Interface documentation linLLT

C/C++
Linux

I.   About this document

The purpose of this document is to enable the reader to integrate a scanCONTROL laser profile sensor into dedicated Linux applications via C/C++. This document is based upon the application interface provided by the linLLT libraries.

To get a general overview, a general introduction to the linLLT library and the measurement principles are given at the beginning. Then the resulting measurement values are specified. This is necessary to get a basic understanding of the measurement data used in the Software. Further, the different measurement and profile data formats are explained and the data transmission types are illustrated. The general part finishes with an illustration of the limitations regarding measurement speed.

For detailed introduction into programming with the scanCONTROL SDK basic programming tasks are illustrated via example code. Detailed programming examples and a full documentation of the API support the actual implementation.

II.   Versions

Versions	Date	Author	State
0.1	10.09.2015	DRa	Initial draft
1.0	27.10.2015	UEi, DRa	Reviewed Version 1
1.1	08.11.2015	DRa	Extensions
1.2	06.11.2017	Dra	Updated for v0.2.0
1.3a	28.02.2019	Uei	Initial Integration sC30xx
1.3	14.06.2019	Uei	Integration of sC30xx completed
1.4	21.10.2019	Uei	Integration of sC25xx
1.5a	28.05.2020	Uei	Initial integration sC30xx-200
1.6	21.04.2021	THl	Update

III.   Content

1   Introduction

1.1   Measurement principle and data

1.1.1   Principle of optical triangulation

Like conventional laser distance sensors, scanCONTROL laser profile sensors make use of the principle of optical triangulation. The beam of a laser diode is widened by special optics and projected onto a measurement target. The receiver optic focuses the diffuse reflected light, which is then detected by a CMOS sensor matrix. To ensure that only the reflection of the projected laser line is detected, a bandpass filter is embedded right before the sensor matrix. This filter allows only light to pass, which correlates to the wavelength of the laser diode.
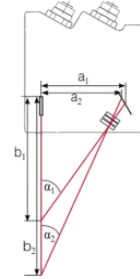
Based on the position of the detected laser beam within one column of the sensor matrix, the distance of one measuring point to a defined reference in the sensor (z axis) can be calculated via triangulation. Usually the bottom of the sensor is chosen as reference. The basic calculation relies on the following formula:

$$b_1 = \frac{a_1}{\tan \alpha_1}$$

The resolution in z direction is determined by the number of pixels in the z axis of the sensor matrix. As reflections are detected by more than one pixel, the center of gravity of the reflection is used to calculate the position (subpixel resolution).

According to the position of a measuring point within one row of the matrix, a distance value can be assigned to an x value (i.e. position). The number of pixels in a sensor row determines how many single measurement points are available.

The resulting measuring data is a two dimensional profile, which is calibrated to a dimensional unit ([mm]) by the sensor. This allows for either a relative or an absolute measurement. 3D measurements can be done via movement of the sensor or the target along the y axis. If a steady movement can be accomplished or if an encoder is used, a data grid with equidistantly distributed points can be generated.

1.1.2   Available measuring values

In addition to the distance and position values (Z / X) scanCONTROL sensors generate and send further information about the current measurement. This includes the intensity, reflection width, moment 0 and moment 1. Additionally the threshold used for every single point is transmitted. These values are described below:
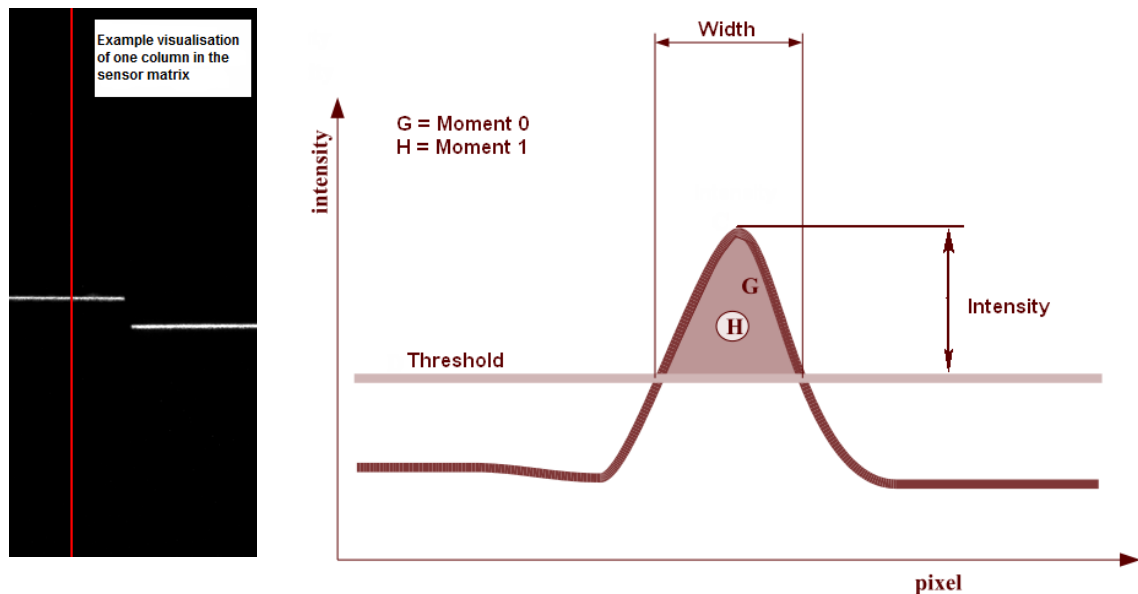
Fig. 1: How to get a measuring value from a reflection (evaluation of one sensor column)

- <u>Distance</u>: To get the distance (i.e. z value) of a measuring point, the center of gravity of the reflection detected by the CMOS sensor column is calculated. Based on a calibration table this value is converted to a real distance coordinate in the sensor. The value is transmitted as 16 bit unsigned integer field which has to be scaled by the sensor specific scaling factors.

- <u>Position</u>: The position (x value) corresponds to a pixel row of the CMOS sensor. For every column one position value is detected. Calibration to the real position is achieved by the calibration table saved on the sensor. A 16 bit unsigned integer field is transmitted which has to be scaled, too.

- <u>Intensity</u>: The transmitted value is the difference between the detected intensity maximum and the currently used threshold. Intensity correlates to how much light one pixel of the matrix has detected while the shutter was open. Prerequisite for detection of a reflection is that the intensity is above the threshold. A 10 bit unsigned integer field is transmitted.

- <u>Reflection width</u>: The reflection width correlates to the number of contiguous pixels the intensity of the current reflection is above the threshold. A 10 bit unsigned integer field is transmitted.

- <u>Moment 0</u>: Corresponds to the integral intensity ("area of reflection") of the current reflection. The moment is thus defined by the integral of the intensity over the reflection width; see Fig. 1 (G). The value is transmitted as 32 bit unsigned integer field.

- <u>Moment 1</u>: Corresponds to the center of gravity of the reflection, which is used as foundation for calculation of positon and distance according to the calibration table. It is transmitted as 32 bit unsigned integer.

- <u>Threshold</u>: The Threshold used for the single measuring point, which consists of the absolute or dynamically calculated threshold and the determined backlight suppression. A 10 bit unsigned integer field is transmitted.

