CTI Universal API Reference

PN P0900-0160

Cambridge Technology

125 Middlesex Turnpike, Bedford, MA 01730
Tel: (781) 266-5700 Fax: (781) 266-5114
www.cambridgetechnology.com
January 28, 2013

Contents

1 CTI Universal API

1.1 Index

1.2 Introduction

The Universal API is a cross-platform API for programming Cambridge Technology Inc.'s (CTI) Scan Controller product family allowing for jobs to be easily ported between the CTI Scan Controller product family. These controllers include the SM1000, EC1000, Lightning II and SC500.

This manual is provided as a reference for the Universal API.

1.3 See also:

Contact and Assistance

System Requirements

Software Installation

2 Contact and Assistance

```
    To get assistance with this software development kit, contact us via Email at ScanControl@cambridgetechnology
```

3 System Requirements

3.1 Minimum Requirements

To use this software, you need a 32-bit PC with Windows XP, a compatible scan controller: SM1000, EC1000, SC500 or Lightning II and, a suitable development environment.

4 Software Installation

Software installation should be carried out prior to connecting the scan controller card to the host PC. This means that the device driver for the SC500 can be located by the Windows operating system.

To install the software, follow these steps

1. Insert the ScanMaster Software Suite CD (CTI PN: PA950-0136) into the host PC drive.

Click the installer for the hardware you have. Either SC500 SDK or EC1000 SDK.

1. Click 'Install CTI UAPI' on the Auto-run Menu

   The installer copies the software development kit onto the host PC in the default location of C:\Program Files\C-TI or in an alternate location selected by the user. The installer adjusts the path environment variable and also checks that all supporting libraries, such as Microsoft C runtime, are at the appropriate revision levels. If necessary, the installer will update the supporting libraries. The installer also installs the necessary device drivers onto the host PC.
2. After the software installation has been completed, apply DC power to the scan controller and connect the scan controller's communication cable to the host PC.
3. For SC500 only - check the Windows Device Manager in the Servo Controllers category to make sure the CTI Lightning II driver is present.
4. The scan controller is now ready to use.

5 Theory of Operation

Basics of Marking

- Transforms
- Field Calibration
- Vector Marking
- Fine Tuning with Delays
- Introduction the Delay Parameters

5.1 Transforms

```
@section MarkingField Marking Field Transforms
A 2-dimensional transform is described in the UAPI by a 3 x 3 matrix that is applied to a point to remap it into
```

$$\begin{vmatrix} X_1 \\ Y_1 \\ 1 \end{vmatrix} = \begin{vmatrix} X_X & Y_X & D_X \\ X_Y & Y_Y & D_Y \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} X_0 \\ Y_0 \\ 1 \end{vmatrix}$$

2-dimensional Marking Field Transform

The above operation will transform Point$_0$ $\{X_0, Y_0\}$ to Point$_1$ $\{X_1, Y_1\}$ as follows:

$$X_1 = X_X * X_0 + Y_X * Y_0 + D_X$$
$$Y_1 = X_Y * X_0 + Y_Y * Y_0 + D_Y$$

D_X and D_Y in the above transform are displacement coefficients, while the other four values address rotation, scaling, etc. The last row is constant, so when defining transforms one only needs to send or retrieve the six Parameters $\{X_X, X_Y, Y_Y, D_X, D_Y\}$

It is often more convenient to define the transformation in terms of Rotation, Scaling, and Translation, so the UAPI provides tools to define the Transform matrices in these terms

5.1.1 Rotate Transform

This transform rotates an object about the origin by an angle θ relative to the positive X direction, with a positive θ resulting in a counter-clockwise rotation

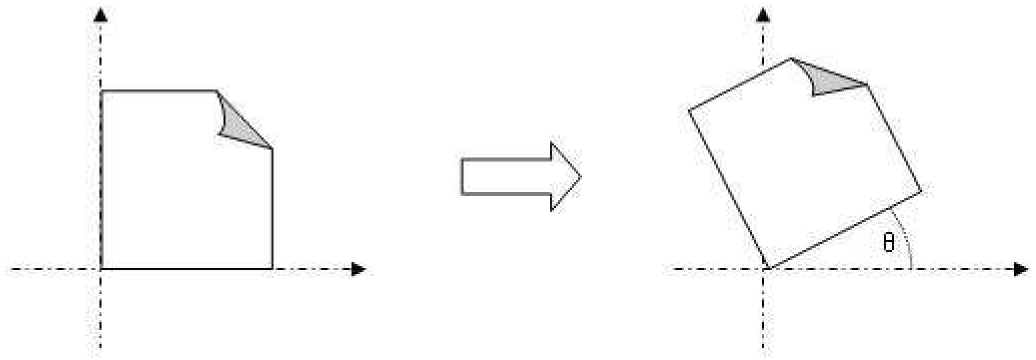

Figure 1: Figure 1: Rotate Transform

For more information see rotate transform section

5.1.2 Scale Transform

This transform scales an object'ss X and Y dimensions

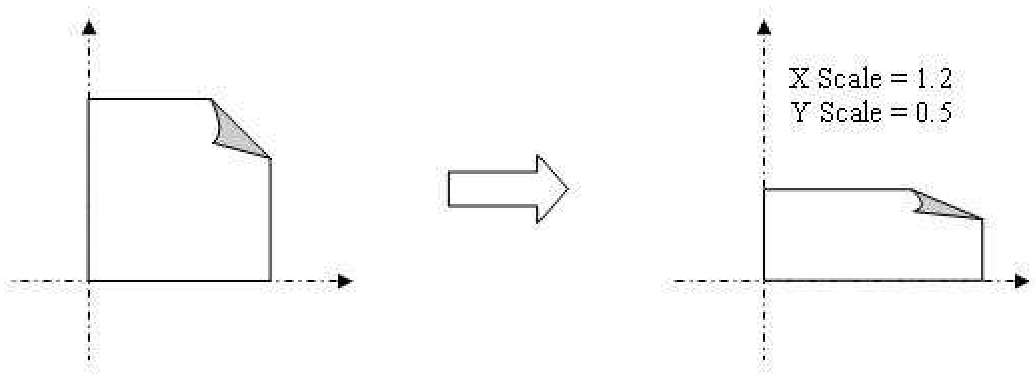

Figure 2: Figure 2: Scale Transform

For more information see scale transform section

5.1.3 Translate Transform

This transform translates (moves) an object along the X and Y axes.

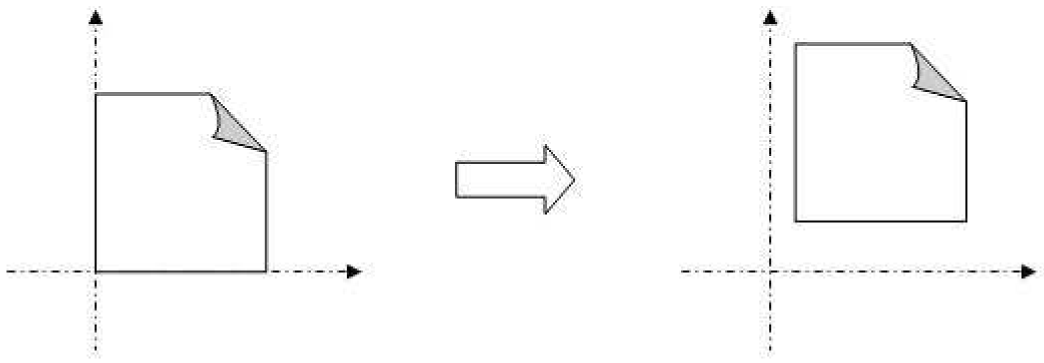

Figure 3: Figure 3: Translate Transform

For more information see translate transform section

5.1.4 Transform Multiplication

Transforms can be combined by multiplication, with the affected product-transform being applied to the data in the reverse of the multiplication order

```
For:     A = Rotate Transform
```

```
B = Scale Transform

C = A * B will result in the object scaled first and then rotated
D = C - 1 will result in the object rotated in the opposite direction and then scaled back
```

An inverse transform can be calculated by inverting the matrix

5.2 Field Calibration

The UAPI is using a Cartesian coordinate system. In reality, two different coordinate systems are involved in the UAPI's operation:

- Un-calibrated Coordinates - the Cartesian coordinate system in the computer environment in terms of one full field
- Workspace Coordinates - the Cartesian coordinate system of the physical work piece

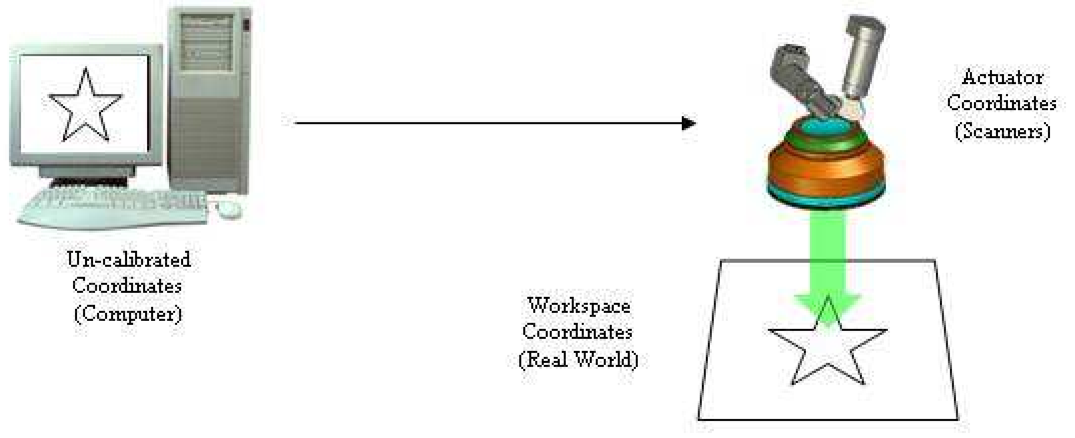

Figure 4: Figure 4: Coordinate Systems, Uncalibrated

The coordinates and other parameters in the computer program are numerical values representing positions from -0.5 to +0.5 of the full range of the servo. The un-calibrated coordinate system may be very different from the workspace coordinate systems that have specific dimensions / units, and might be influenced by residual errors from optics or angular distortions. The purpose of the system calibration process is to unify these two different environments so that the computer program is a true representation of the expected results on the actual work piece

Starting with the workspace coordinates, the calibration procedure calculates a transformation into field coordinates. The transformation consists of a linear component, as well as a correction component (in the form of a vector field) to address non-linear distortions

The transformation is then used to generate a 2-dimensional correction table that allows accurate conversion of the computer's Cartesian coordinates into workspace coordinates

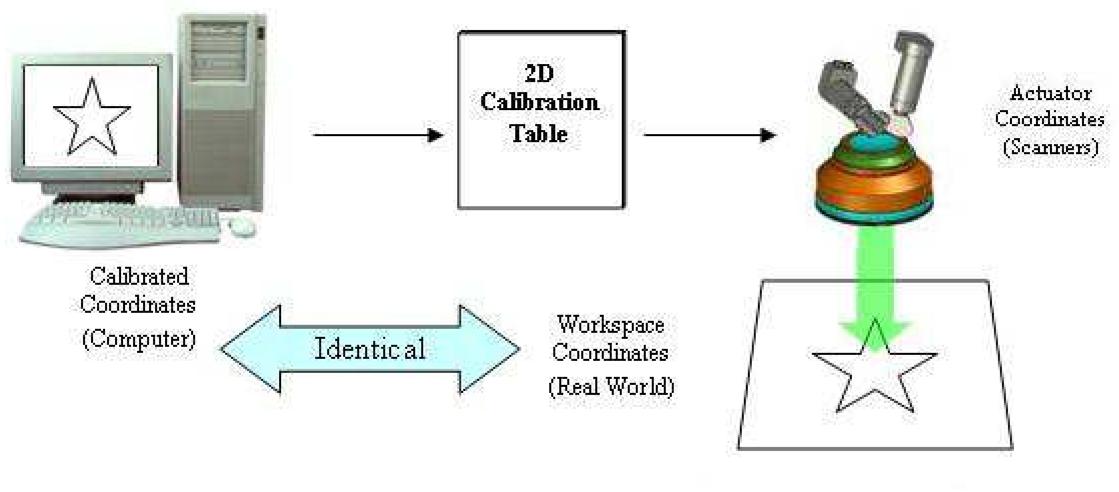

Figure 5: Figure 5: Coordinate Systems, Calibrated

The Correction table's vector field is a set of 2-dimensional vectors where the start of the vector is the desired point (defined in the job), and the end of the vector is the actual point (as needed to achieve the required position on the work piece). If a correction vector does not exist for a specific location, the software will interpolate the surrounding vector field entries in two-dimensional space

