured. The USB and TCP communication stacks are maintained to keep up the connection. The boot start file on flash isn't started.

rtResetCalibration();

This function resets the target's calibration.

rtResetCounter();

This function appends a reset counter event to the command list.

rtResetResolver(long Nr);

This function adds a reset resolver command to the list. When the command is executed, the target controller will reset the selected on-the-fly counter.

parameters:
Nr =1: on-the-fly counter X axis
Nr =2: on-the-fly counter Y axis

rtRunServer(long Id, void* Params1, void* Params2);

reserved rhothor function

rtScanCanLink(long Address, long Node, long Index, long SubIndex);

This function queues a CANopen configuration. CANopen devices can be configured to report status changes to the network. The CUA32-FE scans all PDO traffic on the CANopen bus and stores messages of interest in local memory. The logged data can be queried by the application using *rtGetCanLink* or the CUA32-MST device can stall command processing until a specific message data has been received *rtWaitCanLink*.

parameter:
Address=0: log transfer data in first message 8 bytes buffer
Address=8: log transfer data in second 8 bytes buffer

Index=0: Setup scanning for PDO messages from node.
SubIndex=1: log PDO1 transfer from node
SubIndex=2: log PDO2 transfer from node
SubIndex=3: log PDO3 transfer from node
SubIndex=4: log PDO4 transfer from node

rtSelectDevice(const char* IP);

This method is used to setup a physical connection with a CUA32 device. Once connected all library functionality becomes available to the application. The device and the connection type are declared by the content of the IP string.

* example *IP*: *USB*
The connection with the CUA32 device will be made over USB. The USB tree will be searched for the first available CUA32 device.

* example *IP*: *USB "newson"*
The connection with the CUA32 device will be made over USB. The USB tree will be searched for a device with USB ID set to "newson".

* example *IP*: *TCP "172.16.224.20"*
The connection with the CUA32 device will be made over the ethernet connection using TCP-IP protocol and targeting IP address 172.16.224.20.

* example *IP*: *UDP "172.16.224.20"*
The connection with the CUA32 device will be made over the ethernet connection using UDP-IP protocol and targeting IP address 172.16.224.20.

rtSetAnalog(long Value, long Mask);

This function adds an analog alteration command to the command list. When executed, the target's analog outputs will be set to the desired value. To change, the analog output must be enabled (rhothor.exe) and the mask bit must be set. The target's DA convertors have a 5V full scale and a resolution of 12 bit.

parameters:
Value : voltage, range 0...5000 mV
Mask : IO bit 5,6,7 and or 8 must be set to change the analog output

rtSetCanLink(long Node, long Index, long SubIndex, char* Data);

This function queues a CANopen communication. Knowledge of this protocol and the accessed CAN device are needed for proper use. The CUA32-FE supports both PDO and SDO type messages. The parameter *Data* points to a hexadecimal string which is converted to binary prior to transmission. The parameters *Node*, *Index* and *SubIndex* are routing directives for the CANopen network.

parameters:
Node : device CANopen number
Index=0: PDO transfer mode
SubIndex=1: start PDO1 transfer
SubIndex=2: start PDO2 transfer
SubIndex=3: start PDO3 transfer
SubIndex=4: start PDO4 transfer
When the function is invoked with an invalid SubIndex (<1 or > 4) the message will be sent as a PDO1 message.

Index>0: SDO transfer mode
Depending on the size of data an expedited or segmented SDO transfers will occur. The parameters *Index* and *SubIndex* are used by the receiving node to route the data to the right object.

example: rtSetCanLink(16,0,0,”14000F”);
This function sends a PDO1 message (20,0,15) to node 16.

example: rtSetCanLink(16,12321,0,B1010B0101000000);
This function sends an SDO message (177,1,11,1,1,0,0,0) to node 16, object 12321

rtSetCfgIO(long Nr, long Value);

This function adds an IO configuration change in the command list. While the rhothor executable is commonly used, automatic application configuration can be implemented. IO functionality and their respective selection values can be depicted from the rhothor.exe configuration page. Using the rhothor software for configuration of IO's is easy. The program guarantees settings of paired configurations (on the fly, pulse & direction…) and configuration execution sequence. When using *rtSetCfgIO* calls for configuration, one has to make sure that the command calls make sense. When IO1 is configured as "OTF XA+", IO9 should be configured as "OTF XA-". Furthermore, all IO's should be configured in sequence for proper operation. First call *rtSetCfgIO* to configure IO1, then IO2 and continue until IO17. The value needed to select the function is the zero-based index from the drop-down list (rhothor executable). Threshold and polarization of the inputs can also be controlled using this function. Setting bit 16 in *Value* will select the high threshold while setting bit 17 will invert the input.

parameters:
Nr : 1:IO1, 2:IO2... 17:IO17
Value : function number

rtSetCounter(long Value);

This function appends a counter preload event to the command list. When executed the target's sequence counter is loaded with the desired value.

parameters:
Value : 0...65535

rtSetFieldSize(double Size);

This function adds a field size change in the command list. When this command is processed the scaling of the coordinates send to Ch1 and Ch2 deflectors is altered. The relation between system and deflector units is defined as follows:
deflector setpoint (bit) = (16640000/field size) * system setpoint (mm)

parameters:
Size : deflectors field size, range -8388.608...8388.607 mm

rtSetImageMatrix(double a11, double a12, double a21, double a22);

This function adds an image transformation matrix change event to the command list. When executed the target's image transformation (*ImgAij*) matrix is altered. An application can rotate and stretch images in the XY- plane using the transformation matrix.

parameters:
a11,a12,a21,a22 : factor

rtSetImageOffsRelXY(double X, double Y);

This function queues an offset change. When executed the declared offset values are transformed (*ImgAij*) and added to the image offset vector. This relative offset can be used to create fonts. A font is a set of images (characters). Each image contains line commands specifying the typeface and a *rtSetImageOffsRelXY* command. The latter defines the character width and can be used to shift the cursor to the next position.

parameters:
X,Y: relative image offset position, range -8388.608...8388.607 mm

rtSetImageOffsXY(double X, double Y);

This function queues an offset change. When executed the image offset vector (*ImgOffsX, ImgOffsY*) will be loaded with the declared values.

parameters:
X,Y: image offset position, range -8388.608...8388.607 mm

rtSetImageOffsZ(double Z);

This function queues an image Z-offset change. When executed the image Z-offset vector (*ImgOffsZ*) will be loaded with the declared value.

parameters:
Z: image offset position, range -8388.608...8388.607 mm

rtSetImageRotation(double Angle);

This function queues an image transformation matrix change. When executed the target's image transformation matrix (*ImgAij*) will be loaded with a rotation function. The function is the same as: *rtSetImageMatrix(cos(Angle),-sin(Angle),sin(Angle),cos(Angle))*

parameters:
Angle : rotation angle in radians

rtSetIO(long Value, long Mask);

This function adds an output change event to the command list. When executed by the target, the output values will be overwritten as follows:
bit 0 of “Value” is copied to the output bit of IO1 only when bit 0 of “Mask” is true,
bit 1 of “Value” is copied to the output bit of IO2 only when bit 1 of “Mask” is true….

parameters:
Value, Mask : range 0...65535

rtSetJumpSpeed(double Speed);

This function queues a configuration event. When executed, the jump speed will be set to *Speed*.

parameters:
Speed : speed in mm/s

rtSetLaser(bool OnOff);

This function queues a laser control event. During command processing, the laser is controlled together with the deflection motors. When *rtSetLaser(1)* is executed, the laser is activated continuously until explicitly switched off by calling *rtSetLaser(0)*. This functionality is useful for laser power calibration.