All of these values are in respect to the current reflection detection setting of the sensor. You can choose to detect the first or last reflection over the threshold (detected for each sensor column), the reflection with the maximum intensity or the reflection with the biggest integral intensity ("largest area") (Fig. 2).
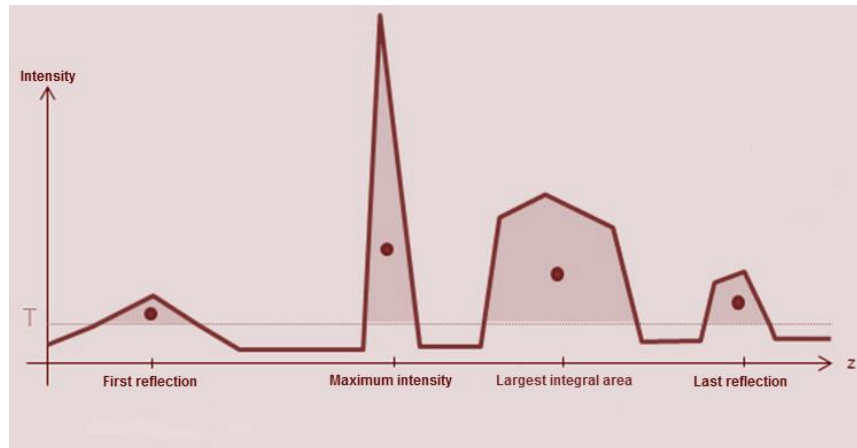
Fig. 2: Detected reflections

1.2 linLLT libraries

1.2.1 Overview

The linLLT interfaces consists of two *shared objects*, the main C library libmescan and the wrapper library libLLT. A shared object is a library, which is loaded automatically while starting an application it is linked to. It has to be available at compile time, but can be exchanged/updated without recompiling a project, as long as the ABI is consistent.

Both available libraries are based on the Open Source (LGPL) interface library aravis [7]. Aravis makes it possible to transmit a video image data over Ethernet, as well as the evaluation of the GeniCam XML. Tested and recommended version is 0.6.1 (01/2019). libmescan offers C-functions to search, initialize and control scanners, as well as for measurement data transmission and data conversion. This functionality is all needed to fully operate a scanner and enables the user to build own flexible C applications. Calling convention is *cdecl*. libLLT wraps libmescan and aravis in a own C++ API and is thus a comfortable way to use both libraries for building own applications. The API is similar to the ME Windows SDK (LLT.dll) to allow for seamless transition.

1.2.2 Compile

To compile an application with linLLT, further libraries are necessary on the computer. Specifically these are aravis-0.6 and the libraries aravis is depending on (e.g. libxml2, glib, ...). The provided examples come with a meson build file, which shows how to use the meson build system for compiling. Following linker flags are necessary to compile an application with linLLT:

```
-lpthread -lglib-2.0 –lxml2 –lgthread-2.0 –lgio-2.0 –lgobject-2.0 –lm –lllt
                                               –laravis-0.6  -lmescan
```

2   Operating Modes for profile generation (only scanCONTROL 30xx)

The scanCONTROL 30xx series provides three Operating Modes for the profile generation. Depending on the needs of the measurement task one of these can be selected.

2.1   High Resolution mode

The High Resolution mode is the standard mode for profile acquisition. The High Resolution mode provides profile data with the best Z linearity compared to the other modes.

2.2   High Speed mode

In the High Speed mode profiles can be acquired at double profile frequency (in comparison to the High Resolution mode at the same measuring field size) while losing a small degree of Z linearity.

2.3   High Dynamic Range mode (HDR)

The High Dynamic Range mode is used for target surfaces that have inhomogenous surface properties (e.g. very dark and very bright parts). Therefore a special sensor matrix feature is used to optimize the profile data of difficult surfaces while not being prone of in-motion unsharpness.

Like in the High Speed mode, the Z linearity is slightly worse than in the High Resolution mode.

3   Measurement data format

3.1   Video Mode

The Video Mode is used for transmitting the 8 bit greyscale bitmaps detected by the CMOS sensor matrix (Fig. 3). Only the actual image data is transmitted – header and mirroring of the image has to be realized externally if necessary. This data format does not have a timestamp. The measurement frequency must not exceed 25 Hz. It has to be considered that scanner of the 29xx series require a Gigabit Ethernet connection to transmit the image data with the minimal internal frequency of 25 Hz. The reason is the big image size, which leads to a necessary bandwidth of 262 Mbps (25 Hz * 1024 * 1280 * 8 bit).

Fig. 3: Example Video Mode

Scanners of the 30x0 series provide a high resolution video mode that requires a bandwidth of 445 Mbps (25 Hz * 1088 * 2048 * 8 bit) while the high resolution video mode of the scanCONTROL 30x2 series requires 112 Mbps (25 Hz * 544 * 1024 * 8 bit). Both series provide a

low resolution video mode that operates at 28 Mbps (25 Hz * 272 * 512 * 8 bit) for 100 Mbit Ethernet connections.

3.2   Single profile transmission

3.2.1   General format of profile data

By default a 64 byte wide data field is transmitted for each measuring point in single profile transmission mode. The height of the data field is determined by the set number of points per profile. The 64 bytes are divided into four stripes (peaks) with 16 bytes each. Every stripe can consist of one complete profile, but usually only the first stripe contains valid profile data. If multiple reflections are detected, the other stripes are filled as well. The last 16 bytes of each transmission contains the timestamp. That means in *Full Set* data transmission (default) the last point of the fourth stripe is overwritten.

For every point in each stripe, all information described in the previous chapter is transmitted. It is structured as following:

0..7	8..15	16..23	24..31
Res. (2 bit)	Reflection width (10 bit)	Max. Intensity (10 bit)	Threshold (10 bit)
Position (16 bit)		Distance (16 bit)	
Moment 0 (32 bit)			
Moment 1 (32 bit)			

Single measurement values are encoded in the big endian byte format. As mentioned previously position and distance data (x/z) has to be scaled with measuring range specific scaling factors. A basic illustration of the data arrangement is shown in Fig. 4.
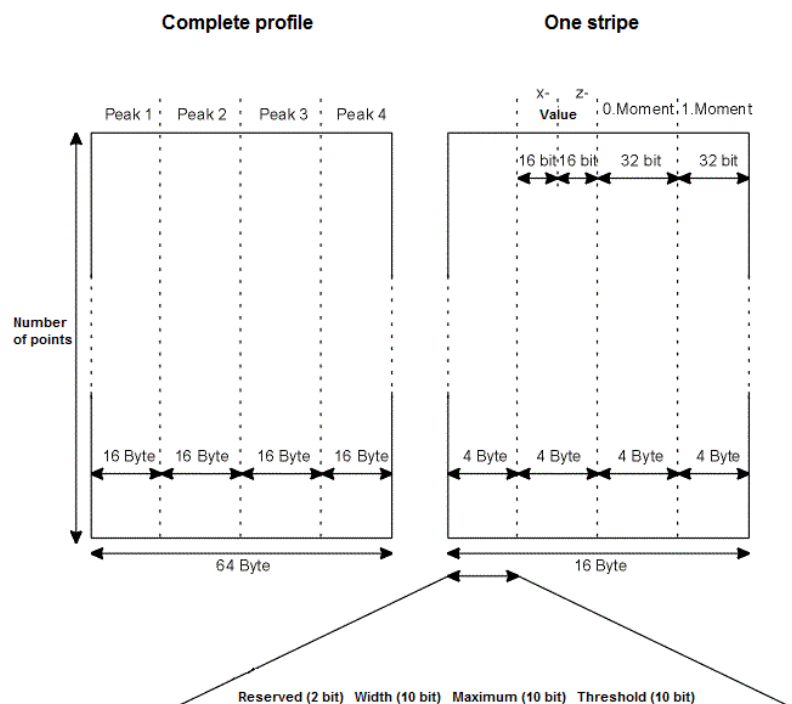
Fig. 4: Data arrangement single profile transmission

3.2.2 Timestamp information

The timestamp transmitted in the last 16 bytes includes the following key data of a measured profile (data format: unsigned integer; big endian):

- Profile counter: Incremental counter for identification of profiles; is increased by one after each measurement. The field consists of 24 bit and is thus able to count until 16777215 profiles. After that the counter is set back to zero.

- Shutter open: Contains the absolute time at which the exposure was started. The internal clock has a period of 128 seconds. The 32 bit wide field consists of a second counter, a cycle counter and a cycle offset. From these values the moment of shutter opening can be calculated.

- Edge counter: Depending on the scanner settings two times the detected encoder edges or the state of the digital inputs is transmitted. The field is 16 bit wide.

- Shutter close: Contains the absolute time at which the exposure was stopped. It is equivalent to the *shutter open* field.