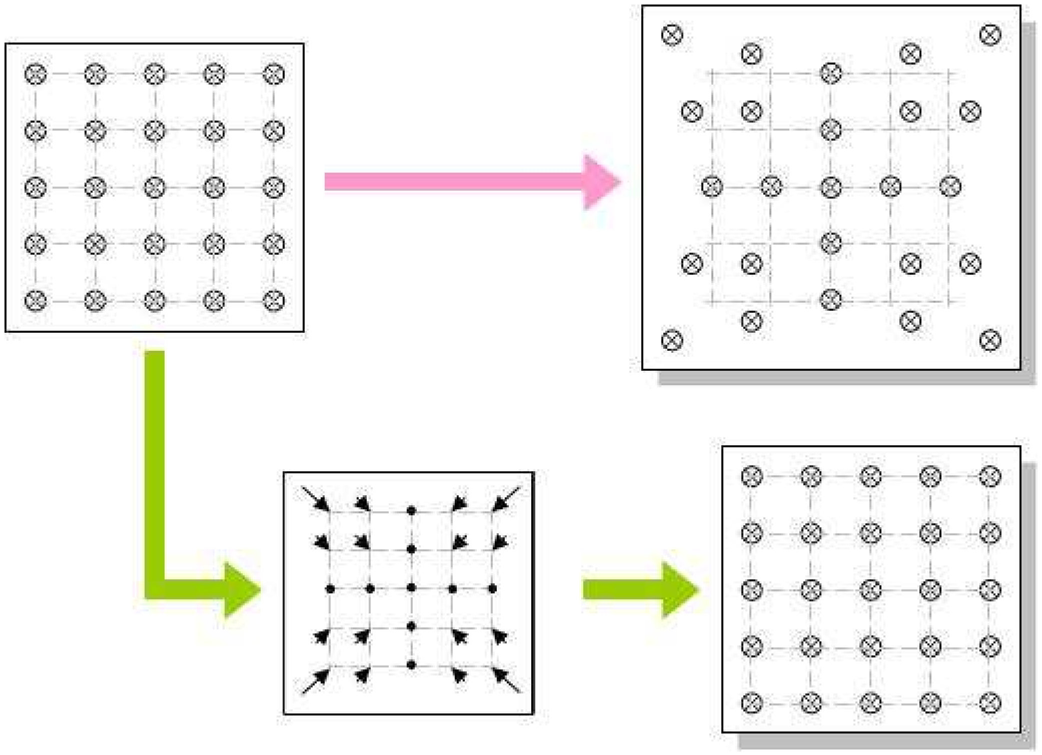

Figure 6: Figure 6: Correction Table Effect

5.3 Vector Marking

A vector marking job consists of a continuous trajectory composed of vectors. The following example illustrates the progression of marking over time by labeling the points visited in numerical order

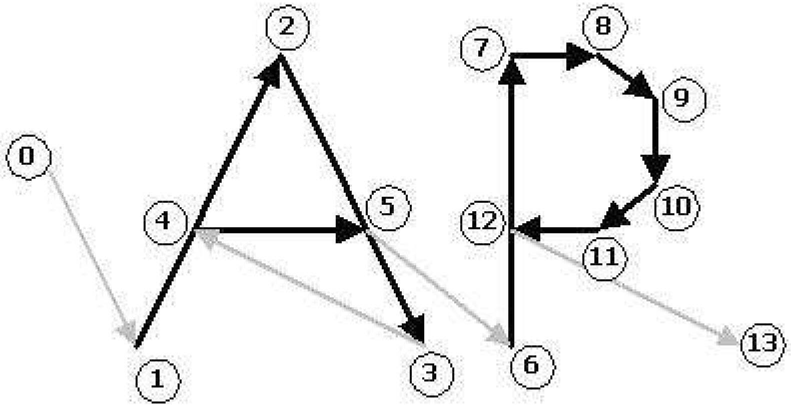

Figure 7: Figure 7: Simple Marking Path Example

In the example the arrows point in the direction of motion. The gray arrows indicate jump vectors, motions where the laser is OFF. The black arrows indicate mark vectors, motions where the laser is ON. The example shows the letters ’AP’ being marked starting at vertex 0 and continuing sequentially to vertex 13

Generally, the coordinate destinations of the mark and jump commands are single precision floating point values. Position Parameters are available for the X, Y and Z-axes. The X and Y-axis position values are converted to command voltages that are subsequently sent to the servo controllers operating the mirrors. The Z-axis typically commands a laser focus control, e.g. a linear translator. If you are simply marking on a planar surface then the traditional method of handling the Z Parameters is to set each value to zero

5.4 Fine Tuning with Delays

5.4.1 Scanning System Physical Limitations

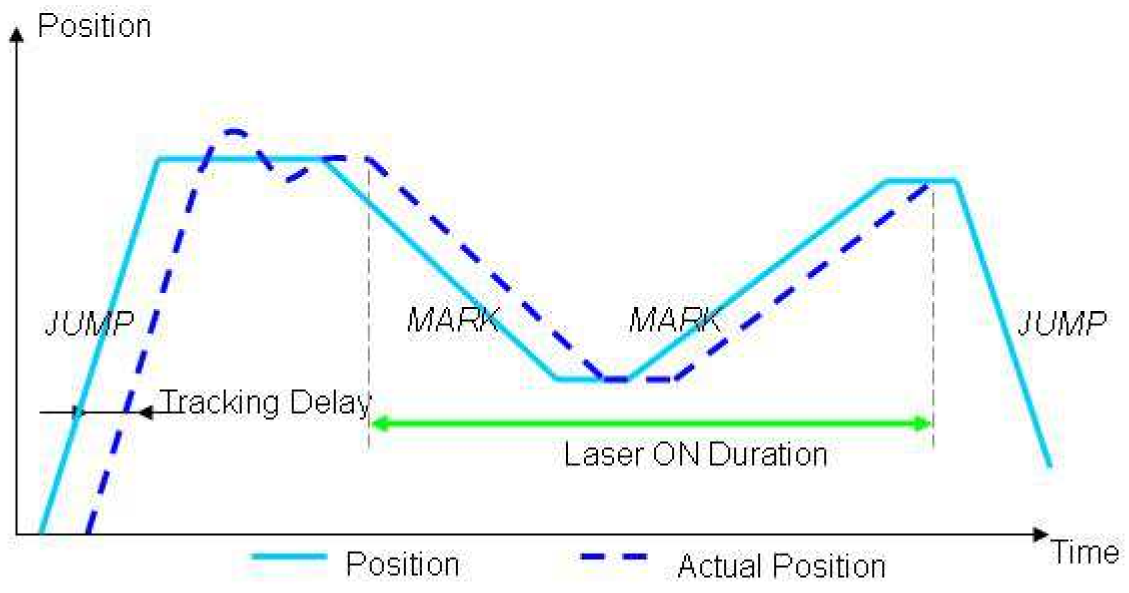

Figure 8: Figure 8: Illustrating Actual vs. Commanded Position Delay

Figure 8 above shows a typical galvo position command signal in which the galvo is directed to move from point to point. It also shows the actual position the galvo achieves which lag behind the commanded position by a "Tracking Delay" (TD)

The cause of the Tracking Delay is that achieving the commanded movement profile would require that the galvo be instantly accelerated to the proper velocity, and then instantly decelerated to a motionless state as it reaches the target position

Such velocity profile would require inducing an infinite current in the windings of the galvo motor which in turn would require the application of an infinite voltage. This is of course impossible, so some delay is required while accelerating to the target velocity, and additional delay while decelerating the mass of the motor, shaft and mirror until it settles at the target position

5.5 Introduction the Delay Parameters

```
\image html theory_fig9.JPG "Figure 9: Introducing the Delay Parameters"
\image latex theory_fig9.eps "Figure 9: Introducing the Delay Parameters"
\n
Figure 9 above introduces the 5 delay Parameters supported by the Universal API. These Parameters allow the user
- Jump Delay: required at the end of a jump (Start-of-Mark) to allow the mirrors to decelerate from the jump spee

- Laser On Delay: required at the Start-of-Mark to allow the mirrors to accelerate to the marking speed when a ne

- Polyline Delay: required between successive mark commands to allow the mirrors to make a directional change wit

- Mark Delay: required at the End-of-Mark to ensure that the mirrors reach the final position of the mark before

- Laser Off Delay: required at the End-of-Mark to keep the laser on for the additional time until the mirrors rea
```

5.5.1 Optimizing the Delay Parameters

As discussed before, the Tracking Delay (TD) reflects the physical limitations of the system and therefore is useful in defining a good starting point for the values of the other five delays as shown in the table below:

Delay Type	Starting Point Value
Laser On Delay	2/3 ∗ TD
Laser Off Delay	4/3 ∗ TD
Jump Delay	2 ∗ TD
Mark Delay	4/3 ∗ TD
Polyline Delay	TD (Nominal)

Table 1: Recommended Delay Values

The above starting values will result in good marks with small distortions. In many applications, some additional distortion is acceptable to achieve better output. For these cases, the following trial-and-error process can be used to optimize the delay Parameters -

- Set the delays to the starting points provided above, mark a sample part and view the results
- If no marking quality problems are seen reduce the delays and repeat the sample
- If there are marking quality problems, address them using the guidelines provided below
- Continue the process until no further improvement can be achieved.

Note that the Laser ON and Laser OFF delays have little or no effect on the total job time. Therefore, try optimizing these delays first, and only then proceed to optimize the Jump Delay, Mark Delay and Polyline Delay

Polyline delay is adjusted by the library proportional to the angle between vectors. Small changes in direction require minimal delay, while large changes, particularly angles greater than $90°$ require long delays

5.5.2 Laser ON Delay Optimization

Modern lasers can be turned on in a period of time that is insignificant in comparison to the time required to move the mirrors. If the laser was turned on at the same time that a new position command was given, the laser would focus on the starting point of the mark while the mirrors are being accelerated. The result would be a higher intensity mark that would not be acceptable for many applications. The Laser ON Delay prevents this problem by delaying laser on until the mirrors begin moving

The following figures show the effect of having too short or too long of a Laser ON Delay.

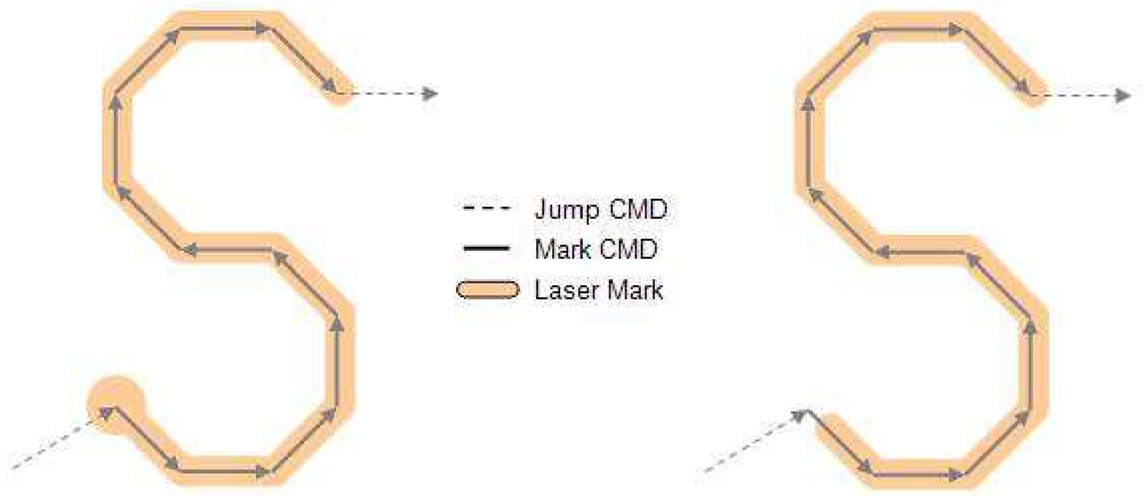

Figure 9: Figure 10 & 11: Laser ON delay too short and too long

Too short of a delay results in burn-in at the start of the mark. With too long of a delay, the beginning of the mark is missing

5.5.3 Laser OFF Delay Optimization

Laser turn off is similar to laser turn on. If the laser was turned off at the same time that a position command is completed, the laser mark would not have reached the intended end point as the mirrors are lagging behind. The result would be a shorter mark that would not be acceptable for many applications. The Laser OFF Delay prevents this problem by delaying turning off the laser until such a point in time when the mirrors complete their move

The following figures show the effect of having too short or too long of a Laser OFF Delay

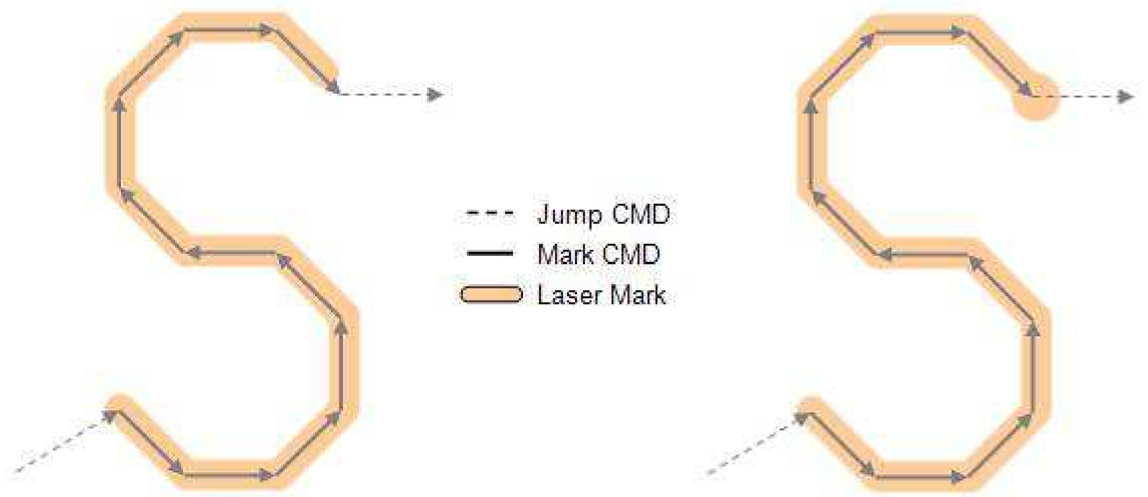

Figure 10: Figure 12 & 13: Laser OFF delay too short and too long

When Laser OFF Delay is set too short, the end of the mark is missing. When Laser OFF Delay is set too long, burn-in effects are seen at the trailing end of the mark

5.5.4 Jump Delay Optimization

The value of the Jump Delay depends on the jump speed and is normally higher than the value of other delays because the mirrors move much faster during the jump. The higher the jump speed the more time the mirrors will require to settle at the end of the jump