parameters:
OnOff =0: gate signal will be active during marking and switched off on idle.
OnOff =1: gate signal will always be on

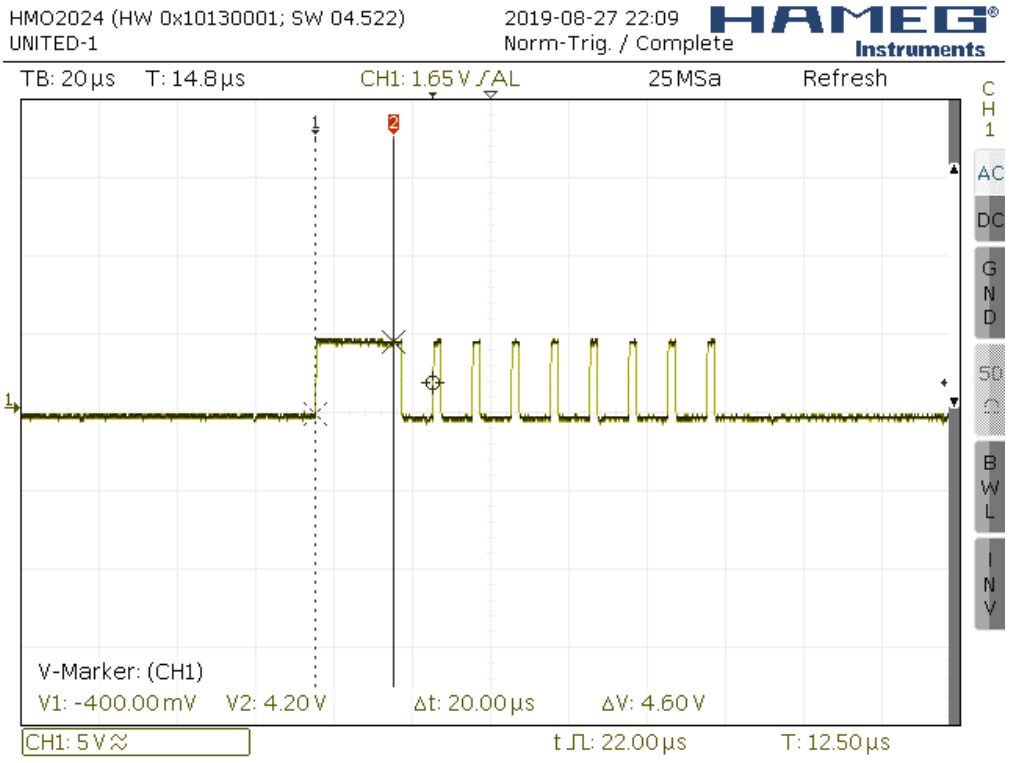
rtSetLaserFirstPulse(double Time);

This function queues a laser control event to the command list. Some lasers have power-up or first pulse suppression features controlled by a lead pulse preceding the actual signal burst. With this function the duration of said pulse can be set. This function is only available when the mode of IO17 was set to “Burst” (rhothor executable)

parameters
Time : duration of the first pulse in µsec.

Example:

```
…
rtSetOscillator(3,10,2); // set laser period 10 µsec and pulse width 2 µsec
rtSetLaserFirstPulse(20); // lead pulse 20 µsec
rtBurst(100); // just to see the signal
…
```



The first pulse measured when this script was run equals 22 µsec. This results from the concatenation of the 20 µsec wake up pulse with the first burst pulse.

rtSetLaserLink(long Address, long Value);

This function adds a laser link data upload event. When executed, the target will load the laser link's addressed register with *Value*. Execution takes 130 µsec during which the laser is idled.

parameters
Address : range 0...127
Value : range 0...255

rtSetLaserTimes(long GateOnDelay, long GateOffDelay);

This function adds a configuration event to the command list. Actual deflector positions are delayed in time. By shifting the laser signal, resynchronization with deflector movement is obtained. *rtGetScannerDelay* and *rtGetSetpointFilter* return the selected time constants from both, deflector and setpoint filter. When laser rise and fall times are known, *GateOnDelay* and *GateOffDelay* can be calculated. When not, a trial-and-error approach should be used. The calculation:
laser on delay = setpoint filter + scanner delay – laser rise time
laser off delay = setpoint filter + scanner delay – laser fall time

parameters:
GateOnDelay, GateOffDelay : 0...2047 µsec

rtSetLoop(long LoopCtr);
rtDoLoop();

These functions define a loop. All commands between *rtSetLoop* and *rtDoLoop* will be repeated. Parameter *LoopCtr* declares the number of executions. A zero value will create in an infinite loop. Loop functions can only be used when the command queue is opened in compile mode. When called within list mode 5, the loop is managed by the target controllers. The CUA32-FE controller processes the loop in all other compile modes. Nesting of control flow commands is limited to 16 levels.

parameters:
LoopCtr: 0...65535

rtSetMatrix(double a11, double a12, double a21, double a22);

This command adds a transformation matrix change event to the command list. When executed the target's transformation matrix is altered. An application can rotate and stretch all graphical output in the XY- plane using the transformation matrix.

parameters:
a11,a12,a21,a22 : factor

rtSetMinGatePeriod(long Time);

This function adds a configuration event to the command list. The minimal gate period is used to limit output frequency when pulse commands (*rtPulse, rtPuls3D*) are processed. After execution, the period between pulse outputs from said functions will not be smaller than the declared time.

parameters:
Time : 0…65535 µsec

rtSetOffsIndex(long Index);

The CUA32-TGT has 8 vectors allowing the application to preset several offsets values. This function adds a configuration event to the command list. When executed it activates the indexed offset. Values stored on this index are offsets the marking while updates (*rtSetOffsXY*) will be stored on this index. Indexing of offset provides a straight forward solution to shift relative positions between images on a single marking.

parameters:
Index: 0…7

rtSetOffsXY(double X, double Y);

This function adds an offset change event in the command list. All output coordinates are shifted by an offset vector. When executed, this command will set this shift to the declared values.

parameters:
X,Y: image offset position, range -8388.608...8388.607 mm

rtSetOffsZ(double Z);

This function adds a Z-offset change event in the command list. All output coordinates are Z-shifted by a Z-offset vector (*OffsZ*). When executed, this command will set this shift to the declared value.

parameters:
Z: image offset position, range -8388.608...8388.607 mm

rtSetOscillator(long Nr, double Period, double PulseWidth);

This function adds a configuration event to the command list. Combined with controlling connected deflectors, the target controller also controls the laser. Several steering modes are available including Gated, Q-switched, PWM-CO2 and speed modulated Q-switching. Configuration is done over three programmable oscillators which can be set up by this function.
parameters:
Nr =1: Function sets period and pulse width CO2 activated
Nr =2: Function sets pulse width CO2 idle (period must be the same as the Nr 1 setting)
Nr =3: Function sets period and pulse width Burst mode
Period : range 0...819.18 µsec
PulseWidth : range 0...Period

rtSetOTF(long Nr, bool On);

This command adds a configuration event to the command list. Its execution switches the on-the-fly offsetting on or off. When activated, the on-the-fly counter value is subtracted from the graphical coordinates. The resulting difference is sent to the deflector. The on-the-fly counter is not controlled by this function and remains activated.

parameters:
Nr = 1: function applies to X axis
Nr = 2: function applies to Y axis
Nr = 3: function applies to Z axis
On : 1 for on, 0 for off

rtSetPower(long pwr)
rtSetPowerLevels(long pwr100, long pwr0);

The function *rtSetPower* queues a laser transparent mean to set laser power between markings. When CO2 laser mode is selected, the function will set the pulse width. When analog power out (IO5) is selected, this command will set the analog value. The parameter *pwr* has to be set to 1000 when full power is desired and zeroed for gating without laser light.