The timestamp is structured like this:

0..7		8..15	16..23	24..31
Flags (2 bit)	Reserved (6 bit)		Profile counter (24 bit)	
Shutter open (32 bit)				
Edge counter or DigIn (16 bit)			Reserved (16 bit)	
Shutter close (32 bit)				

The parts of the timestamp for shutter open/closed are compounded as following:

Seconds counter (7 bit)	Cycle counter (13 bit)	Cycle offset (12 bit)

The absolute time can be calculated via:

$$Timestamp = seconds\ counter + \frac{cycle\ counter}{8000} + \frac{cycle\ offset}{8000 * 3072}$$

(Remark: The cycle count is overflowing at 8000 and the cycle offset at 3072!)

3.2.3 CMM timestamp

If the *Coordinate Measuring Machine* (CMM) trigger is activated the format of the timestamp changes. Instead of the 16 bit encoder edge counter / the reserved field, following CMM specific information is transmitted:

CMM edge counter (16 bit)	CMM trigger flag (1 bit)	CMM active flag (1 bit)	CMM trigger impulse count (14 bit)

3.2.4 Complete measurement data set (Full Set, PROFILE)

By default all key data values described previously are transmitted. The predefined profile configuration format *Full Set* extracts all of this data from the transmission buffer. The amount of

data in this configuration is represented by 64 bytes for every point of the profile. In context of the DLL this configuration is called *PROFILE*. The output encoding is big endian.

3.2.5   One stripe (QUARTER_PROFILE)

The profile configuration *QUARTER_PROFILE* extracts one stripe of the *Full Set* data. Consequently the amount of data to be evaluated is decreased. The timestamp information is attached to the measurement data, which means the amount of data is 16 byte per profile point plus 16 byte timestamp. Is has to be mentioned that the amount of transmitted data is not reduced, instead only the specified part (the selected stripe) of the data buffer is evaluated and passed to the application. Thus the data is still transmitted completely. This configuration is also encoded in big endian.

3.2.6   X/Z data (PURE_PROFILE)

With the profile configuration *PURE_PROFILE* only position and distance values are extracted from the currently set stripe. The behavior is same as in *QUARTER_PROFILE* configuration, which means the transmitted amount of data is not reduced. The data to be evaluated is reduced to 4 bytes per profile point plus 16 byte timestamp. The values are extracted in little endian encoding.

3.2.7   Partial profile (PARTIAL_PROFILE)

A partial profile (*PARTIAL_PROFILE*) is a custom cropped profile generated directly on the scanner. It is therefore possible to reduce the amount of transmitted data significantly. Also on-scanner data processing is accelerated, which is important while operating with high data rates. The size of the profile can be defined by the following four parameters:

1. *StartPoint*:   First measuring point included in the profile

2. *StartPointData*:   Data offset, from which byte the data of a point should be included in the profile

3. *PointCount*:   Number of measuring points included in the profile counted from the *StartPoint*

4. *PointDataWidth*:   Number of bytes from the *StartPointData* offset which should be included in the profile
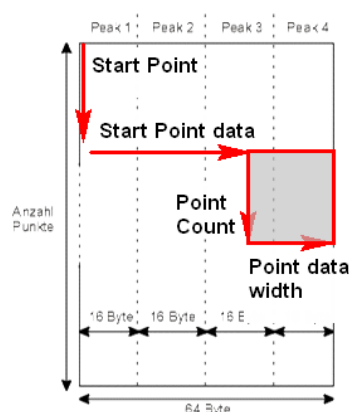
Fig. 5: Illustration Partial Profile

The reference point for setting the *StartPoint* or the *StartPointData* is the upper left edge. Counting starts with the index 0. The timestamp overwrites the last 16 bytes of the partial profile. The transmitted amount of data is reduced to (PointDataWidth x PointCount) bytes. This configuration is encoded in big endian.

3.3 Container Mode

The container mode allows combining data from several profiles into one big transmission container. The advantages of this mode include reduction of the necessary reaction intervals of the software application and of data overhead.

3.3.1 Standard Container Mode

In Standard Container Mode profiles are combined into one logical transmission container. The sensor collects data until the requested amount is reached and transmits it as one package. The maximum container size depends on the sensor (128 Mbyte / 4096 profiles). In this case the main advantage is the longer reaction time intervals in the software.

3.3.2 Rearranged Container Mode (Transposed Container Mode)

The *Rearrangement* feature enables the sensor to send only the really necessary data. The user can freely define from which stripe which measuring values should be transmitted. These values are saved continuously per profile. The timestamp can be saved in an additional field. The chosen values are collected for the set amount of profiles and then transmitted.

One rearrangement configuration often used is the so called transposed container mode. Here only the distance data is transmitted (optional: position as well). This data is then arranged as a 16-bit grey scale bitmap, which can be analyzed with standard vision tools. The width of the bitmap is determined by the number of points per profile; the height by the number of profiles in the container. The default data format is big endian, but can be changed to little endian.
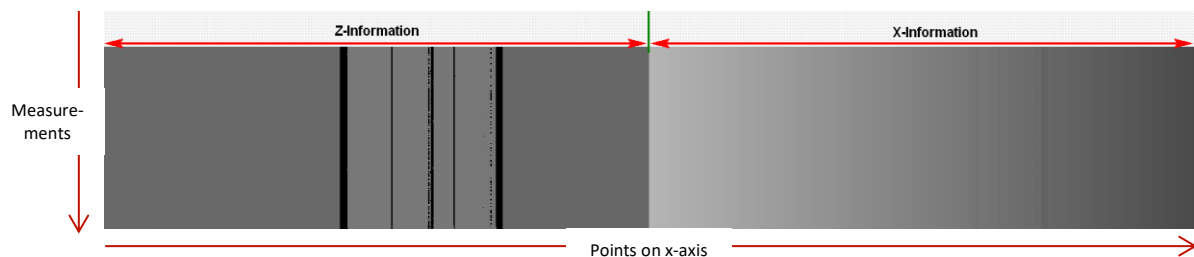
Fig. 6: Example image transposed container mode

4 Data transmission scanCONTROL sensor

4.1 Data transmission

Data transmission is started and stopped by the user application. If the transmission is active, received data is written into the receiving buffer according to the profile frequency. It can be chosen between continuous data extraction from the profile buffer (NORMAL_TRANSFER) and shot wise data extraction from the profile buffer (transmission of a defined number of profiles = SHOT_TRANSFER).

4.2 Polling of measurement data

For non-time critical application or applications which do not need all profiles or video images to be evaluated, it is possible to extract the data actively from the receiving buffer. For every poll the last profile/container/image received is copied from the buffer to an additional buffer reserved by the application, which then can be used for further evaluation. If there is no new profile since the last poll, the application is noticed.

4.3   Using callbacks

The second way of fetching data is by using a callback: For every received profile or container the callback function is executed. If the callback is finished an event is to be set. One parameter of the callback is a pointer to the currently received profile/container. The data size depends on the profile configuration set. The pointer enables the callback to copy the received data into an evaluation buffer. The callback function has to be very fast to make sure the next buffer can be fetched by the driver.

5   Measuring speed

The maximum measuring speed depends on several sensor parameters and is different for every sensor type. Additionally the given network infrastructure can be a limiting factor. If the maximum speed is exceeded data is lost and/or corrupted and thus should be completely avoided.

To avoid performance problems resulting from the network environment a Gigabit Ethernet connection should be established between PC and Sensor. Especially if sensors of the 29xx series or the 30xx series are used, 100 Mbps networks are quickly at their bandwidth limitations. Furthermore a PC with a sufficient hardware performance should be used, as a huge amount of data has to be analyzed – especially if there is more than one sensor connected.

Most of the time, the sensor speed is limited by the measuring field used by the sensor. The measuring field corresponds to the part of sensor matrix which is read out by the scanCONTROL sensor. The smaller the read out area, the faster is the maximal measuring frequency. A detailed listing of possible frequencies for each measuring field is given in the scanCONTROL *Quick Reference*, which is part of the sensor documentation. In general, only the necessary part of the matrix should be evaluated, especially during high frequency measurements.