The following figures show the effects of having the Jump Delay set too long or too short

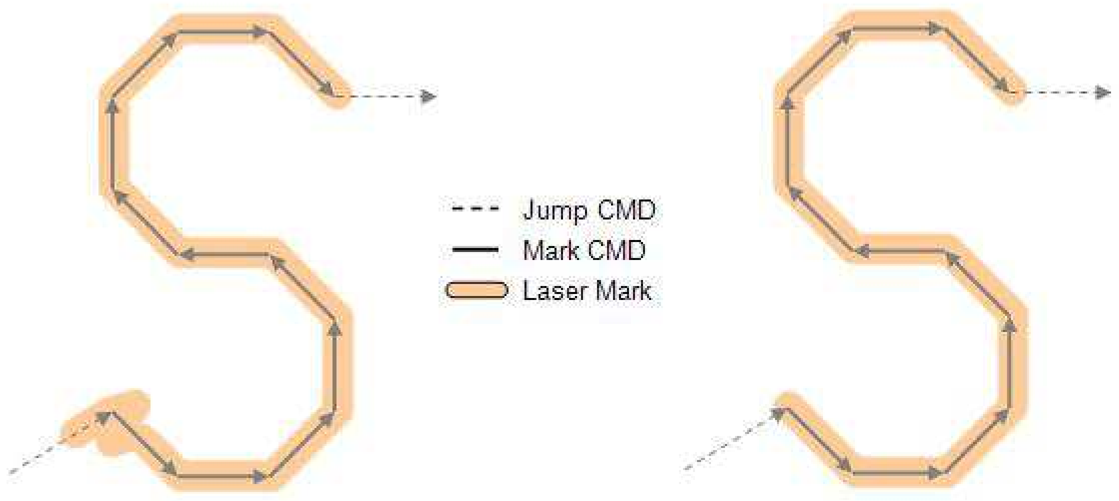

Figure 11: Figure 14 & 15: Jump Delay delay too short and too long

When the Jump Delay is set too short, an oscillation pattern is visible at the beginning of the first mark. When Jump Delay is set too long the line quality is acceptable however the scanning time is longer than necessary

5.5.5 Mark Delay Optimization

During marking the mirrors move at a lower speed than during a jump so the Mark Delay value is normally set lower than the Jump Delay

The following figures show the effects of having the Mark Delay set too short or too long

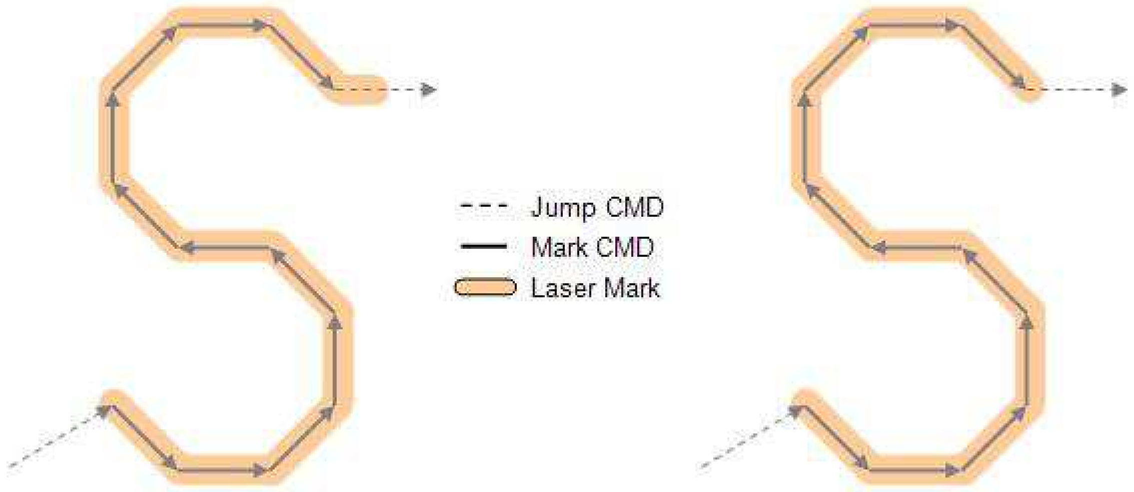

Figure 12: Figure 16 & 17: Mark Delay delay too short and too long

When Mark Delay is set too short, the end of the mark is curved in the direction of the next jump command. No noticeable effects are observed when the Mark Delay is set too long; however, scanning time will be unnecessarily extended

5.5.6 Polyline Delay Optimization

The inertia of the mirrors prevents them from responding immediately to a change in direction. Without a Polyline Delay, the mirror will begin accelerating in the direction of the next mark before it has completed the current one, resulting in a rounded corner. Introducing a Polyline Delay provides time to complete the current mark before starting the next, ensuring a perfect mark

The following figures show the effect of having too short or too long of a Polyline Delay

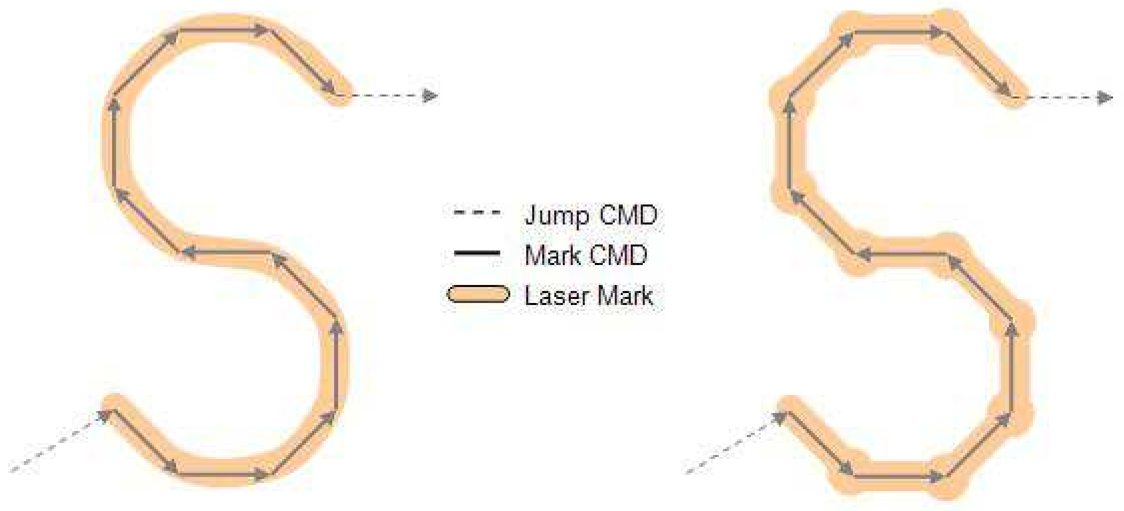

Figure 13: Figure 18 & 19: Polyline Delay delay too short and too long

If Polyline Delay is too short, what should be sharp corners are instead rounded off. If the Polyline Delay is set too long, the corners are burned in

5.5.7 Additional Comments on Polyline Delay

To mark a perfect corner, the required Polyline Delay varies in proportion to the angle θ between the successive marks (Figure). If θ is small, only a short delay is needed, while if θ is larger a longer delay is required

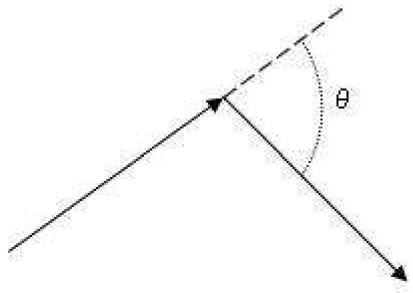

Figure 14: Figure 20: The Angle Between Two Successive Marks

To accommodate the above requirement, the actual Polyline Delay is set according to the formula:

$$\text{Actual Polyline Delay} = \text{Nominal Polyline Delay} * (1 - \cos(\theta))$$

```
<table border>
<tr> <td><b> Angle (degrees) </b></td> <td><b> Actual Polyline Delay </b></td> </tr>
<tr> <td> 0 </td> <td> 0 </td> </tr>
```

```
<tr> <td> 45 </td> <td> Nominal Polyline Delay * 0.29 </td> </tr>
<tr> <td> 90 </td> <td> Nominal Polyline Delay * 1.00 </td> </tr>
<tr> <td> 135 </td> <td> Nominal Polyline Delay * 1.71 </td> </tr>
<tr> <td> 180 </td> <td> Nominal Polyline Delay * 2.00 </td> </tr>
</table border>
Table 2: Actual vs. Nominal Polyline Delay
```

6 Rotate Transform

In-depth transform review Page is empty because content does not yet exist

7 Scale Transform

In-depth transform review Page is empty because content does not yet exist

8 Translate Transform

In-depth transform review Page is empty because content does not yet exist

9 Setting up the Universal API SDK

-Getting Started

-Compiling and Linking your Application to the SDK

-Synchronization Domains (SyncDomains) and Handles

-Lists

9.1 Getting Started

In this section we cover and explain the basics of the Universal API (UAPI) SDK setup after installation.

CTI UAPI Software Development Kit (SDK) is composed of a Dynamic Link Library (DLL) and associated header and lib files that connects to the hardware.

The header, .Dll and other configuration files can be found in the root directory of installation for the Common Tools SDK (the default root directory is C:\Program Files\CTI\Common).

In this section we cover and explain the basics of the Universal API (UAPI) SDK setup after installation.

CTI' UAPI Software Development Kit (SDK) is composed of the ScanPack Dynamic Link Library (DLL) and associated header and lib files.

The header, .Dll and other configuration files can be found in the root directory of installation for the Universal API SDK (the default root directory is C:\Program Files\CTI\Universal API).

9.2 Compiling and Linking your Application to the SDK

The environment variable CTI_COMMON_ROOT is set by the installer. We recommend using it to configure your development environment to find the required header and lib files.

The environment variable SCANPACK_SC500_ROOT is set by the installer. We recommend using it to configure your development environment to find the required header and lib files.

9.3 Synchronization Domains (SyncDomains) and Handles

SyncDomains and Handles are referred to often in the next section (Using the UAPI SDK). This section covers the meaning of a SyncDomain.

In the most general sense, a SyncDomain is a set of Servo drivers, I/O device(s) and any other device(s) that operate synchronously at the clock rate of a system. The SyncDomain consists of command channels, running in lock step with one another, implemented on one or more separate devices. Devices on two separate SyncDomains do not run synchronously with one another as they have different clocks.

In the context of the scan controller, a SyncDomain consists of up to three motor axes, laser control and the I/O implemented on the board, and is essentially synonymous with a Servo scan head.

A SyncDomain Handle is used in the UAPI to refer to a specific SyncDomain in systems containing multiple Sync-Domains, similar to a file handle.

9.4 Lists

The Universal API utilizes the execution of multiple lists of instructions so that one list is executed while the next list is being created by the software.

It is recommended that lists be kept within 8000 lines of code. While not limited to this length, conforming to this constraint is beneficial for two reasons:

- It allows for backwards compatibility with existing CTI controllers, enabling for easy replacement by CTI's current generation controllers.
- In programs that implement sequential list executions, lists around the same size will improve execution efficiency.

Lists encapsulate all List specific API command calls, and when executed will terminate all commands stored within that list (polyline functionality, laser settings and galvo position data are all terminated or reset to default).

The Universal API utilizes the execution of multiple lists of instructions so that one list is executed while the next list is being created by the software.

The concept of a list is to provide a collection of operations such as marks, jumps, etc., in a precisely timed manner. The list is built up in memory and then "closed". It can then be committed for execution. Once the job is started, immediate mode commands such as UA_goto_xyz must not be invoked, since they will interfere with execution of the list in progress, possibly causing it to fail. It is recommended that lists be kept within 8000 lines of code. While not limited to this length, conforming to this constraint is beneficial for two reasons:

- It allows for backwards compatibility with existing CTI controllers, enabling for easy replacement by CTI's current generation controllers.

- In programs that implement sequential list executions, lists around the same size will improve execution efficiency.

Lists encapsulate all List specific API command calls, and when executed will terminate all commands stored within that list (polyline functionality, laser settings and galvo position data are all terminated or reset to default).

10 Using the Universal API SDK

In this section we cover the basic coding conventions used when developing with the UAPI SDK. All code snippets can be directly referenced back to the API documentation or the example functions list.

- UAPI Naming Conventions
- Error checking
- Board Status
- Getting a handle
- System calibration
- Transforms
- List execution
- Trajectory and Primitives
- System I/O

10.1 UAPI Naming Conventions

The UAPI commands have been deliberately named with certain markers to identify different command types and their application use.

Structures such as Point_3d_t are defined using the **_t** suffix as a Type or Data Structure within the UAPI.

Functions are defined with either the **UA** or **UAL** prefix for "Universal API" and "Universal API List" respectively.

UA prefixed functions handle data, parameters and, settings over the scope of the entire job execution. **UA** commands are implemented on an ASAP basis, whether a list is executing or not. It is not recommended to issue a **UA** command while a list is executing.

UAL prefixed functions handle only data, parameters and, settings within the scope of a specified list. For more information regarding lists see List execution.

10.2 Error checking

The UAPI provides error checking using C style return codes. Error checking is relatively simple: any function can be checked for its return code to see the state of the operation. Typically we only want to continue the job execution when the code returns UA_OPERATION_OK. We can use a single if statement to see if the function does *not* return UA_OPERATION_OK:

```
UA_RC rc = UA_execute_list(sdh, listno, 0);
if (rc != UA_OPERATION_OK)
        return rc;
```

In the code above, we commit a list for execution and check to ensure that it completed without error. In the case that it does return an error the error code is returned to the parent function. In most cases, this kind of simple error check-with a proper reaction from the user code, is all that is needed.