The actual values, voltage or pulse width, needed to obtain the desired power levels is likely to change over time. When a laser ages, its power output decreases. The function *rtSetPowerLevels* is added to cope with this. With said function the application can set the pulse width or voltage for full and idle power output. Those values will be used to scale the parameter value *pwr* of the *rtSetPower* command. This power mapping is also used when executing functions *rtSetPowerProfile* and *rtPowerProfileTo*

Parameters:
Pwr : desired power level in 0.1%
Pwr100: steering value in 0.1% for full power (pwr = 1000)
Pwr0: steering value in 0.1% for idle power (pwr = 0)

Power control over IO5 when set as “Power Output”
IO5 has priority over PWM or laser-link power output control. When configured as power output the function will set the output voltage on IO5 without changing the pulse width or forwarding a power set command to the laser-link.
IO5 voltage=(*pwr**(*pwr100*-*pwr0*)/1000 + *pwr0*)*5mV

Power control over IO17 when set as “CO2(Gate*Osc1+!Gate*Osc2)”
CO2 laser are controlled using 2 oscillators. When the gate signal is active, IO17 is connected to oscillator 1, when inactive the pin is driven by oscillator 2. The function *rtSetPower* controls the laser energy when asserted so it only impacts the *PulseWidth* of oscillator 1.
PulseWidth=(*pwr**(*pwr100*-*pwr0*)/1000 + *pwr0*)*(*Period* /1000) µsec

rtSetPowerProfile(char* Pixels);

This function queues a power profile configuration to the command list. The parameter *pixels* points to a hexadecimal string. After conversion to binary its content is used to set the power profile used when wobbling (*rtSetWobbleEx*). When initializing the wobble controller, the CUA32-TGT calculates the greatest common divisor between normal and tangent wobble frequencies. Besides being used as a reference signal for said frequencies, the resulting period is used as a mapping input for the power look up table. Its 360° period is divided into a number of segments equal to the length of the binary pixel string. Every segment will be executed using the power level stored in his pixel byte.

Some laser types support dynamic power steering. When laser power modulation is selected, the pixel string controls the power setting of the laser during the wobble movement. Pixel data 255 sets the laser output (analog output, PWM or laser link) at the 100% level while 0 sets it 0% level. Like with deflectors, every 10 µsec a new power value is set. The CUA32-TGT controller uses a linear interpolation scheme to harmonize the available power data points with the wobble period of the reference wave. A scaling mechanism defining 0% and 100% settings is available to cope with laser-to-laser differences and aging issues (*rtSetPowerLevels*).
Actual power steering output:
IO5 voltage=(*pwr**(*pwr100*-*pwr0*)/256 + *pwr0*)*5mV
PulseWidth=(*pwr**(*pwr100*-*pwr0*)/256 + *pwr0*)*(*Period* /1000) µsec

When speed modulation is activated (*rtSetWobbleEx Type* 102 or 202) the frequencies and starting phases of both normal and tangential wobble components remain the same. During a period of the reference signal, its rotating speed is modulated to achieve the desired speed profile over the wobble. The wobble figure remains unaffected. Segments containing smaller pixel data will be executed more quickly to compensate the differences in line densities (ref. *rtSetWobbleEx*)

rtSetPulseBulge(double Factor);

rtSetQueryTarget(long Index);

This function sets the query target index. A single CUA32 device can comprise up to 8 target controllers, each having 17 IO's and up to three connected rhothor deflectors. When IO's are polled and target status is queried, the query index is used to select the target of interest. At default, the query index is set to 1. This function takes 50 milliseconds to execute.

parameters
Index : 1…8

rtSetResolver(long Nr, double StepSize, double Range);

This mode adds an on-the-fly-configuration event to the command list. Using the CUA32 controller, on the fly marking is easily realized. The target controller uses a A/B resolver input scheme connected to an on-the-fly-offset counter. This counter in altered with step size whenever a flank is detected. The direction is determined by phase shift. The minimal time between flank changes is 2 µsec. When this command is executed, the step size is defined, and the offset counter cleared. The range parameter is obsolete and should be set to zero.

parameters:
Nr = 1: function applies to X axis
Nr = 2: function applies to Y axis
Nr = 3: function applies to Z axis
StepSize : mm
Range : obsolete parameter, set to 0

rtSetResolverCal(const char* FileName);

This function call sets the resolver calibration offset data on the selected target.

parameters:
FileName : name of the text file holding the offset data

rtSetResolverPosition(long Nr, double Position);

This function adds an on-the-fly-data change event to the command list. When executed, the target's on-the-fly-offset counter is loaded with the declared value.

parameters:
Nr = 1: function applies to X axis
Nr = 2: function applies to Y axis
Position : -8388.607... 8388.607 mm

rtSetRotation(double Angle);

This function adds a transformation matrix change event to the command list. When executed the target's transformation matrix (*Aij*) will be loaded with a rotation function maintaining the previous loaded scaling (*rtSetScale(Scale)*) The function is the same as:
*rtSetMatrix(Scale*cos(Angle),- Scale*sin(Angle), Scale*sin(Angle), Scale*cos(Angle))*

parameters:
Angle : NAN or rotation angle in degrees

The CUA32 supports the use of NAN constants to select an analog input as the source for the rotation angle. A 5 V voltage on the selected analog port rotates the image 180° (counter clockwise) while grounding it rotates the image -180° (clockwise). A 2.5 V voltage corresponds with no rotation.
NAN constants:
0x7F800005: analog input 5 will be used to set the rotation
0x7F800006: analog input 6 will be used to set the rotation
0x7F800007: analog input 7 will be used to set the rotation
0x7F800008: analog input 8 will be used to set the rotation

rtSetScale(double Scale);

This function adds a transformation matrix change event to the command list. When executed the target's transformation matrix (*Aij*) will be scaled maintaining the previously loaded rotation (*rtSetRotation(Angle)*). The function is the same as :
*rtSetMatrix(Scale*cos(Angle),- Scale*sin(Angle), Scale*sin(Angle), Scale*cos(Angle))*

The CUA32 supports the use of NAN constants to select an analog input as the source for the scale. A 5 V voltage on the selected analog port sets the scale to the *Scale* value of the previous *rtSetScale* command while grounding it minimizes the scale to 0.
NAN constants:
0x7F800005: analog input 5 will be used to set the rotation
0x7F800006: analog input 6 will be used to set the rotation
0x7F800007: analog input 7 will be used to set the rotation
0x7F800008: analog input 8 will be used to set the rotation

rtSetSpeed(double Speed);

This function adds a speed change event to the command list. During execution of marking functions (rtLineTo, rtMoveTo, rtArcTo...) the deflectors setpoints are moved at marking speed. When executed, this function sets the target's marking speed to the declared value.

parameters:
Speed : NAN or speed in mm/s

The CUA32 supports the use of NAN constants to select an analog input as the source for the speed. A 5 V voltage on the selected analog port sets the speed to the *Speed* value of the previous *rtSetSpeed* command while grounding the analog pin minimizes the speed to 100 mm/sec.
NAN constants:
0x7F800005: analog input 5 will be used to set the speed
0x7F800006: analog input 6 will be used to set the speed
0x7F800007: analog input 7 will be used to set the speed
0x7F800008: analog input 8 will be used to set the speed

rtSetTable(long Nr, double Position);

This function adds a table-data-event to the command list. Besides controlling deflectors and laser, the target is also able to control stepper motors. The target controller comprises three stepper position regulators to provide the needed logic. When this command is executed, the controller doesn't move the connected stepper but loads it's setpoint and actual position with the declared value. This function could be used after calling rtSetTableWhileIO as a part of the axis reference cycle.