An important parameter to consider in regard of measuring speed is the exposure time. The maximum frequency of a measuring task can be determined by calculation the reciprocal value of the exposure time. If there are advanced post processing operations configured on the sensor (SMART or GAP sensors), which can be a bottle neck as well. The duration of the current post processing can be seen in the *scanCONTROL Configuration Tools.* If the sensors are operated with higher frequencies, the basic processing can limit the speed, even without having configured post processing tasks. This can be countered by only reading out the area of the matrix which contains the points of interest and reducing the transmitted data using the partial profile configuration (e.g. only one stripe or x/z data). If the maximum sensor speed is used, it is especially important to use a configuration for lowest possible data transmission. This has no influence on the quality of measurements, because all additional points are not in the measuring field and would just contain invalid values.

6   Typical code examples with references to the SDK

The following chapter shows basic code examples for different integration steps. Complete projects with e.g. error handling can be found in the project folder of the SDK. Except for example 6.1 the sensor has to be connected prior of code execution. In some examples, registers are set (*SetFeature(…)*) – note that register addresses and function values are partly mapped to macros in the SDK (e.g. `FEATURE_FUNCTION_EXPOSURE_TIME` for exposure time). Detailed description of the registers can be found in Operation Manual Part B, which is part of the scanner documentation (see chapter 8.4).

6.1   Connect to the sensor

This example shows how to find a sensor and how to connect to it.

```cpp
std::vector<char *> Interfaces(5);

// Create a device handle
CInterfaceLLT pLLT = new CInterfaceLLT();  // (C) LLT *pLLT = create_llt_device();

// Search for scanners on interface
CInterfaceLLT::GetDeviceInterfaces(&Interfaces[0], Interfaces.size());

// Set Path to device_properties.dat (optional)
pLLT->SetPathtoDeviceProperties("./device_properties.dat");

// Set device interface to handle
pLLT->SetDeviceInterface(Interfaces[0]);

// Connect sensor
pLLT->Connect();
```

See API: GetDeviceInterfaces(), SetPathToDeviceProperties(), SetDeviceInterface(), Connect()

6.2   Set profile frequency and exposure time (only scanCONTROL 30xx)

This example shows how to change the profile frequency and the exposure time (shutter time) for scanCONTROL 30xx. The set value marks the time in 1 µs steps. The profile frequency cannot be set directly but is composed from the exposure time (*ExposureTime*) and the idle time (*IdleTime*). Calculation:

$$Profile\ frequency = \frac{1}{(ExposureTime + IdleTime)}$$

```cpp
unsigned int ExposureTime     = 1005;
unsigned int IdleTime         = 8995;

// Set shutter time
pLLT->SetFeature(FEATURE_FUNCTION_EXPOSURE_TIME,
    (((ExposureTime % 10) << 12) & 0xF000) +
    ((ExposureTime / 10) & 0xFFF));

// Set idle time
pLLT->SetFeature(FEATURE_FUNCTION_IDLE_TIME,
    (((IdleTime % 10) << 12) & 0xF000) +
    ((IdleTime / 10) & 0xFFF));
```

See API: SetFeature(), FEATURE_FUNCTION_EXPOSURE_TIME, FEATURE_FUNCTION_IDLE_TIME
See Operation Manual Part B: *OpManPartB.html#exposuretime, OpManPartB.html#idletime*

The example code sets the exposure time to 1.005 ms and the profile frequency to 100 Hz.

6.3   Set profile frequency and exposure time (all scanCONTROL types)

This example shows how to change the profile frequency and the exposure time (shutter time). The set value marks the time in 10 µs steps. The profile frequency cannot be set directly but is composed from the exposure time (*ExposureTime*) and the idle time (*IdleTime*). Calculation:

$$Profile\ frequency = \frac{1}{(ExposureTime + IdleTime) * 10\ \mu s}$$

```cpp
unsigned int ExposureTime    = 100;
unsigned int IdleTime        = 900;


// Set shutter time to 1 ms (100*10 us)
pLLT->SetFeature(FEATURE_FUNCTION_EXPOSURE_TIME, ExposureTime);

// Set idle time to 9 ms (900*10 us)
pLLT->SetFeature(FEATURE_FUNCTION_IDLE_TIME, IdleTime);
```

See API: SetFeature(), FEATURE_FUNCTION_EXPOSURE_TIME, FEATURE_FUNCTION_IDLE_TIME
See Operation Manual Part B: *OpManPartB.html#exposuretime, OpManPartB.html#idletime*

The example code sets the exposure time to 1 ms and the profile frequency to 100 Hz.

6.4   Poll measurement data

This example shows how to fetch data from the sensor via active polling. To poll data the function *GetActualProfile()* copies the last received profile from the receiving buffer to an application buffer for further evaluation. After that the function *ConvertProfile2Values()* extracts the x/z data from the buffer. This function automatically calculates the position and distance values in mm according to the scaling factors.

```cpp
unsigned int Resolution = Resolutions[0];
TScannerType LLTType;
unsigned int LostProfiles = 0;

// Set buffer for complete profile and x/z values
std::vector<unsigned char>ProfileBuffer(Resolution * 64);
std::vector<double>ValueX(Resolution);
std::vector<double>ValueZ(Resolution);

// Query scanner type
pLLT->GetLLTType(&LLTType);

// Poll a profile from receiving buffer
// Hint: If no new profile has been received since the last call, the function
// returns -104. This may be used to constantly query for new data in a loop.
pLLT->GetActualProfile(&ProfileBuffer[0], (unsigned int)ProfileBuffer.size(),
                                          PROFILE, &LostProfiles));

// Convert buffer values into x/z data
CInterfaceLLT::ConvertProfile2Values(&ProfileBuffer[0], ProfileBuffer.size(),
    Resolution, PROFILE, LLTType, 0, NULL, NULL, NULL, &ValueX[0], &ValueZ[0],
    NULL, NULL);
```

See API: GetLLTType(), TransferProfiles(), GetActualProfile(), ConvertProfiles2Values()

6.5   Get measurement data via callback

This example shows how to fetch profile data via callback. For this purpose a callback is registered, which is executed for every received profile. The callback function copies the data from the receiving buffer and emits an event after one profile. After the profile is received the transmission is stopped.

```cpp
unsigned int Resolution;
unsigned int ProfileBufferSize;
TScannerType scanCONTROLType;
void NewProfile (const void *data, size_t data_size, gpointer ptr);
// Event handle
EHANDLE *event;

[…] // Init, Connect und read out of Resolution and TScannerType

// Allocate Buffers
std::vector<double> ValueX(Resolution);
std::vector<double> ValueZ (Resolution);
std::vector<unsigned char>ProfileBuffer(Resolution * 64);

event = CInterfaceLLT::CreateEvent();

// Register Callback function
pLLT->RegisterBufferCallback((gpointer)&NewProfile, NULL)

// Start profile transmission
pLLT->TransferProfiles(NORMAL_TRANSFER, true);

CInterfaceLLT::ResetEvent(event);

// Wait for Event
if (CInterfaceLLT::WaitForSingleObject(event, 5000) != WAIT_OBJECT_0)
{
    cout << "Timeout!" << endl;
}

// Stop profile transmission
pLLT->TransferProfiles(NORMAL_TRANSFER, false);

// Convert raw data to mm
CInterfaceLLT::ConvertProfile2Values(&ProfileBuffer[0], ProfileBuffer.size(),
    Resolution, PROFILE, LLTType, 0, NULL, NULL, NULL, &ValueX[0], &ValueZ[0],
    NULL, NULL);

CInterfaceLLT::FreeEvent(event);

// Callback funktion (copies received data to buffer and sets an event)
void NewProfile (const void *pucData, size_t uiSize, gpointer user_data)
{
    if (uiSize == ProfileBuffer.size())
    {
        memcpy(&ProfileBuffer[0], pucData, uiSize);
    }
    CInterfaceLLT::SetEvent(event);
}
```

See API: RegisterCallback(), TransferProfiles(), ConvertProfiles2Values(), Event handling

6.6   Set profile filter

This example shows how to set resampling, median and average filter.

```cpp
// Set average filter to 7 taps
unsigned int ProfileFilter = FILTER_AVG_7;
// Set median filter to 5 taps
ProfileFilter |= FILTER_MEDIAN_5;
// Set resampling (inter/extrapolation of invalid points; alle Info; huge)
ProfileFilter |= FILTER_RESAMPLE_EXTRAPOLATE_POINTS |
                 FILTER_RESAMPLE_ALL_INFO | FILTER_RESAMPLE_HUGE;


// Set configured filters
pLLT->SetFeature(FEATURE_FUNCTION_PROFILE_FILTER, ProfileFilter);
```