In the following sections, code snippets are made with error checking in mind and are to be used as a reference for structuring your own error checking.

For a complete list of return codes and their explanations, see UA_RC

Additionally, monitoring board status allows for more in-depth and specific error checking. See the Board Status page for more information.

10.3 Board Status

Checking the board status is a critically important feature required for any application. The UAPI offers data on the board's ..... This data can be applied for a large set of functionality. Below are some common applications:

10.4 Getting a handle

-Device Name Known -Device Name Unknown -Device Not Connected

10.4.1 Device Name Known

The simplest way to retrieve a SyncDomainHandle is by knowing the name of the device. In the below code, establishing this kind of a connection is shown:

```
SyncDomainHandle sdh = 0;
UA_RC rc = UA_get_handle("CTI_HEAD_A", &sdh);
if (rc != UA_OPERATION_OK) return rc;
```

This method is used when all device names in the system are known. Here we have used the generic CTI_HEAD_A name which all CTI heads ship with.

If the name is not known and this function is used, the API will pick the first head connected to the host and its handle value. If there is only one head in the system, this is not an issue since there is no ambiguity.

10.4.2 Device Name Unknown

This method is used when there are multiple device names connected to the host that are not known or you wish to check a misconfigured system (such as two devices with the same name).

```
SyncDomainInfo sdis[MAX_HEADS];
int numValidSyncDomains = 0;

UA_RC rc = UA_get_sync_domain_info(sdis, MAX_HEADS,
       &numValidSyncDomains);
if (rc != UA_OPERATION_OK) return rc;

SyncDomainHandle sdh = sdis[0].handle;
```

Here we use an array of SyncDomainInfo structures, query for information for every connected device, and select the first in the array. We choose the first in the array because only the first device connected is accessible to be controlled by the API due to current limitations.

10.4.3 Device Not Connected

In the case that the device is not connected to the host before execution, we create a function call to wait for the device to be connected before acquiring a SyncDomainHandle.

```
UA_RC wait_for_device_connection(SyncDomainHandle *sdh)
{
        try
        {
                UA_RC rc = UA_OPERATION_OK;
                SyncDomainInfo sdis[MAX_HEADS];
                int numValidSyncDomains = 0;

                while (numValidSyncDomains < 1)
                {
                        rc = UA_get_sync_domain_info(
      sdis, MAX_HEADS, &numValidSyncDomains);
                        if (rc != UA_OPERATION_OK) return rc;
                        Sleep(100);
                }

                *sdh = sdis[0].handle;
        }
        catch ( ... )
        {
                return UA_UNKNOWN_ERROR;
        }
        return UA_OPERATION_OK;
}
```

Here we use a try-catch block to ensure we do not miss any unhandled exceptions during runtime, returning UA_UNKNOWN_ERROR if an explicit error check is overlooked.

This function uses the same method described in the Device Name Unknown page but continuously does so until a SyncDomain connection is found.

10.5 System calibration

-System Field of View (FOV) -Delay parameters -Laser parameters

10.5.1 System Field of View (FOV)

After working with Cambridge Technology, Inc to get the desired FOV number which will reflect the units of measure and correction data file, use the example code below to configure the calibration data:

```
UA_RC rc = UA_OPERATION_OK;
```

```
const float target_FOV = 100.0F;
const float scale = (1/target_FOV);
Xform_2d_t ws_xform = UA_scale_xform_2d(scale, scale
      );

rc = UA_set_workspace_transform_2d(sdh, &ws_xform)
      ;
if (rc != UA_OPERATION_OK) return rc;

rc = UA_load_calibration_file (sdh, "NoCorrections.xml"
      );
if (rc != UA_OPERATION_OK) return rc;
```

In the code above, we create a scale by taking the inverse of the FOV and generating a 2-dimensional workspace transform from this scale. We then apply the workspace transform to the job and load the calibration xml file.

10.5.2 Delay parameters

Application-related parameters set speeds and delays related to the entire job.

Parameter sets for this section include jump speed and delay, mark speed and delay, and polyline delay for the galvos.

```
UA_RC rc = UA_OPERATION_OK;
float jump_speed = 0.9f, // seconds
        mark_speed = 0.8f, // seconds
        jump_delay = 250e-6f, // seconds
        mark_delay = 260e-6f, // seconds
        polyline_delay = 80e-6f; // seconds

rc = UA_set_jump_speed(sdh, jump_speed);
if (rc != UA_OPERATION_OK) return rc;

rc = UA_set_jump_delay(sdh, jump_delay);
if (rc != UA_OPERATION_OK) return rc;

rc = UA_set_mark_speed(sdh, mark_speed);
if (rc != UA_OPERATION_OK) return rc;

rc = UA_set_mark_delay(sdh, mark_delay);
if (rc != UA_OPERATION_OK) return rc;

rc = UA_set_polyline_delay(sdh, polyline_delay);
if (rc != UA_OPERATION_OK) return rc;
```

10.5.3 Laser parameters

The laser should be configured before lasing to some default values. If you need to reconfigure these parameters while lasing, these functions have a UAL equivalent for list-specific control.

```
UA_RC rc = UA_OPERATION_OK;
float laser_period = 100e-6f, // seconds
        laser_on_width = 50e-6f, // seconds
        laser_standby_width = 1e-6f, // seconds
        laser_power = 100.0f, // %
        pulse_width_none = 0; // the second pulse width is only used for
       special applcations

rc = UA_set_laser_power(sdh, laser_power);
if (rc != UA_OPERATION_OK) return rc;

rc = UA_set_laser_power_standby(sdh, laser_power);
if (rc != UA_OPERATION_OK) return rc;

rc = UA_set_laser_timing (sdh, laser_period, laser_on_width,
       pulse_width_none);
if (rc != UA_OPERATION_OK) return rc;
```

```
rc = UA_set_laser_timing_standby(sdh,laser_period,
        laser_standby_width, pulse_width_none);
if (rc != UA_OPERATION_OK) return rc;
```

For more information on list-specific laser parameters see <ref>.

First Pulse Suppression

10.5.3.1 First Pulse Suppression

First Pulse Suppression (FPS) is used to suppress the initial energy build up found in lasers throughout the industry.

Please refer to your laser manual for the FPS width and advance times, all values used in example code are arbitrarily selected for example purposes **only**.

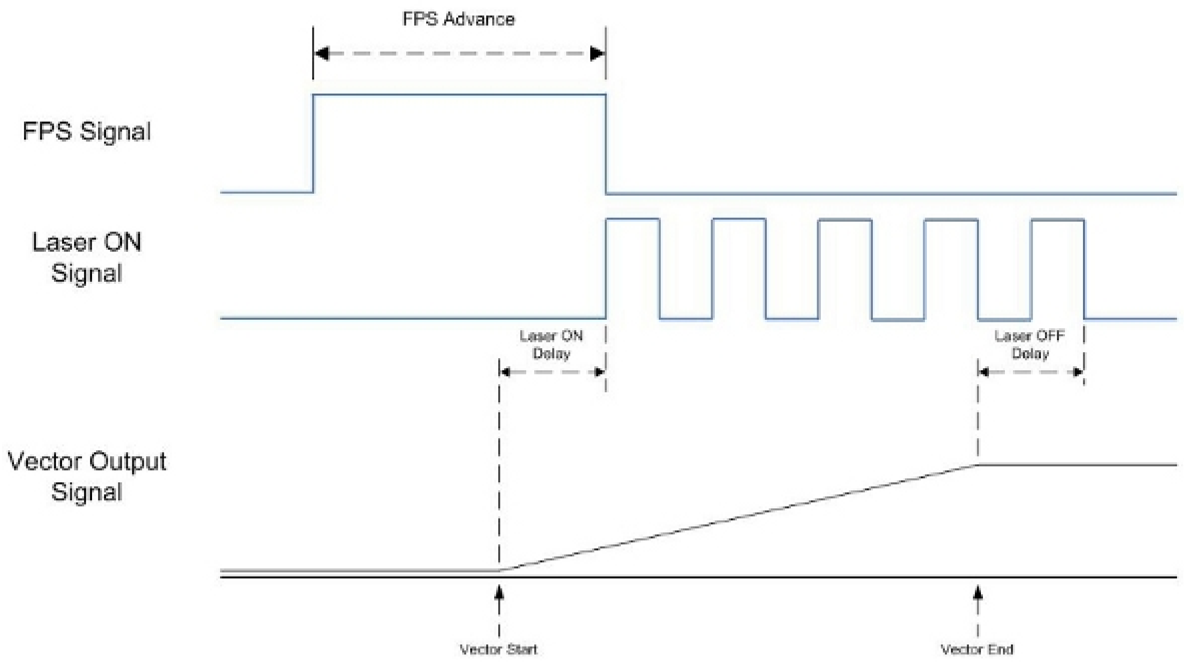

Figure 15: Figure 21: First Pulse Suppression

In the context of the UAPI, the FPS parameter is a subset of and should follow the laser parameters:

```
UA_RC rc = UA_OPERATION_OK;
float FPSwidth = 50e-6f;
float FPSadvance = 50e-6f;

rc = UA_set_FPS_timing(sdh, FPSwidth, FPSadvance);
if (rc != UA_OPERATION_OK) return rc;
```

Note

First Pulse Suppression requires a specific board configuration. If you require this feature and do not know if your configuration is setup correctly, please contact your CTI sales associate.

10.6 Transforms

This section **only** covers the UAPI's implementation of transforms. For an in-depth look at the theory behind transforms, see the Transforms section

There are three separate types of transforms available for use plus a combinational transform modifier. In all cases the transforms are straight forward to use:

10.6.1 The Rotate Transform

The rotate transform angle is by default in radians and moves in the counter-clockwise direction for a positive θ value.

```
UA_RC rc = UA_OPERATION_OK;
float theta = 1.6f; // default radians
UA_rotate_xform_2d(theta);

// alternatively, the transform data can be saved:
Xform_2d_t rotate_xform = UA_rotate_xform_2d(theta)
      ;
```

10.6.2 The Scale Transform

The scale transform units are dependent on the field of view and working distance set. See System Field of View for more information.

```
UA_RC rc = UA_OPERATION_OK;
// units are FOV/workspace transform dependent
float x_scale = 2.0f,
        y_scale = 1.4f;
UA_scale_xform_2d(x_scale, y_scale);

// alternatively, the transform data can be saved:
Xform_2d_t scale_xform = UA_scale_xform_2d(x_scale,
      y_scale);
```

10.6.3 The Translate Transform

The translate transform units are dependent on the field of view and working distance set. See System Field of View for more information.

```
UA_RC rc = UA_OPERATION_OK;
// units are FOV/workspace transform dependent
float x_dist = 2.0f,
        y_dist = 1.4f;
UA_translate_xform_2d(x_dist, y_dist);

// alternatively, the transform data can be saved:
Xform_2d_t translate_xform = UA_translate_xform_2d
      (x_dist, y_dist);
```

10.6.4 The Multiplier Transform

The multiplier transform can combine two Xform_2d_t structures together at one time.

```
UA_RC rc = UA_OPERATION_OK;
float theta = 1.6f; // default radians
// units are FOV/workspace transform dependent
```

```
float x_scale = 2.0f,
        y_scale = 1.4f,
        x_dist = 2.0f,
        y_dist = 1.4f;

// create and store transform data
Xform_2d_t rotate_xform,
        scale_xform,
        translate_xform,
        tmp_xform, // temporarily stores the combination of two transform's
       data
        obj_xform; // stores the combination of all three transforms

rotate_xform = UA_rotate_xform_2d(theta);
scale_xform = UA_scale_xform_2d(x_scale, y_scale);
translate_xform = UA_translate_xform_2d(x_dist, y_dist);

// create a combinational transform
tmp_xform = UA_mult_xform_2d(&rotate_xform, &translate_xform);
obj_xform = UA_mult_xform_2d(&tmp_xform, &scale_xform);
```

The order in which the transforms are multiplied is important, see Theory of Operation for more information.

The scale transform is applied last to properly (as you would expect) apply the scale to the rotated and translated object (here the object is some lased path). Once the object transform data is created, it can be applied in the scope of a single list:

```
UA_RC rc = UA_OPERATION_OK;
.
.
.
rc = UAL_set_object_transform_2d(sdh, &obj_xform);
if (rc != UA_OPERATION_OK) return rc;
.
.
.
```

10.7 List execution

In order to create a list for execution the start and end of the list must be specified within the code. This is in place to allow for multiple list execution.

```
UA_RC rc = UA_OPERATION_OK;
// start the list
rc = UAL_set_start_list(LIST_1); // alternatively you
       can set it to LIST_2 for multiple lists
if (rc != UA_OPERATION_OK) return rc;
.
.
.
// here is the body of the list, where UAL commands should be used
.
.
.
// now end it
rc = UAL_set_end_of_list();
if (rc != UA_OPERATION_OK) return rc;

// if we are ready, we can commit the list to hardware for execution
rc = UA_execute_list(sdh, LIST_1, 0);
if (rc != UA_OPERATION_OK) return rc;
```

10.8 Trajectory and Primitives

The UAPI only facilitates point and vector primitives and does not offer any pre-built shape primitives. This gives you complete control over your marking application without restriction. All marking and movement commands in the

UAPI must be executed within the scope of a list.