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
Position : range -8388.608...8388.607 mm

rtSetTableDelay(long Nr, long Delay);

The table logic operates under direct (sperate mode) or implicit control (hybrid mode). The stepper's actual position counter is regulated to align with the obtained setpoint. When correctly set, this regulation guarantees that the outgoing pulse signals can be handled by the actual stepper motor. To limit pulse frequency changes, the logic comprises a low pass filter. When executed, this function sets its time constant. An easy way to determine the value to be set is using a trial-and-error approach. Choose a delay time which feels right and invoke a table move command at the desired speed. Make sure that the movement is long enough so the full speed could be reached. When executed correctly, decrease the delay time and repeat the trial. When the first trial already faulted increase the delay. The maximal frequency that can be generated by the CUA32 controller is 100 KHz. However, limited pull up torque of the stepper motors will reduce this frequency. The application should make sure that no table movements are issued when the speed (*rtSetSpeed*) is set too high.

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
Delay : range 0...2147483.647 µsec

rtSetTableLimitSwitches(long Nr, long MinStop; long MaxStop);

This function adds a configuration event to the command list. When executed, this function sets the limit switches for a table axis. Using limit switches is a commonly used strategy for limiting the mechanical travel range. As soon as the moving axis hit's the limit switch, the movement will be stopped. The limit switches will be polled during execution of the commands *rtSystemTableMove* and *rtSystemTableMoveRel.* When during execution of those functions the table-coordinates increase only the *MaxStop* input will be polled while the *MinStop* input will be polled during the opposite movement. The limit switch input should be high to enable the table movements.

Limit switches may be located on a target different from the target selected by *rtSetTarget*. The parameters *MinStop* and *MaxStop* use the following format:

- Parameter value divided by 100 equals the target number that holds the limit switch
- Parameter value modulo 100 equals the IO number on said target

Values smaller than 100 locate the limit switches on the target device currently being addressed (*rtSetTarget*)

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
MinStop : input number of low side limit switch, zero for none
MaxStop : input number of high side limit switch, zero for none

example:

```
…
rtSetTarget(2);                          // select target nr. 2
rtSetTableLimitSwitches(1,301,302);   // X axis uses IO1 and IO2 of target nr. 3 as limit switches
…
```

rtSetTableMaxSpeed(long Nr, double Speed);

This function adds a configuration event to the command list. When executed, this function sets the maximal speed for the table axis. The value is used as a clipping level whenever the table speed is being set. Combined with the axis step size, it also determines the high time of the PULS signal. When running at maximal speed, the duty cycle of the *PULS* signal will be 50%. The maximum output frequency of the *PULS* output (*IO9*, *IO11*, *IO13* on a *CUA32-TGT* controller) is 500 KHz. When the axis step size is set to 2 µm, the *Speed* value is limited to 1000 mm/s. (2 µm * 500.000 steps/sec = 1000 mm/s).

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
Speed : speed in mm/s

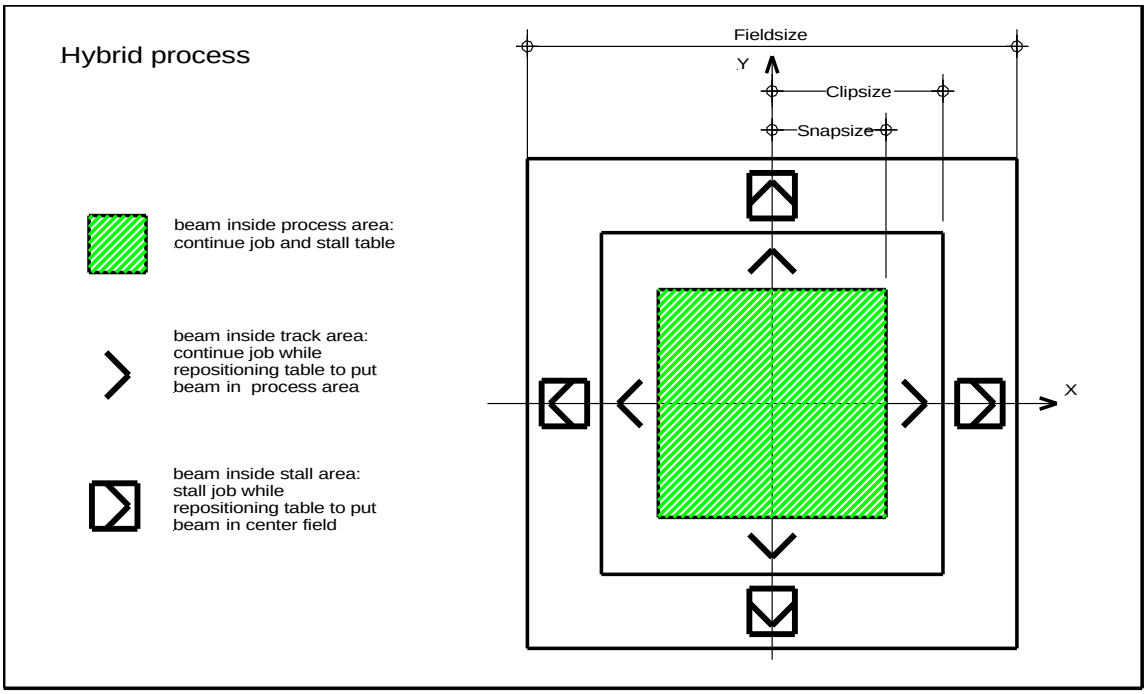
rtSetTableRange(long Nr, long Type, double Value);

This function adds a configuration event to the command list. With this function the travel range of an axis can be limited. When parameter *Type* equals one, the function parameter *Value* defines the minimal table position. Maximal table position can be set by calling this function with parameter *Type* set to two. The target controller uses a clipping mechanism when table is steered beyond its ranges.

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
Type =1: minimal table setpoint
Type =2: maximal table setpoint
Value =1: clipping value in mm

rtSetTableSnapSize(long Nr, double SnapSize);
rtSetTableSnapSizeEx(long Nr, double SnapSize, double ClipSize);

This function queues a configuration event. When executed, the command activates the hybrid axis mode. Normally marking functions (*rtArcTo, rtLineTo...*) are executed by the deflectors limiting the coordinate range of their parameters. When deflectors are steered outside their field size, a clipping protecting mechanism jumps in. Hybrid marking is an easy-to-use tool to overcome this limitation. When activated the table axis's as well as the deflectors track the graphical coordinate stream. High frequency components of the coordinate stream are covered by deflectors while the longer low frequency movements are executed using the table axes. Table movement is started as soon as the deflector is steered above *SnapSize* or below -*SnapSize*. The hybrid mode is switched off when the function is called with zero as *SnapSize* or by calling direct table functions (*rtTableMove, rtTableLineTo...*). The parameter *ClipSize* defines the operational range for the deflectors. The speed used to position the tables is set by *rtSetTableSpeed* while the process speed is set by *rtSetSpeed*. It can be that the latter is higher than the table speed so during long runs the table axis's will not be able to keep up. When the deflectors reach an outer boundary, set by *ClipSize*, the target controller will stall. As soon as the deflectors coordinates are back within the area defined by *SnapSize* the command processing resumes. When the hybrid mode is activated with the function *rtSetTableSnapSize* the clip size is set to twice the snap size.



parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
SnapSize : range 0...8388.607 mm
ClipSize : range *SnapSize*...8388.607 mm

rtSetTableSpeed(double Speed);

This function queues a configuration event. When executed, the command sets the speed used for all table axis moves. The CUA32-TGT target controller will not move any axis faster as set by *rtSetTableMaxSpeed*. An application should avoid invoking this function setting the speed above that of the slowest axis.