See API: SetFeature(), FEATURE_FUNCTION_PROFILE_FILTER
See Operation Manual Part B: *OpManPartB.html#profilefilter*

6.7   Encoder

This example shows how to activate encoder triggering via digital inputs.

```cpp
unsigned int Encoder = 0;

// Set trigger input to encoder
unsigned int Trigger = TRIG_MODE_ENCODER;
// Set digital inputs as trigger input and activate ext. triggering
Trigger |= TRIG_INPUT_DIGIN | TRIG_EXT_ACTIVE;
// Set trigger settings
pLLT->SetFeature(FEATURE_FUNCTION_TRIGGER, Trigger);

// Set multi-function port to bidirectional 24V encoder mode
unsigned int MultiPort = MULTI_DIGIN_ENC_INDEX | MULTI_LEVEL_24V
                                               | MULTI_ENCODER_BIDIRECT;
pLLT->SetFeature(FEATURE_FUNCTION_DIGITAL_IO, MultiPort);

// Read maintenance register and activate encoder
pLLT->GetFeature(FEATURE_FUNCTION_MAINTENANCE, &Encoder);
Encoder |= MAINTENANCE_ENCODER_ACTIVE;
pLLT->SetFeature(FEATURE_FUNCTION_MAINTENANCE, Encoder);
```

See API: SetFeature(), FEATURE_FUNCTION_TRIGGER, FEATURE_FUNCTION_MAINTENANCE,
FEATURE_FUNCTION_DIGITAL_IO
See Operation Manual Part B: *OpManPartB.html#trigger, OpManPartB.html#maintenance, OpManPartB.html#ioconfig*

6.8   External triggering

This example shows how to activate external triggering via digital inputs.

```cpp
// Set trigger input to pos. pulse mode
unsigned int Trigger = TRIG_MODE_PULSE | TRIG_POLARITY_HIGH;
// Set digital input as trigger input and activate ext. triggering
Trigger |= TRIG_INPUT_DIGIN | TRIG_EXT_ACTIVE;
// Set trigger settings
pLLT->SetFeature(FEATURE_FUNCTION_TRIGGER, Trigger);

// Set multi-function port to 5V digital input triggering
unsigned int MultiPort = MULTI_DIGIN_TRIG_ONLY | MULTI_LEVEL_5V;
pLLT->SetFeature(FEATURE_FUNCTION_DIGITAL_IO, MultiPort);
```

See API: SetFeature(), FEATURE_FUNCTION_TRIGGER, FEATURE_FUNCTION_DIGITAL_IO
See Operation Manual Part B: *OpManPartB.html#trigger, OpManPartB.html#ioconfig*

6.9   Software profile trigger

This example shows how to trigger a profile via software profile trigger.

```
// Activate ext. triggering
pLLT->SetFeature(FEATURE_FUNCTION_TRIGGER, TRIG_EXT_ACTIVE);

// Software triggering; triggers one profile
pLLT->TriggerProfile();
```

See API: SetFeature(), FEATURE_FUNCTION_TRIGGER
See Operation Manual Part B: *OpManPartB.html#trigger*

6.10   Software container trigger

This example shows how to trigger a container via software container trigger. To use this function, the sensor either has to be in frametrigger mode (see Digital IO configuration) or the functionality has to be activated separately by TriggerContainerEnable() / TriggerContainerDisable().

```
// Activate container trigger if necessary
pLLT->TriggerContainerEnable();

// Start container transfer
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, true);

// Trigger one container
pLLT->TriggerContainer();

[…] // Wait for container and fetch it from buffer

// Stop container transfer
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, false);

// Deactivate container trigger if necessary
pLLT->TriggerContainerDisable();
```

See API: SetFeature(), TriggerContainer(), FEATURE_FUNCTION_TRIGGER
See Operation Manual Part B: *OpManPartB.html#trigger*

Note: Firmware version v46 or newer required

6.11   Set peak filter

This example shows how to set the peak filters of the scanner. These filters exclude points which are not in a certain range of intensity and/or reflection width from the profile. To activate a 0 must be written to FEATURE_FUNCTION_EXTRA_PARAMETER.

```
// Set desired peak filter values
unsigned short min_width      = 2;   // values <2 increase noise significantly
unsigned short max_width      = 1023;
unsigned short min_intensity  = 0;
unsigned short max_intensity  = 1023;

pLLT->SetPeakFilter(min_width, max_width, min_intensity, max_intensity);

// Write 0 to EXTRA_PARAMETER Register
pLLT->SetFeature(FEATURE_FUNCTION_EXTRA_PARAMETER, 0);
```

See API: SetPeakFilter, SetFeature(), FEATURE_FUNCTION_EXTRA_PARAMETER
See Operation Manual Part B: *OpManPartB.html#extraparameter*

6.12   Compute sensor matrix regions

To set a free measuring field (Firmware versions < v43, see chapter 6.13), a region of interest (see chapter 6.14), a region of no interest or a reference region for the automatic exposure time on the sensor matrix, the percentage on the correctly rotated sensor matrix (cf. scanCONTROL Configuration Tools) have to be converted to the correct matrix values. The formulas to be used depend on the used sensor type.

```
// Percentage of measuring field
double start_z = 25.0;
double end_z = 75.0;
double start_x = 20.0;
double end_x = 80.0;

// scanCONTROL 30xx
unsigned short col_start = start_x / 100 * 65535;
unsigned short col_size = (end_x – start_x) / 100 * 65535;
unsigned short row_start = start_z / 100 * 65535;
unsigned short row_size = (end_z – start_z) / 100 * 65535;

// scanCONTROL 29xx, 27xx, 25xx
unsigned short col_start = 65535 - (end_x / 100 * 65535);
unsigned short col_size = (end_x – start_x) / 100 * 65535;
unsigned short row_start = 65535 – (end_z / 100 * 65535);
unsigned short row_size = (end_z – start_z) / 100 * 65535;

// scanCONTROL 26xx
unsigned short col_start = 65535 - (end_x / 100 * 65535);
unsigned short col_size = (end_x – start_x) / 100 * 65535;
unsigned short row_start = start_z / 100 * 65535;
unsigned short row_size = (end_z – start_z) / 100 * 65535;
```

6.13   Set free measuring field

This example shows how to set the free measuring field. This setting makes it possible to define a custom size for the measuring field. It can be set via sequential register. To activate a 0 must be written to FEATURE_FUNCTION_EXTRA_PARAMETER.

```
// Activate free measuring field
pLLT->SetFeature(FEATURE_FUNCTION_ROI1_PRESET, MEASFIELD_ACTIVATE_FREE);

// Set free measuring field
pLLT->SetFreeMeasuringField(col_start, col_size, row_start, row_size);

// Write 0 to EXTRA_PARAMETER Register
pLLT->SetFeature(FEATURE_FUNCTION_EXTRA_PARAMETER, 0);
```

See API: <u>SetFeature()</u>, <u>FEATURE_FUNCTION_ROI1_PRESET</u>, <u>SetFreeMeasuringField()</u>, <u>FEATURE_FUNCTION_EXTRA_PARAMETER</u>
See Operation Manual Part B: *OpManPartB.html#roi1, OpManPartB.html#extraparameter*

See chapter 6.12 how to compute the correct matrix region values.