In the following example we will define a box trajectory using both point and vector primitives:

```
// assuming we have already configured our system and laser delays and
       workspace transforms
UA_RC rc = UA_OPERATION_OK;
const int MAX_PATH_TRACES = 10, // how many times to trace the same box
       primitive
        shoot_time = 1.5; // 1.5 seconds to lase at each point
// first define the corner points of the box, we will reuse these for all cases
       of movement
float box_length = 10.0f; // units dependent on FOV and workspace transform
Point_3d_t bottom_left  = {0, 0, 0};
Point_3d_t bottom_right = {box_length, 0, 0};
Point_3d_t top_left     = {0, box_length, 0};
Point_3d_t top_right    = {box_length, box_length, 0};

// define which list our trajectories belong to
LISTNO box_traj = LIST_1,
        point_traj = LIST_2;

// create a marked box trajectory to be marked 10 times
UAL_set_start_list(sdh, box_traj);
for (int mark_times = 0; mark_times < MAX_PATH_TRACES; mark_times++)
{
        UAL_jump(sdh, bottom_left); // always jump to the starting
       location to ensure the mirrors are in place first
        // now start to mark the box edges
        UAL_mark(sdh, bottom_left);
        UAL_mark(sdh, bottom_right);
        UAL_mark(sdh, top_right);
        UAL_mark(sdh, top_left);
        UAL_mark(sdh, bottom_left); // we need to mark to the bottom
       left point again
                                                                // to mark from
       the top left point to the bottom left point
}
UAL_set_end_of_list(sdh);

// create a point box trajectory to be marked 10 times
UAL_set_start_list(sdh, box_traj);
for (int mark_times = 0; mark_times < MAX_PATH_TRACES; mark_times++)
{
        UAL_jump(sdh, bottom_left); // always jump to the starting
       location to ensure the mirrors are in place first
        // now start to mark the box points
        UAL_shoot(sdh, shoot_time); // turn on the laser for 1.5
       seconds

        UAL_jump(sdh, bottom_right);
        UAL_shoot(sdh, shoot_time);

        UAL_jump(sdh, top_right);
        UAL_shoot(sdh, shoot_time);

        UAL_jump(sdh, top_left);
        UAL_shoot(sdh, shoot_time);
        // since we lased each point, we do not need to return to the bottom
       left corner
        //  and lase it again let the next iteration mark the bottom left
       corner
}
UAL_set_end_of_list(sdh);

// now that we have both lists defined, we can submit them for execution
UA_execute_list(sdh, box_traj, 0);

// wait for the list to finish execution before continuing to the next list
// using one of our status helper functions
wait_for_list_execution(sdh, box_traj);

UA_execute_list(sdh, point_traj, 0);

// wait for the list to finish again before continuing
```

```
wait_for_list_execution(sdh, point_traj);
```

10.9 System I/O

I/O in the UAPI is done by asserting and clearing different bits representing an I/O pin on the device. In this section we will see how to use the I/O port commands to assert and clear the Master Oscillator bit and Guide Laser bits. The below code shows how to draw a box with the guide laser then lase each point of the box by setting the Master Oscillator bit.

In this example, we will assume the trajectory lists used in the Trajectory and Primitives page are loaded and awaiting an auxiliary I/O pin to signal execution:

```
// All I/O pin/bit data assumes a standard configuration for the SC500, if you
       are
// using the UAPI with another controller, please verify your values!!

UA_RC rc = UA_OPERATION_OK;
uint32 digital_io_val = 0;
const uint32 aux_io_1 = 1<<5; //Auxiliary I/O pin 13, Laser pin 1
const uint32 aux_io_2 = 1<<6; //Auxiliary I/O pin 14, Laser pin 2
// wait for an auxiliary I/O pin value to be set
do
{
        rc = UA_read_io_port(sdh, DIGITAL_INPUTS,
      &digital_io_val);
        if (rc != UA_OPERATION_OK) return rc;
        Sleep(100);
}
while (digital_io_val == 0)

// check to see which I/O pin was set and execute a different list
// which is assumed to already be loaded
if (digital_io_val & aux_io_1)
        UA_execute_list(sdh, box_traj, 0);
else if (digital_io_val & aux_io_2)
        UA_execute_list(sdh, point_traj, 0);
```

11 Examples

This section includes all examples

Every example should be its own file (default here is the code_example.cpp file) and be copy-paste compileable as long as the development environment is setup. Examples should be a full program, no uncompileable snippets or lone functions are allowed.

-
  ```
  /*****************************************************************************
  Examples should be full source files linked to from the mainpage.h file and
  included in the api_examples subpage. Example sources should also include
  full inline comment and require only high-level external comments.
  The code below is not exemplary of good documentation.
  *****************************************************************************/

  // This function counts to fifty from the input integer value 'val'.
  // Returns true if the count was completed successfully.
  bool count_to_fifty(int val)
  {
          // ensure we can count the input value
          if (val > 50)
                  return false;

          // count to 50
          for (int i = val; i < MAX_COUNT; i++)
                  i++;
  ```

```
        // return the operation successfully
        return true;
}
```

"Code Example"

12 Change Log

12.1 1/17/2013 Rev.B

- Re-tooling of the Universal API Reference manual to new style.

13 Module Documentation

13.1 CTI Universal Controller Adapter

Data Structures

- struct UA_version_t

  Structure that is used for version information. More...
- struct UA_status_t

  Used to hold the status data from the scan controller. More...
- struct CALTABLE

  Square Grid. More...
- struct Point_3d_t
- struct Point_2d_t
- struct Xform_2d_t

  Describes a two dimensional linear tranform. More...
- struct SyncDomainInfo

  Description of a synchronization domain. More...

Macros

- #define **UA_SPEC** __declspec(dllimport)
- #define **MAX_AVAILABLE_SYNC_DOMAINS** (16)

Typedefs

- typedef uint32 SyncDomainHandle

  Handle to identify hardware.

Enumerations

- enum { **UA_STRING_LEN** = 80 }
- enum { **MAX_HEADS** = 16 }
- enum { **MAX_INFO_STRING_LEN** = 1024 }
- enum STATUS_LIST {
  STATUS_INACTIVE = -0x0001,
  STATUS_EMPTY = 0x0000,
  STATUS_OPEN = 0x0001,
  STATUS_LOAD = 0x0002,
  STATUS_STANDBY = 0x0004,
  STATUS_READY = 0x0008,
  STATUS_IN_JOB_QUEUE = 0x0010,
  STATUS_EXECUTING = 0x0011,
  STATUS_NOT_AVAIL = 0x0012 }

  List Status.
- enum STATUS_QUEUE {
  QUEUE_EMPTY = 0x0000,
  QUEUE_LIST_EXECUTING = 0x0001,
  QUEUE_HARDWARE_HOLDING = 0x0003 }

  Enumeration that describes the status of the list Queue.
- enum **PIXEL_MODE** {
  STEP_AND_FIRE = 0,
  MOVE_WHILE_FIRING = 1 }
- enum { **UA_GRID_SIDE** = 65 }
- enum { **UA_GRID_SIZE** = (UA_GRID_SIDE ∗ UA_GRID_SIDE) }
- enum UA_RC {

UA_OPERATION_OK = 0,
UA_UNKNOWN_ERROR = 1,
UA_UNKNOWN_EXCEPTION_CAUGHT = 2,
UA_BAD_CALIBRATION_FILE = 3,
UA_NO_HARDWARE_FOUND = 4,
UA_PARAM_OUT_OF_LIMITS = 5,
UA_UNABLE_TO_SPAWN_THREAD = 6,
UA_CIRCULAR_QUEUE_ENABLED = 7,
UA_UNABLE_TO_WRITE_TEMP_FILE = 8,
UA_BAD_CORRECTION_FILE = 9,
UA_INVALID_LIST_NUMBER = 10,
UA_UNABLE_TO_QUEUE = 11,
UA_UNABLE_TO_PROCESS_QUEUE = 12,
UA_NO_LOADABLE_LIST = 13,
UA_HARDWARE_NOT_READY = 14,
UA_HARDWARE_NOT_CONNECTED = 15,
UA_HARDWARE_ALREADY_EXCLUSIVE = 16,
UA_HARDWARE_NOT_CAPABLE = 17,
UA_UAPI_NOT_INITIALIZED = 18,
UA_PORT_NOT_VALID = 19,
UA_NAME_REQUIRED_FOR_MULTIPLE_SYNC_DOMAINS = 20,
UA_NO_DLLS_FOUND = 21,
UA_BROADCAST_MONITOR_NOT_RUNNING = 22,
UA_LIST_NOT_CLOSED = 23,
UA_EMPTY_LIST = 24,
UA_CONNECTION_LOST = 25,
UA_INVALID_SDH = 26,
UA_PARAMETERSET_FAIL = 27,
UA_CONFLICTING_PARAMETER_VALUES = 28,
UA_FILE_NOT_FOUND = 29,
UA_BAD_LICENSE_FILE = 30,
UA_INVALID_CONNECTION_OPTION = 31,
UA_NO_VERSION_AVAILABLE = 32 }

Return codes from calls to this library.

- enum LISTNO {
NULL_LIST = 0,
LIST_1 = 1,
LIST_2 = 2 }
- enum Digital_Port_ID {
DIGITAL_INPUTS = 0,
DIGITAL_INPUTS_EXT,
DIGITAL_OUTPUTS,
DIGITAL_OUTPUTS_EXT,
DIGITAL_LASER_POWER }

  I/O ports.
- enum Analog_Port_ID {
ANALOG_OUT1 = 0,
ANALOG_OUT2,
ANALOG_IN1,
ANALOG_IN2 }

Analog Ports.

- enum THREAD {
  LIBRARY_THREAD = 0,
  CALLER_THREAD = 1 }

  Thread control.
- enum LASER_ENABLE {
  DISABLE_LASER_CONTROL,
  ENABLE_LASER_CONTROL }

  Laser control modes.
- enum LIST_MODE {
  SINGLE_PASS = 0,
  CIRCULAR_QUEUE = 1 }

  Queue behavior control.
- enum CORRECTION_TABLE_ID {
  CORR_TABLE_1 = 1,
  CORR_TABLE_2 = 2 }

  Correction table identifier.
- enum **CONNECT_OPTIONS** {
  CONNECT_NON_EXCLUSIVE = 0x0000,
  CONNECT_EXCLUSIVE = 0x0001 }
- enum CONTROLLER_TYPE {
  EC1000 = 0,
  SC500 = 1,
  L2_GSB = 2 }

  Describes the type of controller connected to or to connect to.
- enum CONTROLLER_STATUS {
  STATUS_AVAILABLE = 0,
  STATUS_ALREADY_EXCLUSIVE_CONNECTED = 1,
  STATUS_UNAVAILABLE = 128 }

  Describes the status of the controller.

Functions

- UA_SPEC UA_RC __stdcall UA_open ()

  Initializes the interface. Must be called first before any other function.
- UA_SPEC UA_RC __stdcall UA_get_version (UA_version_t ∗p_version)

  Retrieves version information of the library that implements this API.
- UA_SPEC UA_RC __stdcall UA_get_name (char ∗name, SyncDomainHandle sdh)

  Returns the friendly name of the controller associated with the SyncDomain.
- UA_SPEC UA_RC __stdcall UA_get_handle (const char ∗name, SyncDomainHandle ∗p_sdh)
- UA_SPEC UA_RC __stdcall UA_get_handles (SyncDomainHandle handles[ ], int ∗length)

  Retrieves an array of handles to all synchronization domains.
- UA_SPEC UA_RC __stdcall UA_get_sync_domain_info (SyncDomainInfo sdi[ ], const int len_sdi_array, int ∗p_numValidSyncDomains)

  Retrieves an array of structures describing all synchronization domains.
- UA_SPEC UA_RC __stdcall UA_get_controller_info (SyncDomainHandle sdh, char ∗info, size_t info_size)

  Retrieves a null terminated string with extended controller information.

- UA_SPEC UA_RC __stdcall UA_connect (SyncDomainHandle sdh, CONNECT_OPTIONS options)
- UA_SPEC UA_RC __stdcall UA_disconnect (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UAL_set_start_list (SyncDomainHandle sdh, LISTNO listno)
- UA_SPEC UA_RC __stdcall UAL_set_end_of_list (SyncDomainHandle sdh)

  Ends a command list. To send the list to the execution queue call UA_execute_list().
- UA_SPEC UA_RC __stdcall UAL_jump (SyncDomainHandle sdh, Point_3d_t point)
- UA_SPEC UA_RC __stdcall UAL_mark (SyncDomainHandle sdh, Point_3d_t point)
- UA_SPEC UA_RC __stdcall UAL_shoot (SyncDomainHandle sdh, float time)
- UA_SPEC UA_RC __stdcall UAL_wait (SyncDomainHandle sdh, float time)
- UA_SPEC UA_RC __stdcall UAL_wait_sync (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UAL_write_io_port (SyncDomainHandle sdh, Digital_Port_ID id, uint32 value)
- UA_SPEC UA_RC __stdcall UAL_clear_io_port (SyncDomainHandle sdh, Digital_Port_ID id, uint32 bitmask)
- UA_SPEC UA_RC __stdcall UAL_assert_io_port (SyncDomainHandle sdh, Digital_Port_ID id, uint32 mask)
- UA_SPEC UA_RC __stdcall UAL_write_analog_port (SyncDomainHandle sdh, Analog_Port_ID port, float value)
- UA_SPEC UA_RC __stdcall UAL_trigger_probes (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UAL_set_object_transform_2d (SyncDomainHandle sdh, const Xform_2d_t ∗p_obj_xform)
- UA_SPEC UA_RC __stdcall UA_get_status (SyncDomainHandle sdh, UA_status_t ∗p_status)

  Retrieves list status.
- UA_SPEC UA_RC __stdcall UA_execute_list (SyncDomainHandle sdh, LISTNO listno, THREAD thread)
- UA_SPEC UA_RC __stdcall UA_stop_execution (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UA_set_workspace_transform_2d (SyncDomainHandle sdh, const Xform_2d_t ∗p_ws_xform)
- UA_SPEC UA_RC __stdcall UA_set_FPS_timing (SyncDomainHandle sdh, float width, float advance)