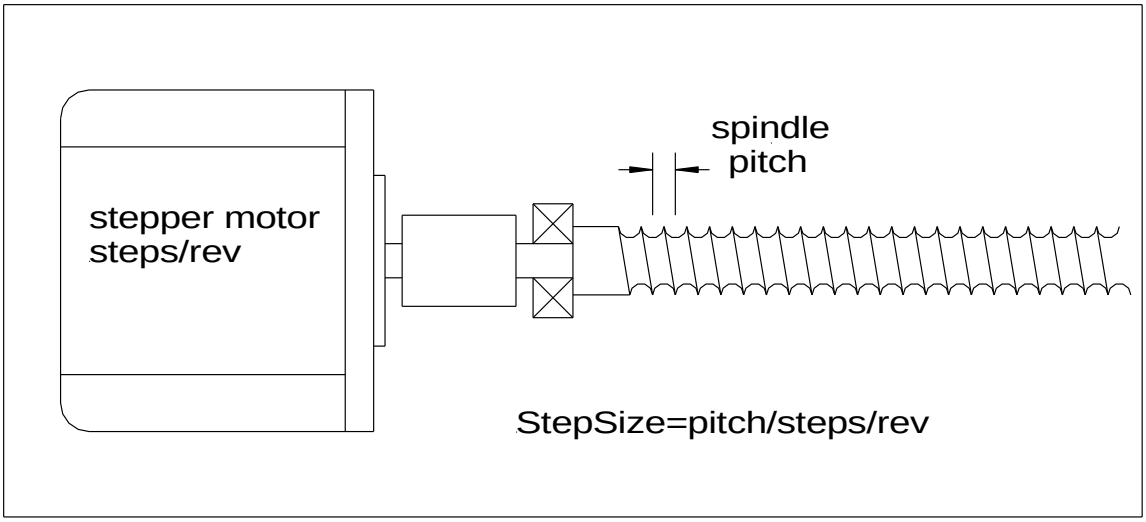
parameters:
Speed : speed in mm/s

rtSetTableStepSize(long Nr, double StepSize);

This function adds a table configuration event to the command list. When executed the axis position increment for every outgoing step is defined. Table axes are controlled by CUA32-TGT target systems using a direction and a step signal. The direction signal specifies the movement's orientation while the frequency and number of outgoing pulses determine speed and distance of the travel. Both signals are controlled by internal logic. This function activates this logic and should be called prior to any other table commands. When called with step size set to zero, the logic is idled. Axis orientation can be altered by using a negative value for step size.

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
StepSize : mm/step

example:
table axis comprising stepper and spindle



rtSetTarget(long Mask);

A complete CUA32 system can comprise up to 8 target controllers. Commands can be sent to one or several target controllers at the same time to provide synchronized or independent control of all connected deflection systems. Activation of a target controller is done by simply setting its mask bit. When single target functions are called, the least significant activated bit is used to define the function's target.

parameters:
Mask : 0...255

Example:
rtSetTarget(0b01000001); single target functions will apply to target 1 while masked target functions will apply to targets 1 and 7

rtSetVarBlock(long i, char data);

Not all the content is known at compile time for some marking jobs. Serial numbers and production code are last minute data that have to fed to the system prior to marking. When the CUA32-MST has a life host connection, changing the marking content is easily achievable. Variable data memory blocks are used to provide similar flexibility whenever the system runs without host. The CUA32-MST controller has two 2 Kbyte variable blocks. A Ping-Pong accessing scheme is used to handle semaphore issues. While one data block provides data for the current marking the other one is free for data entry.

Parameters:
I : 0...2047, address to store the character code
Data : character code to be stored

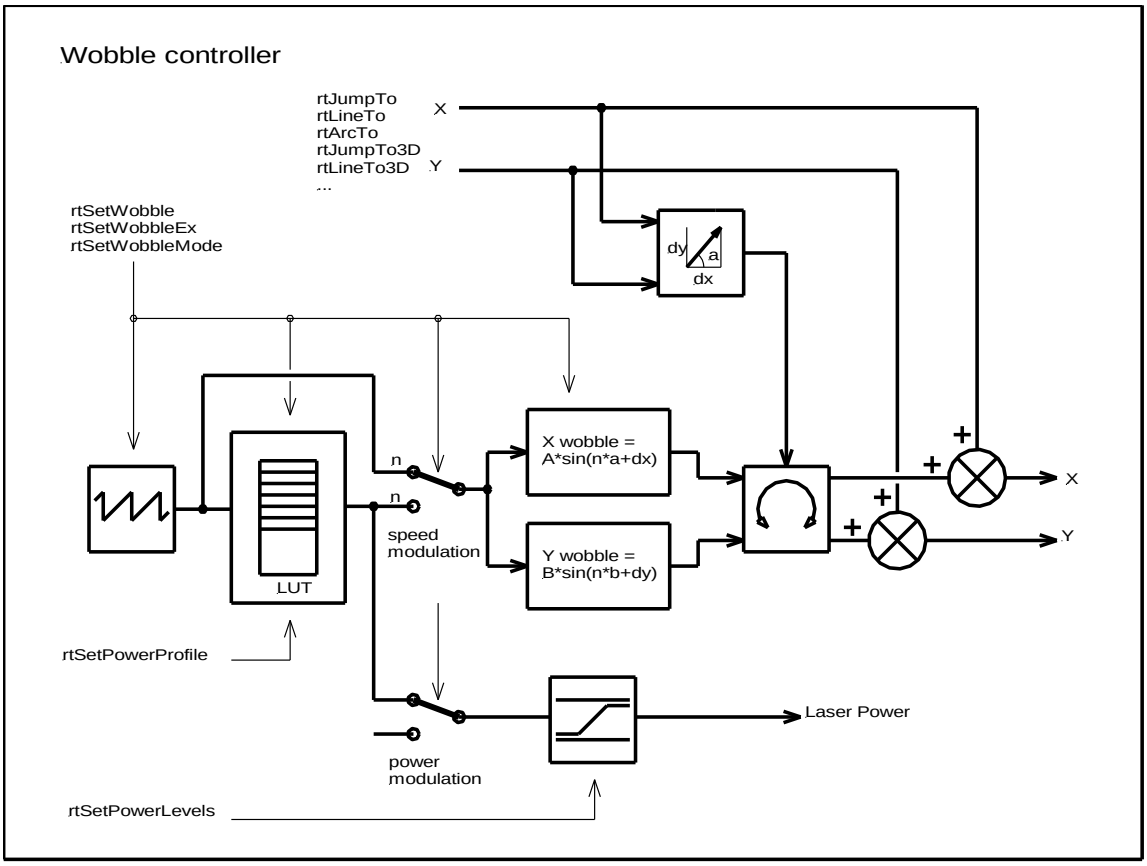
rtSetWhileIO(long Value, long Mask);

This function adds an interlock configuration to the command list. Interlocking is an easy way to abort lasering when some IO events happen. IO's with their mask bit set will be compared against *Value* . Any deviation will stop the laser and aborts normal command processing on all installed target controllers. This feature should not be used as a safety system! During an interlock error all queued commands are ignored. The interrupt function *rtSystemSetWhileIO* provides means to disable the interlock.