6.14 Set region of interest 1 (measuring field)

This example shows how to set the region of interest 1 (free measuring field 1) on the sensor matrix. This setting makes it possible to define a custom size for the measuring field.

```cpp
// Activate region of interest 1
pLLT->SetFeature(FEATURE_FUNCTION_ROI1_PRESET, MEASFIELD_ACTIVATE_FREE);

// write values to sensor
pLLT->SetFeature(FEATURE_FUNCTION_ROI1_DISTANCE, (row_start << 16) + row_size);
pLLT->SetFeature(FEATURE_FUNCTION_ROI1_POSITION, (col_start << 16) + col_size);

// Write 0 to Extraparameter register to activate the ROI setting
pLLT->SetFeature(FEATURE_FUNCTION_EXTRA_PARAMETER, 0);
```

See API: <u>SetFeature()</u>, <u>FEATURE_FUNCTION_ROI1_PRESET</u>, <u>FEATURE_FUNCTION_EXTRA_PARAMETER</u>
See Operation Manual Part B: *OpManPartB.html#roi1, OpManPartB.html#extraparameter*

See chapter 6.12 how to compute the correct matrix region values.

Note for scanCONTROL 27xx and scanCONTROL 26xx/29xx (Firmware version < v43):
Set free measuring field has to be used instead, see 6.13.

6.15 Calibrate sensor position

This example shows how to calibrate the sensor position. This is useful if a horizontal mounting position cannot be achieved, but the profile has to be straight. It is possible to calibrate the x/z offset and angle.

```cpp
// Rotational centrum and angle
double center_x = -9; // mm
double center_z = 86.7; // mm
double angle = -45; // °

// Shift rotational centrum
double shift_x = 0; // mm
double shift_z = 0; // mm

// Set calibration
pLLT->SetCustomCalibration(center_x, center_z, angle, shift_x, shift_z);

// Reset calibration
pLLT->ResetCustomCalibration();
```

See API: <u>SetCustomCalibration()</u>, <u>ResetCustomCalibration()</u>

6.16 Container Mode for evaluation with vision tools

This example shows how to configure the data transmission for further evaluation with standard vision tools. The resulting data format enables the vision tool to work directly with the data.

```c
#include <math.h>

[…]

unsigned int ProfileCount = 500;// Number of profiles in one image/container

// Calculate flag for current resolution
unsigned int Res = (unsigned int)floor((log((double)Resolution)*1.0/log(2.0))+0.5);
unsigned int Rearrangement = CONTAINER_DATA_Z | CONTAINER_STRIPE_1 |
                                        CONTAINER_DATA_LSBF | Res << 12);

 // Set rearrangement parameters for z data extraction
pLLT->SetFeature(FEATURE_FUNCTION_PROFILE_REARRANGEMENT, Rearrangement);


// Set container size
pLLT->SetProfileContainerSize(Resolution, uiProfileCount);

// Allocate buffer (z value has 2 bytes)
ProfilerBuffer.resize(Resolution * 2 * ProfileCount);

// Start profile transmission
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, true);

[…] // Callback

// Stop profile transmission
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, false);

// Extract/convert complete raw data to mm
CInterfaceLLT::ConvertRearrangedContainer2Values(&ContainerBuf[0],
  ContainerBuf.size(), Rearrangement, ProfileCount, LLTType, 0, NULL, NULL, NULL,
                                              &ValueX[0], &ValueZ[0]);
```

See API: SetFeature(), SetProfileContainerSize(), TransferProfiles(), FEATURE_FUNCTION_PROFILE REARRANGEMENT, ConvertRearrangedContainer2Values()

See Operational Manual Part B: *OpManPartB.html#profilerearrangement*

6.17   Transmission of partial profiles

This example shows how to set up transmission of partial profiles. The transmitted profile corresponds to the profile configuration PURE_PROFILE with a reduced number of points. This means only the x/z data of a defined range of points is transmitted.

```
// Struct defining the partial profile
TPartialProfile PartialProfile;

[…] // Init

// Set partial profile size
PartialProfile.nStartPoint = 20; // Offset 20 -> start point = point 21
PartialProfile.nStartPointData = 4; // Data offset 4 bytes -> start x data
PartialProfile.nPointCount = m_uiResolution / 2; // Half the resolution
PartialProfile.nPointDataWidth = 4; // 4 bytes -> x and z (2 bytes each)

// Allocate profile buffer
ProfileBuffer.resize(PartialProfile.nPointCount * PartialProfile.nPointDataWidth);

// Set partial profile
pLLT->SetPartialProfile(&PartialProfile);

[…] // Callback

// Convert buffer content to real measured values
CInterfaceLLT::ConvertPartProfile2Values(&ProfileBuffer[0], ProfileBuffer.size(),
&PartialProfile, LLTType, 0, NULL, NULL, NULL, &ValueX[0], &ValueZ[0], NULL, NULL);
```

See API: SetPartialProfile(), TransferProfiles(), ConvertPartProfile2Values()

Eventual change of the profile resolution has to be done before the partial profile configuration is sent to the sensor because calling *setResolution()* resets the partial profile setting.

6.18 Use several sensors in one application

This example shows how to use two sensors in one application.

```
// Create handle for each sensor
CInterfaceLLT pLLT  = new CInterfaceLLT();
CInterfaceLLT pLLT2 = new CInterfaceLLT();

// Search interfaces
CInterfaceLLT::GetDeviceInterfaces(Interfaces, Interfaces.GetLength(0));

// Set interfaces
pLLT->SetDeviceInterface(Interfaces[0]);
pLLT2->SetDeviceInterface(Interfaces[1]);

// Connect to sensors
pLLT->Connect();
pLLT2->Connect();

[…]

int userData1 = 1;
int userData2 = 2;

// Register callback sensor 1
pLLT->RegisterBufferCallback((gpointer)&NewProfile, &userData1);

// Register callback sensor 1
pLLT2->RegisterBufferCallback((gpointer)&NewProfile, &userData2);
```

```
[…] // Start of transmission equivalent

// Callback with differentiation of sensor data
void NewProfile(const void *pucData, size_t uiSize, gpointer userData)
{
    if (*(int *)userData == 1
    {
        // Data sensor 1
    }

    if (*(int *)userData == 2)
    {
        // Data sensor 2
    }
}
```

See API: CreateLLTDevice(), GetDeviceInterfaces(), SetDeviceInterface(), Connect(), RegisterBufferCallback()

6.19   Error message if sensor connection is lost

This example shows how to register a callback for error message handling. This is especially important in case of a lost sensor connection. In this context a Message Box is generated in a Windows Forms application.

```
// Register Callback
pLLT->RegisterControlLostCallback((gpointer)&ControlLostCallback, NULL)

void ControlLostCallback(ArvGvDevice *mydevice)
{
    if (mydevice)
    {
        /* Example behavior after lost connection */
        cout << "Control lost!" << endl;
        exit(0);
    }
}
```

See API: RegisterControlLostCallback()

6.20   Read temperature

This example shows how to read the core temperature of the sensor. The value describes the temperature in 0.1 K steps.

```
unsigned int Temperature = 0;

// Before reading temperature TEMP_PREPARE_VALUE has to be written to register
pLLT->SetFeature(FEATURE_FUNCTION_TEMPERATURE, TEMP_PREPARE_VALUE);

// Read temperature
pLLT->GetFeature(FEATURE_FUNCTION_TEMPERATURE, &Temperature);
```

See API: SetFeature(), FEATURE_FUNCTION_TEMPERATURE
Siehe Operational Manual Part B: *OpManPartB.html#temperature*

6.21   Calculate and set packet delay

This example shows how to determine the minimal and maximal packet delay times for a sensor. This is necessary for network infrastructures with more than one scanner connected to a switch. The packet delay depends on the parameters packet size, network bandwidth, the amount of

data to be transmitted and the number of sensors. The amount of data can be calculated from the profile configuration used and the profile frequency set. The minimal packet delay can be calculated as following:

$$PD_{min} = (Number\ of\ sensors - 1) * \frac{Packet\ size}{Network\ bandwidth}$$

The maximum delay is calculated like this:

$$PD_{max} = \left(1000 * \frac{\frac{1000}{Profile\ frequency}}{\frac{KByte\ per\ profile * 1024}{Packet\ size} + 1} - \frac{Packet\ size}{Network\ bandwidth}\right) * 0{,}8$$

The value to be configured has to be within these borders. For every connected sensor the same packet delay must be set. The scanners then are trying to find a transmission slot on their own. Once a sensor finds a free slot, it uses it permanently for sending packets.