  Negative advance indicates pulse is issued after laser modulation begins.
- UA_SPEC UA_RC __stdcall UAL_set_FPS_timing (SyncDomainHandle sdh, float width, float advance)

  Negative advance indicates pulse is issued after laser modulation begins.
- UA_SPEC UA_RC __stdcall UA_set_laser_delays (SyncDomainHandle sdh, float laserOn, float laserOff)
- UA_SPEC UA_RC __stdcall UA_set_jump_speed (SyncDomainHandle sdh, float speed)
- UA_SPEC UA_RC __stdcall UA_set_mark_speed (SyncDomainHandle sdh, float speed)
- UA_SPEC UA_RC __stdcall UA_set_scanner_delays (SyncDomainHandle sdh, float Jump, float Mark, float Polygon)
- UA_SPEC UA_RC __stdcall UA_set_laser_timing_standby (SyncDomainHandle sdh, float Period, float PulseWidth1, float PulseWidth2)
- UA_SPEC UA_RC __stdcall UAL_set_laser_timing_standby (SyncDomainHandle sdh, float Period, float PulseWidth1, float PulseWidth2)
- UA_SPEC UA_RC __stdcall UA_set_laser_timing (SyncDomainHandle sdh, float Period, float PulseWidth1, float PulseWidth2)
- UA_SPEC UA_RC __stdcall UAL_set_laser_timing (SyncDomainHandle sdh, float Period, float PulseWidth1, float PulseWidth2)
- UA_SPEC UA_RC __stdcall UA_set_laser_power (SyncDomainHandle sdh, float power)
- UA_SPEC UA_RC __stdcall UAL_set_laser_power (SyncDomainHandle sdh, float power)
- UA_SPEC UA_RC __stdcall UA_laser_signal_on (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UA_laser_signal_off (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UAL_laser_signal_on (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UAL_laser_signal_off (SyncDomainHandle sdh)

- UA_SPEC UA_RC __stdcall UA_read_io_port (SyncDomainHandle sdh, Digital_Port_ID id, uint32 ∗p_value)
- UA_SPEC UA_RC __stdcall UA_write_io_port (SyncDomainHandle sdh, Digital_Port_ID id, uint32 value)
- UA_SPEC UA_RC __stdcall UA_clear_io_port (SyncDomainHandle sdh, Digital_Port_ID id, uint32 mask)
- UA_SPEC UA_RC __stdcall UA_assert_io_port (SyncDomainHandle sdh, Digital_Port_ID id, uint32 mask)
- UA_SPEC UA_RC __stdcall UA_read_analog_port (SyncDomainHandle sdh, Analog_Port_ID port, float ∗value)
- UA_SPEC UA_RC __stdcall UA_write_analog_port (SyncDomainHandle sdh, Analog_Port_ID port, float value)
- UA_SPEC UA_RC __stdcall UA_load_calibration_file (SyncDomainHandle sdh, char ∗FileName)
- UA_SPEC UA_RC __stdcall UA_load_calibration_fileEx (SyncDomainHandle sdh, WCHAR ∗FileName)
- UA_SPEC UA_RC __stdcall UA_goto_xyz (SyncDomainHandle sdh, Point_3d_t point)
- UA_SPEC UA_RC __stdcall UA_close ()
- UA_SPEC Xform_2d_t __stdcall UA_rotate_xform_2d (float angle)
- UA_SPEC Xform_2d_t __stdcall UA_mul_xform_2d (const Xform_2d_t ∗pA, const Xform_2d_t ∗pB)
- UA_SPEC Xform_2d_t __stdcall UA_inv_xform_2d (const Xform_2d_t ∗pA)
- UA_SPEC Xform_2d_t __stdcall UA_scale_xform_2d (float x, float y)
- UA_SPEC Xform_2d_t __stdcall UA_translate_xform_2d (float x, float y)
- UA_SPEC UA_RC __stdcall UA_rotate_transform_2d (float angle, Xform_2d_t ∗xform)
- UA_SPEC UA_RC __stdcall UA_mul_transform_2d (const Xform_2d_t ∗pA, const Xform_2d_t ∗pB, Xform-_2d_t ∗xform)
- UA_SPEC UA_RC __stdcall UA_inv_transform_2d (const Xform_2d_t ∗pA, Xform_2d_t ∗xform)
- UA_SPEC UA_RC __stdcall UA_scale_transform_2d (float x, float y, Xform_2d_t ∗xform)
- UA_SPEC UA_RC __stdcall UA_translate_transform_2d (float x, float y, Xform_2d_t ∗xform)
- UA_SPEC UA_RC __stdcall UA_load_correction_file (SyncDomainHandle sdh, char const ∗filename, CORR-ECTION_TABLE_ID cortable, double kx, double ky, double phi, double yoffset, double xoffset)
- UA_SPEC UA_RC __stdcall UA_set_offset (SyncDomainHandle sdh, float xoffset, float yoffset)
- UA_SPEC UA_RC __stdcall UA_set_laser_control (SyncDomainHandle sdh, LASER_ENABLE control)
- UA_SPEC UA_RC __stdcall UAL_begin_job (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UAL_end_job (SyncDomainHandle sdh)
- UA_SPEC UA_RC __stdcall UAL_set_pixel_line (SyncDomainHandle sdh, PIXEL_MODE mode, float period, float dx, float dy)

  List command that configures a raster image line.
- UA_SPEC UA_RC __stdcall UAL_set_pixel (SyncDomainHandle sdh, float pulse_width, float laser_power)

  List command that defines the parameters for one pixel of a raster scan.

13.1.1 Detailed Description

This describes the interface to the CTI Universal Adapter, which provides standard connectivity to CTI Controllers.

13.1.2 Enumeration Type Documentation

13.1.2.1 enum Analog_Port_ID

Analog Ports.

Enumerator:

ANALOG_OUT1 Analog Laser Port 1.

ANALOG_OUT2 Analog Laser Port 2.

ANALOG_IN1 Reserved.

ANALOG_IN2 Reserved.

13.1.2.2 enum Digital_Port_ID

I/O ports.

Enumerator:

DIGITAL_INPUTS Digital Input Port.

DIGITAL_INPUTS_EXT Extended Digital Input port.

DIGITAL_OUTPUTS Digital output Port.

DIGITAL_OUTPUTS_EXT Extended digital output port.

DIGITAL_LASER_POWER Digital Laser Power Port.

13.1.2.3 enum LISTNO

List identifiers There can only be two lists in memory

Enumerator:

NULL_LIST Null List.

LIST_1 List 1.

LIST_2 List 2.

13.1.2.4 enum STATUS_LIST

List Status.

Enumerator:

STATUS_INACTIVE The list is inactive.

STATUS_EMPTY The list is empty.

STATUS_OPEN The list is open.

STATUS_LOAD The list is loading.

STATUS_STANDBY The list is in standby.

STATUS_READY The list is ready for execution.

STATUS_IN_JOB_QUEUE The list is in the job queue for execution.

STATUS_EXECUTING The list is being executed.

STATUS_NOT_AVAIL The list is retired.

13.1.2.5 enum STATUS_QUEUE

Enumeration that describes the status of the list Queue.

Enumerator:

QUEUE_EMPTY Queue is empty.
QUEUE_LIST_EXECUTING Queue is executing a list.
QUEUE_HARDWARE_HOLDING Queue is waiting on a hardware hold.

13.1.2.6 enum UA_RC

Return codes from calls to this library.

Enumerator:

UA_OPERATION_OK Function executed as expected.
UA_UNKNOWN_ERROR Something unanticipated happened.
UA_UNKNOWN_EXCEPTION_CAUGHT Something unanticipated happened.
UA_BAD_CALIBRATION_FILE Problem with the calibration file.
UA_NO_HARDWARE_FOUND No hardware found for specified SyncDomain.
UA_PARAM_OUT_OF_LIMITS A parameter fell outside of limits.
UA_UNABLE_TO_SPAWN_THREAD Unable to spawn list execution thread.
UA_CIRCULAR_QUEUE_ENABLED Can't start a list when the circular queue mode is enabled.
UA_UNABLE_TO_WRITE_TEMP_FILE Unable to write temporary files.
UA_BAD_CORRECTION_FILE Correction file is incorrect.
UA_INVALID_LIST_NUMBER An invalid list number was specified.
UA_UNABLE_TO_QUEUE No more memory available to queue more lists.
UA_UNABLE_TO_PROCESS_QUEUE Can't process queue.
UA_NO_LOADABLE_LIST There is no list available to insert this command.
UA_HARDWARE_NOT_READY Device is not ready to connect yet.
UA_HARDWARE_NOT_CONNECTED Client API is not connected to Device.
UA_HARDWARE_ALREADY_EXCLUSIVE Device is connected to another client.
UA_HARDWARE_NOT_CAPABLE API call not applicable to this hardware.
UA_UAPI_NOT_INITIALIZED UA_open() must be called to init dll.
UA_PORT_NOT_VALID Attempting to use a port not available.
UA_NO_DLLS_FOUND If more than one controller is connected to a computer the SyncDomain must be specified. No controller specific UAPI dlls present for CTI_UAPI to use.
UA_BROADCAST_MONITOR_NOT_RUNNING Please start EC1000 BroadcastMonitor tool.
UA_LIST_NOT_CLOSED List is not closed, can not execute.
UA_EMPTY_LIST List is Empty, nothing to execute.
UA_CONNECTION_LOST Connection at been establiches, but was lost.
UA_INVALID_SDH An invalid SyncDomain handle was specified.
UA_FILE_NOT_FOUND File not found.
UA_BAD_LICENSE_FILE License file is invalid.
UA_INVALID_CONNECTION_OPTION For the controller type.

13.1.3 Function Documentation

13.1.3.1 UA_SPEC UA_RC __stdcall UA_assert_io_port (SyncDomainHandle *sdh*, Digital_Port_ID *id*, uint32 *mask*)

Asserts bits for the chosen 32-bit output port subject to a mask value. If the mask bit has a value of 1 the port bit is asserted. If the mask bit has a value of 0 the port bit is unchanged. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports and bits may be available). This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

`in`	*sdh*	Sync domain handle
`in`	*id*	The logical port that will be written to
`in`	*mask*	Mask of bits to assert;

13.1.3.2 UA_SPEC UA_RC __stdcall UA_clear_io_port (SyncDomainHandle *sdh*, Digital_Port_ID *id*, uint32 *mask*)

Unasserts bits for the chosen 32-bit output port subject to a mask value. If the mask bit has a value of 1 the port bit is unasserted. If the mask bit has a value of 0 the port bit is unchanged. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports and bits may be available). This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

`in`	*sdh*	Sync domain handle
`in`	*id*	The logical port that will be written to
`in`	*mask*	Mask of bits to clear;

13.1.3.3 UA_SPEC UA_RC __stdcall UA_close ( )

Causes the CTI Universal API to shut down various threads started by lower level DLL's, ensuring that all commands have been sent to hardware and gracefully releasing library resources. It is necessary to run this before unloading lower level DLLs.

13.1.3.4 UA_SPEC UA_RC __stdcall UA_connect (SyncDomainHandle *sdh*, CONNECT_OPTIONS *options*)

Establishes communications with a device/SyncDomain. The connections supported are hardware dependent. An error will be returned if an unsupported connection type is requested. Supported connection options Non-Exclusive - Multiple processes can access and control the initialized SyncDomain Exclusive - Only a single process can access and control the initialized SyncDomain Platform Default - Use if it necessary for software to connect the different controllers where they all do not support the same connection types.

Parameters

`in`	*sdh*	Specific device reference.
`in`	*options*	Specifies how to connect to the device.

13.1.3.5 UA_SPEC UA_RC __stdcall UA_disconnect (SyncDomainHandle *sdh*)

Disconnects from the SyncDomain. IMPORTANT: Disconnecting from a SyncDomain causes the controller to abort execution. Use the Wait for Execution Complete.vi to assure that execution is complete before disconnecting.

Parameters

in	*sdh*	Specifies SyncDomainHandle

13.1.3.6 UA_SPEC UA_RC __stdcall UA_execute_list (SyncDomainHandle *sdh,* LISTNO *listno,* THREAD *thread*)

Places the last command list to be created onto the execution queue. Calling the Execute List VI multiple times causes the command list to be placed on the queue multiple times. List execution occurs as soon as previously queued non-list commands and executed lists have completed.

Parameters

in	*sdh*	Specifies sync domain Handle
in	*listno*	Specifies the list to execute
in	*thread*	Selects between the library spawning it's own thread for list execution or running on the callers thread. If thread is set to CALLER_THREAD, the function will not return until the entire list has executed. Passing LIBRARY_THREAD causes the library to spawn it's own thread which terminates on list completeion.

13.1.3.7 UA_SPEC UA_RC __stdcall UA_get_controller_info (SyncDomainHandle *sdh,* char ∗ *info,* size_t *info_size*)

Retrieves a null terminated string with extended controller information.

Parameters

in	*sdh*	Specific device reference.
out	*info*	
in	*info_size*	size of memory allocated for info Caller should allocate MAX_INFO_STRIN-G_LEN bytes for the info string. Lines begin with a keyword, followed by "=" followed by a value and terminated with a CR/LF. For example:

FIRMWARE=1.2.3.4 NIOS=3.10C

13.1.3.8 UA_SPEC UA_RC __stdcall UA_get_handle (const char ∗ *name,* SyncDomainHandle ∗ *p_sdh*)

Retrieves a Sync Domain Handle to the friendly name for the Sync Domain. If a NULL name is passed to this function and there is only one scanning head in the system (one Sync Domain) it's handle is returned. If there are multiple Sync-Domains a Name is required (an error is returned if one is not provided when multiple SyncDomains are present). The friendly name is factory set, and may be changed by the user. Refer to the controller documentation for the default, and for how to reset the friendly name.