Parameters:
The parameter *Value* is the binary presentation of the input levels. I1 mapped to bit 0, I2 to bit 1 and so on.
The parameter Mask is similar but has one additional bit. When bit 23 in Mask is set, interlock error will be triggered when the system detects deflector tracking or communication error.

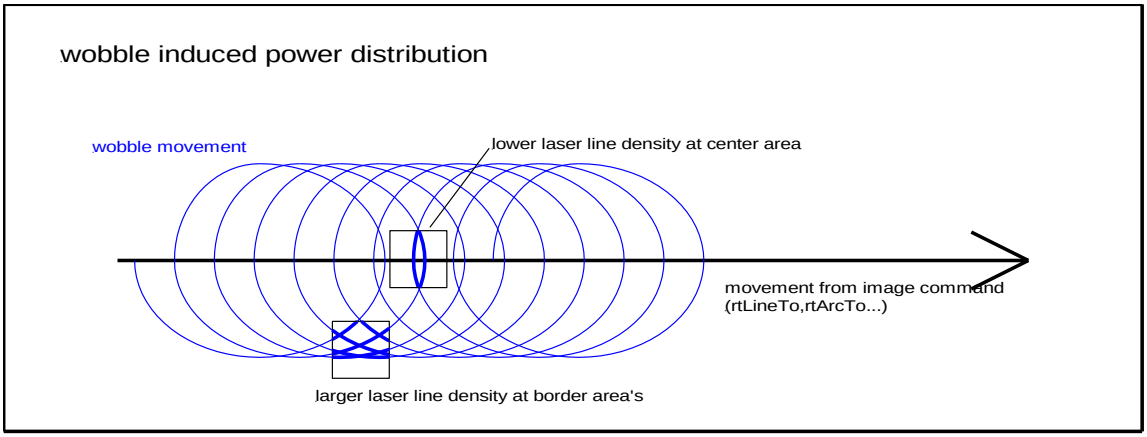
rtSetWobble(double Diam, long Freq);
rtSetWobbleEx(long nType, double nAmpl, long nFreq, long tType, double tAmpl, long tHarm, long tPhase);
rtSetWobbleEx(long Type, double B, long b, long dy, double A, long a, long dx);

This function adds a wobble control function to the command list. When image commands are processed (*rtLineTo,rtArcTo,…*) the deflection moves the bare laser beam across the workpiece. In some cases, the resulting marking will be too thin. Hatching techniques can be used to create more solid lines. However, generating the needed hatch movements to obtain the desired effect is not always easy. Wobbling is a known alternative and easier to implement. Wobble adds oscillation style movements when image commands are executed. With the function *rtSetWobbleEx* the application can configure two sine wave generators. The output of the first lets the beam oscillate orthogonal to the marking direction while the second lets the beam oscillate in the marking direction. Both oscillators can be parameterized separately. The CUA32 controller matches the selected frequencies to guaranteed predictable phase shifts.



The CUA32 uses following wobble function definitions:
X=A*sin(a*2*PI*t+dx*PI/180)
Y=B*sin(b*2*PI*t+dy*PI/180)
The resulting Lissajous figure is automatically aligned with the momentary direction of the image command being processed. So, in these definitions, X is not the X axis of the deflection system, but the momentary (tangent) direction of the laser beam movement (excluding wobble). Y direction lies perpendicular (normal) to said movement. When initializing the wobble controller, the CUA32-TGT calculates the greatest common divisor between normal and tangent frequencies. The result will be used to set the base frequency for the wobble generators. Wobbling will be disabled when said frequency is lower than 10 Hz.

During the wobble, an additional power signal can be generated to modulate the laser. When the laser doesn't support power output modulation, a similar effect can be achieved by using speed modulation. In the latter, the output of the wobble sinewave generators aren't just sine waves. They will run more quickly in wobble areas where to power needs to be reduced.. The need of power modulation during wobble is quite obvious. When spiraling during a movement it is easy to see that the outside areas are getting much more energy than the center. Power modulation enables equalizing power density over the surface. When power modulation is needed, the power levels need to be set prior by using the function *rtSetPowerProfile*.



Wobbling is automatically switched off when the laser goes inactive to minimize power consumption. After configuration, the application can suspend or change wobble direction and phase between vector lines by using the function *rtSetWobbleMode*

rtSetWobbleEx parameter usage depends on the selected wobble type:
Type = 0 or 1: old style parameters, kept for backwards compatibility
Type = 100: add Lissajous style wobble
Type = 101: add Lissajous style wobble + laser power modulation
Type = 102: add Lissajous style wobble + speed modulation
Type = 200: run Lissajous style wobble until aborted (*rtAbort*)
Type = 201: run Lissajous style wobble + laser power modulation until aborted (*rtAbort*)
Type = 200: run Lissajous style wobble + speed modulation until aborted (*rtAbort*)

Type 10x and 20x style parameters:

B : normal amplitude, mm
A : tangent amplitude, mm
b : normal frequency 10...4000 Hz
a : tangent frequency 10...4000 Hz
dx;dy : degrees

Type 101 and 102 laser power modulation:
When power modulation is activated the power profile string (*rtSetPowerProfile*) is used to change the laser power during the wobble. Linear interpolation is used to smooth out the power changes.

Type 102 and 202 speed modulation:
When speed modulation is activated the frequencies of the normal and tangent component stay the same. Only the speed distribution within their periods is altered. The speed distribution is calculated using the power profile string (*rtSetPowerProfile*).
Type 20x wobble style laser milling:
When wobble type 20x is selected, its movement isn't added to the successive image vectors but executed immediately on the current position. The command fetcher is halted until the wobbling is aborted. (*rtAbort*). This mode can be quite useful when the deflection system is used like a mill. The controller spins the laser while the application uses another CUA32-TGT or third-party controller to move the workpiece.

Type 0 and 1 style parameters: (kept for backwards compatibility).

nType =0: normal component disabled
nType =1: normal sine wave
nAmpl : normal amplitude, mm
nFreq : normal frequency 10...4000 Hz
tType =0: tangent component disabled
tType =1: tangent sine wave
tAmpl : tangent amplitude, mm
tHarm : tangent frequency = tHarm*normal frequency
tPhase : -180...180 degrees

The function *rtSetWobble* activates a regular circular wobble.