To set the value, the following code has to be executed (here a value of 50 µs is set):

```cpp
unsigned int PacketDelay = 50;

// Set packet delays in us
pLLT->SetFeature(FEATURE_FUNCTION_PACKET_DELAY, PacketDelay);
```

See API: SetFeature(), FEATURE_FUNCTION_PACKET_DELAY

7   API

This chapter lists the complete API (Application Program Interface). Every function is illustrated with its parameters and return values. This chapter focuses on the C++ API (CInterfaceLLT class). For a pure C implementation the corresponding C functions follow the naming scheme: llt_handle->CppInterfaceFunc([args]); → c_api_func(llt_handle, [args]); e.g. hllt->SetFeature(([args]); → set_feature(hllt, ([args]);. In C the llt_handle can be created with *create_llt_device()* (see comment in code chapter 6.1) and deleted with *del_device()*.

7.1   Interface and initialization functions

- **GetDeviceInterfaces ()**

```cpp
static int
CInterfaceLLT::GetDeviceInterfaces(char *interfaces[], unsigned int size);
```

Query the available scanCONTROL device interfaces at the PCs interface cards. The returned device interfaces is an Ethernet IP address.

Parameter
    interfaces   Array for available interfaces (unique identifiers)
    size   Size of array

Return value
    Number of device interfaces found

ERROR_GETDEVINTERFACES_REQUEST _COUNT	-251	The size of the passed field is to small
ERROR_GETDEVINTERFACES _INTERNAL	-253	A error occurred during the scanCONTROL enumeration

- **InitDevice ()**

```
static int
CInterfaceLLT::InitDevice(const char *camName, MEDeviceData *devData,
                                              const char *pathDevProp);
```

Initialization of scanner - is called automatically while executing Connect() .

Parameter
       camName   Identificator of sensor
       devData   Device data of sensor
       pathDevProp Path to device_properties.dat

Return value
       General error codes
       Specific return values*:*

ERROR_DEVPROP_NOT_FOUND	-1300	device_properties.dat (file) not found
ERROR_DEVPROP_DECODE	-1301	Decoding of file failed
ERROR_DEVPROP_DEPRECATED	-1302	Dataversion of file obsolete
ERROR_DEVPROP_READ_FAILURE	-1303	Reading the file failed

- **SetPathtoDeviceProperties ()**

```
int
CInterfaceLLT::SetPathtoDeviceProperties(const char *pathDevProp);
```

Set path to the device_properties.dat. Optionally; must be called before Connect. If set more information about the sensors is available in MEDeviceData.

Parameter
       pathDevProp   Path to device_properties.dat

Return value
       General error codes
       Specific return values*:*

ERROR_DEV_PROP_NOT_FOUND	-1300	device_properties.dat (file) not found

- **SetDeviceInterface ()**

```
int
CInterfaceLLT::SetDeviceInterface(unsigned int interface);
```

Assign a scanCONTROL device interface to a sensor instance in the library. The interface is described by a unique identifier. This identifier consists of the device name and the serial number used.

Parameter
>   *interface*   Interface of scanCONTROL to be connected

Return value
>   *General return values*
>   *Specific return value*:

`ERROR_GETDEVINTERFACES _CONNECTED`	-252	The scanCONTROL is connected, call *Disconnect()*

7.2 Connection functions

- **Connect ()**

```
int
CInterfaceLLT::Connect();
```

Connect to the scanCONTROL sensor assigned to the device handle. Only possible if a valid device interface is assigned (via *SetDeviceInterface()*).

Return value
>   *General return values*
>   *Return values of InitDevice()*
>   *Specific return values*:

`ERROR_CONNECT_SELECTED_LLT`	-301	The selected interface is not available -> choose a new interface with *SetDeviceInterface()*
`ERROR_CONNECT_ALREADY_CONNECTED`	-302	There is already a scanCONTROL connected with this ID
`ERROR_DEVPROP_NOT_AVAILABLE`	-999	device_properties.dat not found

- **Disconnect ()**

```
int
CInterfaceLLT::Disconnect();
```

Disconnect from scanCONTROL sensor. All set parameters are preserved on the sensor, except the driver parameters *Packetsize*, *Buffer count* and *Profile config*.

Return value
>   *General return values*

7.3 Identification functions

- **GetDeviceName ()**

```
int
CInterfaceLLT::GetDeviceName(const char **devName, const char **venName);
```

Query device name and vendor name of scanCONTROL sensor.

Parameter
>   *devName*   Pointer device name
>   *venName*   Pointer vendor name

Return value
>   *General return values*

- **GetLLTVersion ()**

```
int
CInterfaceLLT::GetLLTVersion(const char **devVersion);
```

Query the device Firmware version.

Parameter
>   *devVersion*   Pointer Firmware

Return value
>   *General return values*

- **GetLLTType ()**

```
int
CInterfaceLLT::GetLLTType(TScannerType *scannerType);
```

Query measuring range and type of scanCONTROL sensor.

Parameter
>   *scannerType*   Scanner type

Return value
>   *General return values*

- **GetLLTTypeByName ()**

```
static int
CInterfaceLLT::GetLLTTypeByName (const char *modelName,
                                              TScannerType *scannerType);
```

Query measuring and scanner type of LLT via device name.

Parameter

> *modelName*   Name of sensor (*devName)*
> *scannerType*   Scanner type

Return value
> *General return values*

- **TScannerType**

TScannerType	Wert	scanCONTROL Type	Measuring range
StandardType	-1	-	-
scanCONTROL27xx_25	1000	27xx	25 mm
scanCONTROL27xx_100	1001	27xx	100 mm
scanCONTROL27xx_50	1002	27xx	50 mm
scanCONTROL27xx_xxx	1999	27xx	-
scanCONTROL26xx_25	2000	26xx	25 mm
scanCONTROL26xx_50	2002	26xx	50 mm
scanCONTROL26xx_100	2001	26xx	100 mm
scanCONTROL26xx_xxx	2999	26xx	-
scanCONTROL29xx_25	3000	29xx	25 mm
scanCONTROL29xx_50	3002	29xx	50 mm
scanCONTROL29xx_100	3001	29xx	100 mm
scanCONTROL29xx_10	3003	29xx	10 mm
scanCONTROL29xx_xxx	3999	29xx	-
scanCONTROL30xx_25	4000	30xx	25mm
scanCONTROL30xx_50	4001	30xx	50mm
scanCONTROL30xx_200	4002	30xx	200mm
scanCONTROL30xx_100	4003	30xx	100mm
scanCONTROL30xx_xxx	4999	30xx	-

scanCONTROL25xx_25	5000	25xx	25mm
scanCONTROL25xx_50	5002	25xx	50mm
scanCONTROL25xx_100	5001	25xx	100mm
scanCONTROL25xx_xxx	5999	25xx	-

- **GetLLTScalingAndOffset ()**

```
int
CInterfaceLLT::GetLLTScalingAndOffset(double *scaling, double *offset);
```

Query scaling and offset factors of connected LLT.

Parameter
       scaling   Scaling factor
       offset   Offset factor

Return value
       General return values

- **GetLLTScalingAndOffsetByType ()**

```
static int
CInterfaceLLT::GetLLTScalingAndOffsetByType (TScannerType scannerType,
                                             double *scaling, double *offset);
```

Query scaling and offset factors of a certain sensor type.

Parameter
       scannerType   Scanner type
       scaling   Scaling factor
       offset   Offset factor

Return value
       General return values

7.4   Feature functions

7.4.1   Set-/get functions

- **GetFeature ()**

```
int
CInterfaceLLT::GetFeature(unsigned int register, unsigned int *value);
```

Query currently set parameter value / Check availability of a feature according to the table in chapter.

Parameter
- *register*   Register address of function (FEATURE or INQUIRY)
- *value*   Value read from sensor

Return value
- *General return values*
- *Specific return value*:

ERROR_SETGETFUNCTIONS_WRONG_FEATURE_ADRESS	-155	The address of the selected property is wrong

- **SetFeature ()**

```
int
CInterfaceLLT::SetFeature(unsigned int register, unsigned int value);
```

Set feature parameter.