Parameters

in	*name*	Friendly Name of device. Can use a null string if there is only one head in the system.
out	*p_sdh*	Pointer to SyncDomainHandle

13.1.3.9 UA_SPEC UA_RC __stdcall UA_get_handles (SyncDomainHandle *handles[ ]*, int ∗ *length*)

Retrieves an array of handles to all synchronization domains.

Parameters

out	*handles*	Array of SyncDomainHandle of length MAX_AVAILABLE_SYNC_DOMAI-NS.
out	*length*	Length of array returned. Caller should allocate at least MAX_AVAILABLE_-SYNC_DOMAINS elements in the handles array

13.1.3.10 UA_SPEC UA_RC __stdcall UA_get_name (char ∗ *name*, SyncDomainHandle *sdh*)

Returns the friendly name of the controller associated with the SyncDomain.

```
\param[out] name   Pointer to name of head
```

Parameters

in	*sdh*	SyncDomainHandle

13.1.3.11 UA_SPEC UA_RC __stdcall UA_get_status (SyncDomainHandle *sdh*, UA_status_t ∗ *p_status*)

Retrieves list status.

Parameters

in	*sdh*	Specifies Sync Domain
out	*p_status*	pointer to status structure

13.1.3.12 UA_SPEC UA_RC __stdcall UA_get_sync_domain_info (SyncDomainInfo *sdi[ ]*, const int *len_sdi_array*, int ∗ *p_numValidSyncDomains*)

Retrieves an array of structures describing all synchronization domains.

```
\param[out] sdi[]  an array of SyncDomainInfoStructures
```

Parameters

in	*len_sdi_array*	the number of elements allocated for sdi[]
out	*p_numValidSync-Domains*	Number of valid sync domains. Caller should allocate at least MAX_HEADS elements in the SyncDomainInfo array

13.1.3.13 UA_SPEC UA_RC __stdcall UA_get_version (UA_version_t ∗ *p_version*)

Retrieves version information of the library that implements this API.

Parameters

out	*p_version*	Pointer to a version structure.

13.1.3.14 UA_SPEC UA_RC __stdcall UA_goto_xyz (SyncDomainHandle *sdh*, Point_3d_t *point*)

Places a command on the execution queue to point the mirrors to the specified point at the jump speed. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*point*	Point in workspace coordinates to move to.

13.1.3.15 UA_SPEC UA_RC __stdcall UA_inv_transform_2d (const Xform_2d_t * *pA*, Xform_2d_t * *xform*)

Returns the inverse of a 2-dimensional coordinate transform

Parameters

in	*pA*	pointer to transform A
out	**xform*	Xform_2d_t alocated by user and filled by function

13.1.3.16 UA_SPEC Xform_2d_t __stdcall UA_inv_xform_2d (const Xform_2d_t * *pA*)

DEPRECATED: Use instead UA_inv_transform_2d(), which provides a UA_RC return value. Returns the inverse of a 2-dimensional coordinate transform

Parameters

in	*pA*	pointer to transform A

13.1.3.17 UA_SPEC UA_RC __stdcall UA_laser_signal_off (SyncDomainHandle *sdh*)

Turns the laser off. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including executed lists. Not all controllers support this command. Refer to the controller documentation.

Parameters

in	*sdh*	Sync domain handle

13.1.3.18 UA_SPEC UA_RC __stdcall UA_laser_signal_on (SyncDomainHandle *sdh*)

Adds a Laser On command to the current list. When executed it turns the laser off. Not all controllers support this command. Refer to the controller documentation

Parameters

in	*sdh*	Sync domain handle

13.1.3.19 UA_SPEC UA_RC __stdcall UA_load_calibration_file (SyncDomainHandle *sdh*, char * *FileName*)

Loads a calibration file from the file system. The file name is given using ASCII (8-bit) characters.

Parameters

in	*sdh*	Sync domain handle
in	*FileName*	Name of calibration file in file system

13.1.3.20 UA_SPEC UA_RC __stdcall UA_load_calibration_fileEx (SyncDomainHandle *sdh*, WCHAR * *FileName*)

Loads a calibration file from the file system. The file name is given using wide (16-bit) characters.

Parameters

in	*sdh*	Sync domain handle
in	*FileName*	Name of calibration file in file system

13.1.3.21 UA_SPEC UA_RC __stdcall UA_load_correction_file (SyncDomainHandle *sdh*, char const * *filename*, CORRECTION_TABLE_ID *cortable*, double *kx*, double *ky*, double *phi*, double *yoffset*, double *xoffset*)

Loads a CTB correction file from the file system.

Parameters

in	*sdh*	Sync domain handle
in	*filename*	Name of calibration file in file system
in	*cortable*	Specifies which table to load into
in	*kx*	Scaling factor for X axis
in	*ky*	Scaling factor for Y axis
in	*phi*	Rotation angle in degrees
in	*xoffset*	Additional X offset
in	*yoffset*	Additional Y offset

13.1.3.22 UA_SPEC UA_RC __stdcall UA_mul_transform_2d (const Xform_2d_t * *pA*, const Xform_2d_t * *pB*, Xform_2d_t * *xform*)

Returns the product of two 2-dimensional coordinate transforms multiplied in the order AB, meaning that point will be transformed as A(B(x,y)).

Parameters

in	*pA*	pointer to transform A
in	*pB*	pointer to transform B
out	*∗xform*	Xform_2d_t alocated by user and filled by function

13.1.3.23 UA_SPEC Xform_2d_t __stdcall UA_mul_xform_2d (const Xform_2d_t * *pA*, const Xform_2d_t * *pB*)

DEPRECATED: Use instead UA_mul_transform_2d(), which provides a UA_RC return value. Returns the product of two 2-dimensional coordinate transforms multiplied in the order AB, meaning that point will be transformed as

A(B(x,y)).

Parameters

in	*pA*	pointer to transform A
in	*pB*	pointer to transform B

13.1.3.24 UA_SPEC UA_RC __stdcall UA_read_analog_port (SyncDomainHandle *sdh*, Analog_Port_ID *port*, float ∗ *value*)

Reads the value of the chosen input port. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports may be available). This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*port*	The logical port that will be written to
out	*value*	Port value mapped to a range of +/-100.0

13.1.3.25 UA_SPEC UA_RC __stdcall UA_read_io_port (SyncDomainHandle *sdh*, Digital_Port_ID *id*, uint32 ∗ *p_value*)

Reads data from the chosen 32-bit input port. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports and bits may be available). This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*id*	The logical port that will be written to
out	*p_value*	value of port;

13.1.3.26 UA_SPEC UA_RC __stdcall UA_rotate_transform_2d (float *angle*, Xform_2d_t ∗ *xform*)

Returns 2-dimensional transform which can be used to rotate coordinates by the input angle.

Parameters

in	*angle*	Angle of rotation in radians. Units are in radians.
out	*xform*	Xform_2d_t alocated by user and filled by function

13.1.3.27 UA_SPEC Xform_2d_t __stdcall UA_rotate_xform_2d (float *angle*)

DEPRECATED: Use instead UA_rotate_transform_2d(), which provides a UA_RC return value. Returns 2-dimensional transform which can be used to rotate coordinates by the input angle. This function is obselete

Parameters

in	*angle*	Angle of rotation in radians. Units are in radians.

13.1.3.28 UA_SPEC UA_RC __stdcall UA_scale_transform_2d (float *x*, float *y*, Xform_2d_t * *xform*)

Returns 2-dimensional transform which can be used to scale coordinates in X and Y by the input values.

Parameters

in	*x*	X multiplier
in	*y*	Y multiplier
out	*xform*	Xform_2d_t alocated by user and filled by function

13.1.3.29 UA_SPEC Xform_2d_t __stdcall UA_scale_xform_2d (float *x*, float *y*)

DEPRECATED: Use instead UA_scale_transform_2d(), which provides a UA_RC return value. Returns 2-dimensional transform which can be used to scale coordinates in X and Y by the input values.

Parameters

in	*x*	X multiplier
in	*y*	Y multiplier

13.1.3.30 UA_SPEC UA_RC __stdcall UA_set_FPS_timing (SyncDomainHandle *sdh*, float *width*, float *advance*)

Negative advance indicates pulse is issued after laser modulation begins.

Configures First Pulse Suppression (FPS) timing. Refer to the controller documentation for the physical location of the FPS signal. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including executed lists.

Parameters

in	*sdh*	Sync domain handle
in	*width*	Width of pulse.
in	*advance*	Time the FPS pulse leads the start of the laser modulation signal in seconds. A negative advance indicates pulse is issued after laser modulation begins.

13.1.3.31 UA_SPEC UA_RC __stdcall UA_set_jump_speed (SyncDomainHandle *sdh*, float *speed*)

Sets the slew rate for jumpss in User Units per Second, with user units being set by the scaling parameters of the Workspace and the Object Transforms. Note that because CTI controllers currently only support one speed setting that applies to both the X and Y axes the speed is set to the average when the X scaling and the Y scaling are different. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*speed*	Slew rate of scanner in workspace units per second

13.1.3.32 UA_SPEC UA_RC __stdcall UA_set_laser_control (SyncDomainHandle *sdh*, LASER_ENABLE *control*)

Enables or disables the ability of other Universal API commands (i.e. Mark and Shoot) to turn on the primary laser. Disabling laser controlallows for the system to be run without affecting the workpiece. Typically, this is used to allow positioning of a workpiece using the trace laser (The trace laser is not driven by the Mark and the Shoot commands). This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*control*	constants for enable/disable

13.1.3.33 UA_SPEC UA_RC __stdcall UA_set_laser_delays (SyncDomainHandle *sdh*, float *laserOn*, float *laserOff*)

Sets laser delays to account for the times required to accelerate and to decelerate the galvos for marking. These delays are applied at the beginning (On Delay) and ending (Off Delay) of a series of mark commands. Negative delays may be used to turn on the laser before the galvos start to move or to turn off the laser before the galvos reach their final position. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*laserOn*	Delay in turning on the laser at the beginning of a series of mark commands in seconds.
in	*laserOff*	Delay in turning off the laser at the end of a series of mark commands in seconds.

13.1.3.34 UA_SPEC UA_RC __stdcall UA_set_laser_power (SyncDomainHandle *sdh*, float *power*)

Configures laser power for marking via a laser power port. Refer to the controller manual for the ports that are driven by this command. This may drive both a digital port and an analog port. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands and lists.

Parameters

in	*sdh*	Sync domain handle
in	*power*	Percent of full scale power

13.1.3.35 UA_SPEC UA_RC __stdcall UA_set_laser_timing (SyncDomainHandle *sdh*, float *Period*, float *PulseWidth1*, float *PulseWidth2*)

Configures digital laser modulation for marking. Both modulation signals are of the same period, one having pulses of Pulse Width 1, the other pulses of Pulse Width 2. Refer to the controller documentation for the physical location of the laser modulation signals. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including executed lists.

Parameters

in	*sdh*	Sync domain handle
in	*Period*	Period of laser pulse in seconds
in	*PulseWidth1*	Length of laser1 pulse in seconds
in	*PulseWidth2*	Length of laser2 pulse in seconds

13.1.3.36 UA_SPEC UA_RC __stdcall UA_set_laser_timing_standby (SyncDomainHandle *sdh,* float *Period,* float *PulseWidth1,* float *PulseWidth2*)

Configures digital laser modulation for jumpimg (when the laser is off or at a non-marking power). Both modulation signals are of the same period, one having pulses of Pulse Width 1, the other pulses of Pulse Width 2. Refer to the controller documentation for the physical location of the laser modulation signals. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including executed lists.

Parameters

in	*sdh*	Sync domain handle
in	*Period*	Period of laser pulses in seconds
in	*PulseWidth1*	Length of laser pulse 1 in seconds
in	*PulseWidth2*	Lenght of laser pulse 2 in seconds

13.1.3.37 UA_SPEC UA_RC __stdcall UA_set_mark_speed (SyncDomainHandle *sdh,* float *speed*)

Sets the slew rate for marks in User Units per Second, with user units being set by the scaling parameters of the Workspace and the Object Transforms. Note that because CTI controllers currently only support one speed setting that applies to both the X and Y axes the speed is set to the average when the X scaling and the Y scaling are different. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*speed*	Slew rate of scanner in workspace units per second

13.1.3.38 UA_SPEC UA_RC __stdcall UA_set_offset (SyncDomainHandle *sdh,* float *xoffset,* float *yoffset*)

Sets offset for field.

Parameters

in	*sdh*	Sync domain handle
in	*xoffset*	offset for X axis
in	*yoffset*	offset for Y axis

13.1.3.39 UA_SPEC UA_RC __stdcall UA_set_scanner_delays (SyncDomainHandle *sdh,* float *Jump,* float *Mark,* float *Polygon*)

Sets the delays between jumps and marks (Jump Delay), marks and marks (Polyline Delay), and marks and jumps (Mark Delay)to account for the galvo lagging behind its commanded position. The Polyline Delay value will be adjusted by library functions to account for the angle between a mark and a subsequent mark [Actual Delay = Nominal Delay $*$ (1-cos(angle))]. This behavior prevents rounded corners while minimizing marking times. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

in	*sdh*	Sync domain handle
in	*Jump*	Delay added at the end of a jump in seconds.
in	*Mark*	Delay added at the end of a series of connected vectors in seconds
in	*Polygon*	Delay added between connected vectors in seconds

13.1.3.40 UA_SPEC UA_RC __stdcall UA_set_workspace_transform_2d (SyncDomainHandle *sdh*, const Xform_2d_t * *p_ws_xform*)

Sets the Workspace Transform in memory that is used to map the scan field to the units, rotation, and offset required by the user. The intrinsic scan field runs from -0.5 to +0.5, has its origin at the center of the field, and has axes that correspond to the axes of the scan head. Scaling can be used to convert to user units from the intrinsic field. If the Field of View (along the x and y axes) is measured in user units, the scaling factor is the inverse of that measured value. This VI is polymorphic, and will accept the transform as a matrix or as a set of parameters (scaling, offset, and rotation). If giving a set of parameters rotation is applied first, then scaling, and then translation (offset). This function is normally called at the beginning of a program as part of system calibration, before any lists are executed.