Parameters:
Diam : mm
Freq : 10...4000 Hz

rtSetWobbleMode(long Dir, long Phase);

This function adds a wobble control function to the command list. After configuration (*rtSetWobble*, *rtSetWobbleEx*) the wobble is activated for counter clockwise operation starting at zero angle. The wobble on the first line will start counter clockwise spiraling with starting angle equal to line direction or arc tangent. *rtSetWobbleMode* provides free choice of start angle and spiraling direction. This function should be called after *rtSetWobble* or *rtSetWobbleEx*.

parameters:
Dir =1: counter clockwise rotation
Dir =0: suspend wobble
Dir =-1: clockwise rotation
Phase : set wobble phase angle, relative to tangent -180...180 degrees

rtSleep(long Time);

This command adds a sleep event to the command list. When the command is executed, the target controller suspends and starts a timer. The command processing is resumed when the timer elapses. The command rtSleep is often used to implement a wait-after-jump feature.

parameters:
Time : 0...2147483647 µsec

rtStoreCalibrationFile(const char* FileName);

This function copies the target's calibration in a readable file on the host.

parameters
FileName : file name under which the calibration will be saved.

rtSuspend();

This function adds a suspend command to the command list. When processed, the targets will go in suspended state. *rtSuspend* is a list command, all pending list commands will be executed. A suspended system can be restarted by calling *rtSystemResume*;

rtSynchronise();

This function adds a synchronization command to the command list. When processed, the front-end controller will poll all targets for idle before resuming uploading commands. When targets mark different images, it is very likely that they will not finish at the same time. This function can be used when those targets share a table system and waiting for the last is needed.

rtSystemResume();

This function restarts command processing in case the system was suspended. It can be used in combination with the list command *rtSuspend()* to control list processing in streaming mode. Execution of command queues opened in compile mode starts after closing them (*rtListClose()*). Queues opened in streaming mode normally start when the dll sends the first commands to the hardware. Execution of streaming data can be suspended (*rtSuspend()*) when this behavior is not desired. The application can afterwards start list execution by calling *rtSystemResume*.

rtSystemSetIO(long Value, long Mask);

This command sets the digital output values
bit 0 of “Value” is copied to the output bit of IO1 only when bit 0 of “Mask” is true,
bit 1 of “Value” is copied to the output bit of IO2 only when bit 1 of “Mask” is true….

parameters:
Value, Mask : range 0…65535

rtSystemSuspend();

Immediately suspend all command processing. The target controllers will stop fetching instructions from their FIFO's. The instructions that are being processed during this call will complete normally.

rtSystemTableMove(long Nr, double Position);
rtSystemTableMoveRel(long Nr, double Offset);
rtSystemTableStop();

The functions *rtSystemTableMove* and *rtSystemTableMoveRel* start a move of the selected table. The speed at which the table moves can be set using *rtSetTableSpeed* command. The movement is stopped by calling *rtSystemTableStop*. Combined with the functions r*tSetTableLimitSwitches* and *rtSetTableRange* these functions are typically used to implement jogging and referencing functionality. The functions are picked up by all targets but only the selected target (*rtSetTarget*) makes the actual move. The other target's need command data to activate their limit switches.

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
Position : target position, range -8388.608...8388.607 mm
Offset: relative table position in mm

rtSystemUartOpen(long baudrate, char parity, char stopbits);

This function opens the serial interface.

parameters:
baudrate : 1200...115600 bit/sec
parity : 'n','e','o','s' for none, even, odd, or space
stopbits : 1 or 2

rtSystemUartWrite(long bytes, char* data);

The parameter *data* points to a string which is send by the UART as is. The function appends the string to the UART transmission queue and returns immediately, before the transmission itself terminates. The latter depends on baudrate and number of bytes to transmit. The UART controller has a 512-byte fifo to store messages for sending. Prior to invocation the application must make sure that the fifo has enough free memory to avoid overflow errors.

parameters:
bytes : number of bytes to be transmitted, range 1 to 511
data : pointer to the data that must be send

rtSystemUDPsend(char* IP, short port, char* data);

This method triggers an internet communication. When in use as a client, the ethernet connection of the CUA32-FE controller supports ARP server addressing. The application should query (*rtGetIP*) to obtain the assigned IP address. Parameter *data* points to a hexadecimal coded string containing the transmission content and length. The string is converted from hex codes to binary prior to being send. The first byte holds the length, limiting the UDP data size to 255 bytes.

parameters
IP : target IP address
port : target port
data : points to hexadecimal string of maximal 512 characters ('0'...'9','A'...'F').

example:
rtSystemUDPsend("172.16.224.20",10000,"020AA0");
Sends (2,10,160) to port 10000 at IP address 172.16.224.20.

rtTableArcTo(double X, double Y, double BF);

This command queues an arc style marking. The arc starts at the current position and goes to coordinate (X, Y) using the BF as bulge factor. The bulge is the tangent of 1/4 the included angle for an arc segment, made negative if the arc goes clockwise from the start point to the end. Marking speed (rtSetSpeed) is used during execution. During execution the XY table is used and not the deflection system. When both systems are used independently to mark lines, the application should make sure that the deflectors are in a known position whenever markings are made using the stepper controls.

parameters:
X,Y: target position, range -8388.608...8388.607 mm

rtTableJog(long Nr, double Speed, long WhileIO);

This function queues a table move event. When executed, the table setpoint will start to move using the requested speed. The function aborts when the designated input (*WhileIO*) becomes low or after invoking *rtAbort* . The command starts the move and returns immediately when invoked with parameter *WhileIO* set to zero. This allows parsing and executing deflector movements concurrent with a moving table axis.

The *rtTableJog* function can also be used to implement referencing functionality. The parameter *WhileIO* equals the input number on which the reference switch is connected. Said switch may be located on a target different from the target selected by *rtSetTarget*. The parameter *WhileIO* uses the following format:

- Parameter value divided by 100 equals the target number that holds the limit switch
- Parameter value modulo 100 equals the IO number on said target

Values smaller than 100 locate the limit switches on the target device currently being addressed (*rtSetTarget*)

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
Speed : mm/s
WhileIO (modulo 100): 0 for none, 1 for IO1, 2 for IO2, ... , 16 for IO16

rtTableLineTo(double X, double Y);
rtTableLineTo3D(double X, double Y, double Z);

These functions queue a line marking. The line starts at current table position and extends towards the target position. Marking speed (*rtSetSpeed*) is used during execution. During execution the XY table is used and not the deflection system. When both systems are needed independently to mark lines, the application should make sure that the deflectors are in a known position whenever markings are made using the stepper controls.

parameters:
X,Y,Z : target position, range -8388.608...8388.607 mm

rtTableMove(long Nr, double Target);
rtTableMoveTo(double X, double Y);
rtTableMoveTo3D(double X, double Y, double Z);

These functions queue a table move to the command list. The move starts at current table position and extends towards the target position. Marking speed (*rtSetSpeed*) is used and laser is idled during execution. *rtTableMove* starts a single, *rtTableMoveTo* starts a dual and *rtTableMoveTo3D* starts a triple axis move.

parameters:
Nr =1: stepper X axis
Nr =2: stepper Y axis
Nr =3: stepper Z-axis
Target,X,Y,Z: target position, range -8388.608...8388.607 mm

rtUartRead(long* bytes, char* data);

This function reads received data from serial interface. The content is stored in location pointed by data. The number of received bytes is stored in parameter bytes. At most, the function returns 511 bytes, so the parameter data should point to a 512-byte array.

parameters:
bytes : placeholder for the number of received bytes
data : 512-byte placeholder for the received data.

rtVarBlockFetch(long Start, long Size, const char* FontName);

This function queues a variable string mark. When executed, the system will mark the string stored in the variable buffer. Prior to invocation a font must be uploaded to the CUA32-TGT devices (*rtFontDef, rtCharDef* and *rtFontDefEnd*).

parameters:
Start : 0...2047, index in variable buffer of the first character
Size : number of characters
FontName : ignored

rtWaitCanLink(long ByteNr, long Value, long Mask);

This function queues a CANopen type wait event. When it executes the front-end controller will stall command processing until scanned PDO commands contained data as set by *Value*. Only bits with corresponding one bit in *Mask* will be evaluated. The node and PDO telegram of interest need to selected (*rtScanCanLink*) prior to invoking.

parameters:
ByteNr : 0…7, data in first log buffer
ByteNr: 8…15, data in second buffer
Value : 0...65535
Mask : 0...65535

rtWaitIdle();

This function queues a conditional stall. When executed, the command fetcher from the target controller will be suspended until all preceding commands including the last are processed. This function uses single target addressing. The function is forwarded as specified by the last *call rtSetTarget(mask)*. However only the least significant non-zero bit in mask will select the target. All other one bits are ignored.