Parameters

in	*sdh*	Specifies sync domain to apply transform to
in	*p_ws_xform*	Pointer to transform

13.1.3.41 UA_SPEC UA_RC __stdcall UA_stop_execution (SyncDomainHandle *sdh*)

Stops processing of the execution queue and empties the queue. This includes any pending non-list commands as well as any pending list commands, but it is most useful for stopping execution of a list. This does not affect a list that has not yet been sent for execution, which will remain available for execution. Starting a new list will clear the existing list if a recovery from stopping execution is necessary.

Parameters

in	*sdh*	Handle of syncdomain to stop

13.1.3.42 UA_SPEC UA_RC __stdcall UA_translate_transform_2d (float *x*, float *y*, Xform_2d_t * *xform*)

Returns 2-dimensional transform which can be used to translate coordinates in X and Y by the input values.

Parameters

in	*x*	X value
in	*y*	Y value
out	**xform*	Xform_2d_t alocated by user and filled by function

13.1.3.43 UA_SPEC Xform_2d_t __stdcall UA_translate_xform_2d (float *x*, float *y*)

DEPRECATED: Use instead UA_translate_transform_2d(), which provides a UA_RC return value. Returns 2-dimensional transform which can be used to translate coordinates in X and Y by the input values.

Parameters

`in`	*x*	X value
`in`	*y*	Y value

13.1.3.44 UA_SPEC UA_RC __stdcall UA_write_analog_port (SyncDomainHandle *sdh*, Analog_Port_ID *port*, float *value*)

Writes the value of the chosen port. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports may be available). This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

`in`	*sdh*	Sync domain handle
`in`	*port*	The logical port that will be written to
`in`	*value*	Value has a range of +/-100.0 and is mapped to the analog range supported by the hardware

13.1.3.45 UA_SPEC UA_RC __stdcall UA_write_io_port (SyncDomainHandle *sdh*, Digital_Port_ID *id*, uint32 *value*)

Writes data to the chosen 32-bit output port. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports and bits may be available). This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including queued lists.

Parameters

`in`	*sdh*	Sync domain handle
`in`	*id*	The logical port that will be written to
`in`	*value*	value of port;

13.1.3.46 UA_SPEC UA_RC __stdcall UAL_assert_io_port (SyncDomainHandle *sdh*, Digital_Port_ID *id*, uint32 *mask*)

Adds an Assert Digital Port command to the current list. When executed it asserts bits for the chosen 32-bit output port subject to a mask value. If the mask bit has a value of 1 the port bit is assserted. If the mask bit has a value of 0 the port bit is unchanged. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports and bits may be available).

Parameters

`in`	*sdh*	Specifies SyncDomain Handle
`in`	*id*	The logical port that will be written to
`in`	*mask*	A 32 bit mask of bits to assert.

13.1.3.47 UA_SPEC UA_RC __stdcall UAL_begin_job (SyncDomainHandle *sdh*)

Adds a Begin Job command to the current list. When executed it asserts a user accessible digital output that can be used for process control. For the EC1000 this drives the Busy light on the front panel as well. Not all controllers support this command. Refer to the controller documentation for the pin driven by this command.

Parameters

`in`	*sdh*	Sync domain handle

13.1.3.48 UA_SPEC UA_RC __stdcall UAL_clear_io_port (SyncDomainHandle *sdh*, Digital_Port_ID *id*, uint32 *bitmask*)

Adds a Clear Digital Port command to the current list. When executed it unasserts bits for the chosen 32-bit output port subject to a mask value. If the mask bit has a value of 1 the port bit is unasserted. If the mask bit has a value of 0 the port bit is unchanged. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports and bits may be available).

Parameters

`in`	*sdh*	Specifies SyncDomain Handle
`in`	*id*	Port The logical port that will be written to
`in`	*bitmask*	A 32 bit mask of bits to clear.

13.1.3.49 UA_SPEC UA_RC __stdcall UAL_end_job (SyncDomainHandle *sdh*)

Adds an End Job command to the current list. When executed it unasserts a user accessible digital output that can be used for process control. For the EC1000 this drives the Busy light on the front panel as well. Not all controllers support this command. Refer to the controller documentation for the pin driven by this command.

Parameters

`in`	*sdh*	Sync domain handle

13.1.3.50 UA_SPEC UA_RC __stdcall UAL_jump (SyncDomainHandle *sdh*, Point_3d_t *point*)

Adds a Jump command to the current list. When executed it moves to the specified workspace coordinate point with the laser off.

Parameters

`in`	*sdh*	Specifies SyncDomain Handle
`in`	*point*	Workspace coordinate to move to in terms of user units

13.1.3.51 UA_SPEC UA_RC __stdcall UAL_laser_signal_off (SyncDomainHandle *sdh*)

Adds a Laser Off command to the current list. When executed it turns the laser off. Not all controllers support this command. Refer to the controller documentation.

Parameters

in	*sdh*	Sync domain handle

13.1.3.52 UA_SPEC UA_RC __stdcall UAL_laser_signal_on (SyncDomainHandle *sdh*)

Turns the laser on. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including executed lists. Not all controllers support this command. Refer to the controller documentation.

Parameters

in	*sdh*	Sync domain handle

13.1.3.53 UA_SPEC UA_RC __stdcall UAL_mark (SyncDomainHandle *sdh*, Point_3d_t *point*)

Adds a Mark command to the current list. When executed it moves to the specified workspace coordinate point with the laser on.

Parameters

in	*sdh*	Specifies SyncDomain Handle
in	*point*	Workspace coordinate to move to in terms of user units

13.1.3.54 UA_SPEC UA_RC __stdcall UAL_set_end_of_list (SyncDomainHandle *sdh*)

Ends a command list. To send the list to the execution queue call UA_execute_list().

Parameters

in	*sdh*	Specifies SyncDomain Handle

13.1.3.55 UA_SPEC UA_RC __stdcall UAL_set_FPS_timing (SyncDomainHandle *sdh*, float *width*, float *advance*)

Negative advance indicates pulse is issued after laser modulation begins.

Adds a Set FPS Timing command to the current list. When executed it will configure First Pulse Suppression (FPS) timing. Refer to the controller documentation for the physical location of the FPS signal.

Parameters

in	*sdh*	Sync domain handle
in	*width*	Width of pulse.
in	*advance*	Time the FPS pulse leads the start of the laser modulation signal in seconds. A negative advance indicates pulse is issued after laser modulation begins.

13.1.3.56 UA_SPEC UA_RC __stdcall UAL_set_laser_power (SyncDomainHandle *sdh*, float *power*)

Adds a Set Laser Power command to the current list. When executed it will set the primary laser power for marking via a laser power port. Refer to the controller manual for the ports that are driven by this command. This may drive both a digital port and an analog port.

Parameters

`in`	*sdh*	Sync domain handle
`in`	*power*	Percent of full scale power

13.1.3.57 UA_SPEC UA_RC __stdcall UAL_set_laser_timing (SyncDomainHandle *sdh,* float *Period,* float *PulseWidth1,* float *PulseWidth2*)

Adds a Set Laser Timing command to the current list. When executed it will configure laser modulation for marking. Both modulation signals are of the same period, one having pulses of Pulse Width 1, the other pulses of Pulse Width 2. Refer to the controller documentation for the physical location of the laser modulation signals. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including executed lists.

Parameters

`in`	*sdh*	Sync domain handle
`in`	*Period*	Period of laser pulse in seconds
`in`	*PulseWidth1*	Length of laser1 pulse in seconds
`in`	*PulseWidth2*	Length of laser2 pulse in seconds

13.1.3.58 UA_SPEC UA_RC __stdcall UAL_set_laser_timing_standby (SyncDomainHandle *sdh,* float *Period,* float *PulseWidth1,* float *PulseWidth2*)

Adds a Set Laser Timing command to the current list. When executed it will configure laser modulation for jumping (when the laser is off or at a non-marking power). Both modulation signals are of the same period, one having pulses of Pulse Width 1, the other pulses of Pulse Width 2. Refer to the controller documentation for the physical location of the laser modulation signals. This command is placed on the end of the execution queue immediately, and will execute after previously queued commands including executed lists.

Parameters

`in`	*sdh*	Sync domain handle
`in`	*Period*	Period of laser pulses in seconds
`in`	*PulseWidth1*	Length of laser pulse 1 in seconds
`in`	*PulseWidth2*	Lenght of laser pulse 2 in seconds

13.1.3.59 UA_SPEC UA_RC __stdcall UAL_set_object_transform_2d (SyncDomainHandle *sdh,* const Xform_2d_t * *p_obj_xform*)

Adds a Set Object Transform command to the current list. Once set, this will apply to all commands that follow within the list either until the end of the list or until it is set again. The Object Transform will not carry forward beyond the end of the list.

Parameters

`in`	*sdh*	Specifies sync domain
`in`	*p_obj_xform*	Pointer to a transform

13.1.3.60 UA_SPEC UA_RC __stdcall UAL_set_pixel (SyncDomainHandle *sdh*, float *pulse_width*, float *laser_power*)

List command that defines the parameters for one pixel of a raster scan.

Parameters

in	*sdh*	Sync domain handle
in	*pulse_with*	Width of pulse in seconds
in	*laser_power*	Laser power in percent of full power Each image line must start with a UAL-_set_pixel_line. Each pixel in the line is defined by UAL_set_pixel. The first subsquent command that is not UAL_set_pixel turns off the pixel output mode

13.1.3.61 UA_SPEC UA_RC __stdcall UAL_set_pixel_line (SyncDomainHandle *sdh*, PIXEL_MODE *mode*, float *period*, float *dx*, float *dy*)

List command that configures a raster image line.

Parameters

in	*sdh*	Sync domain handle
in	*mode*	Selection between step and shoot and shoot while slewing
in	*period*	Pixel output period in seconds
in	*dx*	X distance between adjacent pixels in workspace units
in	*dy*	Y distance between adjacent pixels in workspace units Each image line must start with the command UAL_set_pixel_line. Individual pixels are defined by U-A_set_pixel functions. The first subsquent command that is not UAL_set_pixel turns off the pixel output mode

13.1.3.62 UA_SPEC UA_RC __stdcall UAL_set_start_list (SyncDomainHandle *sdh*, LISTNO *listno*)

Starts a command list. Any list commands called after starting a list will be added to the list. Any non-list commands are still placed on the execution queue immediately. To send the list to the execution queue call UAL_set_end_of-_list() followed by UA_execute_list(). Calling UAL_set_start_list() before executing a list will cause the list to be overwritten.

Parameters

in	*sdh*	Specifies SyncDomain Handle
in	*listno*	List number select from enum LISTNO.

13.1.3.63 UA_SPEC UA_RC __stdcall UAL_shoot (SyncDomainHandle *sdh*, float *time*)

Adds a Shoot command to the current list. When executed it turns the laser on for the specified time with the galvos stationary. While the laser is on futher commands to the Sync Domain are blocked.

Parameters

in	*sdh*	Specifies SyncDomain Handle
in	*time*	Time in seconds to fire laser.

13.1.3.64 UA_SPEC UA_RC __stdcall UAL_trigger_probes (SyncDomainHandle *sdh*)

Adds a Trigger Probes command to the current list. When executed it triggers probes internal to the controller. Not all controllers support this command. Refer to the controller documentation for which probes, if any, are available, and for which software is needed to recover the data.

Parameters

`in`	*sdh*	Specifies sync domain

13.1.3.65 UA_SPEC UA_RC __stdcall UAL_wait (SyncDomainHandle *sdh*, float *time*)

Adds a Wait command to the current list. When executed it blocks list execution for the specified time with the laser turned off.

Parameters

`in`	*sdh*	Specifies SyncDomain Handle
`in`	*time*	Time in seconds to wait.

13.1.3.66 UA_SPEC UA_RC __stdcall UAL_wait_sync (SyncDomainHandle *sdh*)

Adds a Wait Sync command to the current list. When executed it blocks execution until an external signal triggers execution to resume. Not all controllers support this command. Refer to the controller documentation for which controllers have a Sync input.

Parameters

`in`	*sdh*	Specifies SyncDomain Handle

13.1.3.67 UA_SPEC UA_RC __stdcall UAL_write_analog_port (SyncDomainHandle *sdh*, Analog_Port_ID *port*, float *value*)

Adds a Write Analog Port command to the current list. When executed it writes the value of the chosen port. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports may be available).

Parameters

`in`	*sdh*	Specifies SyncDomain Handle
`in`	*port*	The logical port that will be written to
`in`	*value*	Value has a range of +/-100.0 and is mapped to the analog range supported by the hardware

13.1.3.68 UA_SPEC UA_RC __stdcall UAL_write_io_port (SyncDomainHandle *sdh*, Digital_Port_ID *id*, uint32 *value*)

Adds a Write Digital Port command to the current list. When executed it writes data to the chosen 32-bit output port. The ports are logical ports. Refer to the documentation for the particular controller for which logical ports and bits are available on a physical connector (Note that not all logical ports and bits may be available).

Parameters

`in`	*sdh*	Specifies SyncDomain Handle
`in`	*id*	The logical port that will be written to
`in`	*value*	A 32 bit value to write