rtWaitIO(long Value, long Mask);

This function queues a conditional stall. When called within list mode 5 the function is handled locally by the target controllers. In all other list modes, the front-end controller polls the inputs of the least significant target to compare them against the declared value. Instruction forwarding is suspended until the directive is met. The comparison only considers those IO's with their corresponding mask bit set. Before polling, the controller waits until all addressed targets are idle to guarantee synchronization. In list mode 5 the command queue is copied and executed independently on every target. Synchronization is not supported reducing command execution time.

parameters:
Value : 0...65535
Mask : 0...65535

rtWaitResolver(long Nr, double TriggerPos, long TriggerMode);

This function adds a local control flow command to the command list. During processing, the target controller suspends command processing until the selected on-the-fly counter has reached its desired value.

parameters:
Nr =1: on-the-fly counter X axis
Nr =2: on-the-fly counter Y axis
TriggerPos : trigger position, range -8388.608...8388.607 mm
TriggerMode = 1: wait until on-the-fly counter is higher than TriggerPos
TriggerMode = 2: wait until on-the-fly counter is lower than TriggerPos

rtWaitStall();

This function queues a conditional stall. When executed, the command fetcher from the target controller will be suspended until all preceding commands excluding the last are processed. This function uses single target addressing. The function is forwarded as specified by the last *call rtSetTarget(mask)*. However only the least significant non-zero bit in mask will select the target. All other one bits are ignored.

rtWhileIO(long Value, long Mask);
rtDoWhile();

These functions add control flow to the command list. It needs calculated instruction locations and therefore can only be used in compiled mode. Processing a *rtWhileIO* command starts by waiting until all connected targets are idle. This waiting guarantees that all target level queues are empty to synchronize looping. Afterwards the CUA32-FE controller polls the IO state of the least significant target. Its inputs are compared against parameter *Value* only considering those with their corresponding *Mask* bit set to one. When the result returns equal, the instruction sequencing embedded between the invocation and its counterpart *rtDoWhile* is repeated. When the comparison returns false, the instruction sequencer fetches the first command after *rtDoWhile*. In list mode 5 a copy of the same loop runs on every target controller using local queue and IO data. List mode 5 excludes the need to wait for idle resulting in a faster running loop. Loops can be 16 level deep nested.

parameters:
Value : 0...65535
Mask : 0...65535

Example:

```
rtSetTarget(3);
rtListOpen(5);
rtJumpTo(-10,10);
rtWhileIO(1,1);
rtLineTo(10,10);
rtLineTo(10,-10);
rtLineTo(-10,-10);
rtLineTo(-10,10);
rtDoWhile();
rtListClose();
```

Both target 1 and 2 will start drawing a 20 mm square while their input 1 is high. Because both loops are running independently, the laser is continuously activated while the loop condition is met. When the same list was opened using mode 1 (*rtListOpen(1)*), the flow would stall after every cycle allowing the front-end controller to check input 1 of target 1.

Not supported functions, kept for compatibility.

bcSamplePoint(double X, double Y, long Row, long Col, double Sweep, double OffsetX, double* OffsetY);*
bcSelectDevice(const char CommPort);*
rtAddCalibrationDataZ(const char FileName);*
rtLoadCalibration();
rtLoadCalibrationFileZ(const char FileName);*
rtResetCalibrationZ();
rtResetEventCounter();
rtSendUartLink(const char Data);*
rtSetHover(long Time);
rtSetImageMatrix3D(double a11, double a12, double a21, double a22, double a31, double a32);
rtSetLead(long Time);
rtSetMaxSpeed(double Speed);
rtSetResolverRange(long Nr, double Range);
rtSetResolverTrigger(long Nr, double Position, long IO);
rtSetTableOffsXY(double X, double Y);
rtSetTableSnap(double Distance);
rtSetTableWhileIO(long Value, long Mask);
rtStoreCalibration();
rtStoreCalibrationFileZ(const char FileName);*
rtWaitEventCounter(long Count);
rtWaitPosition(double Window);

library functions : return codes

ERR_OK (-1)
The function call completed successfully.

ERR_BUSY (2)
The application tried to invoke a function which requires an idle system on a non-idle system. To solve, make sure that the command list is closed (*rtListClose, rtFileClose or rtAbort*) and try again.

ERR_JOB (3)
The application invoked a command list function without having the queue properly opened. To solve, make sure that the command list is ready to receive commands (*rtListOpen, rtFileOpen*) and try again. Open the command list in compile mode (*rtListOpen(4)*) when control flow commands are needed.

ERR_HARDWARE (5)
Failed to set up or maintain physical connection with the CUA32 hardware. Verify IP-address, USB ID string and electrical connection with host and try again.

ERR_DATA (13)
The application invoked a function with invalid parameters. Verify if the function parameters are within the specification and ranges as described in this manual.

ERR_IMPLEMENTATION (23)
The function is not supported by the library. Function entry is kept for compatibility reasons with previous library versions.

ERR_INTERLOCK (48)
The front-end systems interlock signal became invalid. The problem source needs to be resolved after which the interlock can be restored by calling *rtAbort* or *rtReset*.

11. document history

CUA32-App01: first draft

CUA32-App01.1

* rtListOpen (corrected)
* rtSetCfgIO (altered)
* rtSetRotation (corrected)
* rtSetTableMaxSpeed (new)
* rtSetTableSpeed (new)
* rtSetWhileIO (new)
* rtSetWobbleMode (new)
* rtScanCanLink (new)
* rtTableLineTo3D (new)
* rtTableMoveTo3D (new)

CUA32-App01.2

* rtAcceptData (new)
* rtCharDef (new)
* rtFontDef (new)
* rtFontDefEnd (new)
* rtGetDeflReplies (altered)
* rtGetCanLink (new)
* rtLineToXD (new)
* rtMoveToXD (new)
* rtOpenCanLink (new)
* rtParse (altered)
* rtPowerProfileTo (new)
* rtPrint (new)
* rtReset (altered)
* rtSetCanLink (new)
* rtSetLaserFirstPulse (new)
* rtSetTableLimitSwitches (new)
* rtSetTableRange (new)
* rtSetTableSnapSize (altered)
* rtSetTableSnapSizeEx (new)
* rtSetTableWhileIO (altered)
* rtSetVarBlock (altered)
* rtSetWhileIO (altered)
* rtSynchronise()
* rtSystemTableMoveTo (new)
* rtSystemTableMoveToRel (new)
* rtSystemTableStop (new)
* rtVarBlockFetch (altered)
* rtWaitCanLink (new)

CUA32-App01.3 (Q3-2021)

* changed document layout
* deflector calibration (extended)
* resolver calibration (new)
* rtGetResolvers (altered)
* rtSetResolverCal (new)
* rtSetOTF (altered)
* rtSetResolver (altered)
* rtSetResolverCal (new)
* rtSetTableSnapSize, rtSetTableSnapSizeEx (altered)
* rtSetWobbleEx (altered)

CUA32-App01.4 (Q3-2022)

* deflector control (altered)
* laser control (altered)
* on-the-fly resolver calibration (altered)
* table axis control (altered)
* rtPowerProfileTo (altered)
* rtSetPower (new)
* rtSetPowerLevels (new)
* rtSetPowerProfile (new)
* rtSetRotation (altered)
* rtSetScale (altered)
* rtSetSpeed (altered)
* rtSetWobbleEx (altered)
* rtWaitIdle (new)
* rtWaitStall (new)