Parameter
- *register*   Register address of function (FEATURE)
- *value*   Value to be written to sensor

Return value
- *General return values*
- *Specific return value:*

ERROR_SETGETFUNCTIONS_WRONG_FEATURE_ADRESS	-155	The address of the selected property is wrong

7.4.2 Features / Parameter

The following paragraph illustrates how to use the FEATURE and INQUIRY registers. INQUIRY registers are used for confirmation of function availability and feature classification. FEATURE registers are used for querying and setting parameter values. The LSB is Bit 0.

The value read from the INQUIRY register can classify a parameter of the FEATURE register. To do so, the register defines the minimum and maximum value of the feature, if there is an automatic functionality and if the feature is available for the connected sensor type:

31	30..26	25	24	23..12	11..0
Feature avail. (1 bit)	Res. (5 bit)	Auto (1 bit)	Res. (1 bit)	Min. value (12 bit)	Max. value (12 bit)

The value of the FEATURE register can be interpreted aided by the Operation Manual Part B.

Example: Laser

Feature name	Inquiry address	Status and control address	Default setting
Laser	0xffffff0f00524	0xffffff0f00824	0x82000002

Bit	Function
1..0	**Laser Power** **0** OFF **1** reduced power **2** full power
11	**Pulsed Mode Enable** The laser is switched on only in the first half of the measurement interval. Additionally the external trigger output is delayed by half of the measurement interval (or 180 degrees). A synchronised slave sensor would measure during the master's idle time. It is recommended to set up Exposure Time < Idle Time.

Fig. 7: Excerpt from Operation Manual Part B

Thus the register address to be written to for changing the laser power is 0xf0f00824. Bits for adjusting the laser power are 0 and 1. The value $0 \vert_{\text{dec}}$ corresponds with laser off, $1 \vert_{\text{dec}}$ with reduced and $2 \vert_{\text{dec}}$ with full laser power. Bit 11 activates the laser pulse mode.

- **SERIAL_NUMBER**

FEATURE_FUNCTION_SERIAL_NUMBER FEATURE_FUNCTION_SERIAL (deprecated)	0xf0000410

Query the serial number of the connected sensor. This register is read-only.

- **CALIBRATION_SCALE and CALIBRATION_OFFSET**

FEATURE_FUNCTION_CALIBRATION_SCALE	0xf0a00000
FEATURE_FUNCTION_CALIBRATION_OFFSET	0xf0a00004

Query the scaling and the offset value of the connected sensor. This register is read-only. Note: Only supported by scanCONTROL 30xx series.

- **LASER**

FEATURE_FUNCTION_LASER FEATURE_FUNCTION_LASERPOWER (deprecated)	0xf0f00824
INQUIRY_FUNCTION_LASER INQUIRY_FUNCTION_LASERPOWER (deprecated)	0xf0f00524

Query and control of the laser power: 0 (off), 1 (reduced), 2 (full). Depending on the device, the polarity of the external laser switch-off or the laser pulse mode can be set. The profile transmitted directly after setting the laser power might be corrupted.
See *OpManPartB.html#laser* or *#laserpower* for the specific type of sensor.

- **ROI1_PRESET**

FEATURE_FUNCTION_ROI1_PRESET FEATURE_FUNCTION_MEASURINGFIELD (deprecated)	0xf0f00880
INQUIRY_FUNCTION_ROI_PRESET INQUIRY_FUNCTION_MEASURINGFIELD (deprecated)	0xf0f00580

Querying or setting of a region of interest on the sensor array or activation of the advanced region of interest configuration. The profile transmitted directly after setting the ROI might be corrupted.
See *OpManPartB.html#roi1* or *#zoom* for the specific type of sensor.
An overview over available predefined measuring fields and the corresponding maximum frequencies can be found in the sensor specific QuickReference.html.

- **ROI1**

`FEATURE_FUNCTION_ROI1_POSITION` `FEATURE_FUNCTION_FREE_MEASURINGFIELD_X (depr.)`	0xf0b0200c
`FEATURE_FUNCTION_ROI1_DISTANCE` `FEATURE_FUNCTION_FREE_MEASURINGFIELD_Z (depr.)`	0xf0b02008

Set the start point and size of the X and Z axis of the region of interest 1. Values can go from 0 to 65535. The sensors matrix rotation has to be considered. To activate a 0 has to be written to FEATURE_FUNCTION_EXTRA_PARAMETER.
Requires: Firmware v43 or newer (with Firmware < v43 use EXTRA_PARAMETER register to set the ROI)
See *OpManPartB.html#roi1 or #extraparameter* for the specific type of sensor.

- **ROI1_TRACKING**

`FEATURE_FUNCTION_ROI1_TRACKING_DIVISOR` `FEATURE_FUNCTION_DYNAMIC_TRACK_DIVISOR (depr.)`	0xf0b02010
`FEATURE_FUNCTION_ROI1_TRACKING_FACTOR` `FEATURE_FUNCTION_DYNAMIC_TRACK_FACTOR (depr.)`	0xf0b02014

Set the encoder-controlled measuring field tracking function. To activate a 0 has to be written to FEATURE_FUNCTION_EXTRA_PARAMETER.
Requires: Firmware v43 or newer (with Firmware < v43 use EXTRA_PARAMETER register)
See *OpManPartB.html#roi1 or #extraparameter* for the specific type of sensor.

- **IMAGE_FEATURES**

`FEATURE_FUNCTION_IMAGE_FEATURES`	0xf0b02100

Register to activate/deactivate region of interest 2, region of no interest, exposure automatic reference region on the sensor matrix. Set the operating mode of the sensor.
Note: Only supported by scanCONTROL 30xx series.
See *OpManPartB.html#image_sensor_features* for the specific type of sensor.

- **ROI2**

`FEATURE_FUNCTION_ROI2_POSITION`	0xf0b02108
`FEATURE_FUNCTION_ROI2_DISTANCE`	0xf0b02104

Set the start point and size of the X and Z axis of the region of interest 2. Values can go from 0 to 65535. The sensors matrix rotation has to be considered.
Note: Only supported by scanCONTROL 30xx series.
See *OpManPartB.html#roi2* for the specific type of sensor.

- **RONI**

FEATURE_FUNCTION_RONI_POSITION	0xf0b02110
FEATURE_FUNCTION_RONI_DISTANCE	0xf0b0210c

Set the start point and size of the X and Z axis of the region of no interest. Values can go from 0 to 65535. The sensors matrix rotation has to be considered.
Note: Only supported by scanCONTROL 30xx series.
See *OpManPartB.html#roni* for the specific type of sensor.

- **TRIGGER**

FEATURE_FUNCTION_TRIGGER	0xf0f00830
INQUIRY_FUNCTION_TRIGGER	0xf0f00530

Query and control of the trigger mode setting. The profile transmitted directly after setting the trigger mode can be corrupted. In context of changing the trigger mode, the trigger interface is to be changed (see *DIGITAL_IO*). Changing the trigger settings resets the profile counter.
See *OpManPartB.html#trigger* for the specific type of sensor.

- **EXPOSURE_TIME**

FEATURE_FUNCTION_EXPOSURE_TIME FEATURE_FUNCTION_SHUTTERTIME (deprecated)	0xf0f0081c
INQUIRY_FUNCTION_EXPOSURE_TIME INQUIRY_FUNCTION_SHUTTERTIME (deprecated)	0xf0f0051c

Query and control of the exposure time in 10 µs steps. The value can be set between 1 and 4095. The automatic exposure mode can be set with this register as well.
See *OpManPartB.html#exposuretime* or *#shutter* for the specific type of sensor.

- **EA_REFERENCE_REGION**

FEATURE_FUNCTION_EA_REFERENCE_REGION_POSITION	0xf0b02118
FEATURE_FUNCTION_EA_REFERENCE_REGION_DISTANCE	0xf0b02114

Set the start point and size of the X and Z axis of the reference region for the automatic

exposure time. Values can go from 0 to 65535. The sensors matrix rotation has to be considered.
Note: Only supported by scanCONTROL 30xx series.
See *OpManPartB.html#exposureautomatic* for the specific type of sensor.

- **EXPOSURE_AUTOMATIC_LIMITS**

FEATURE_FUNCTION_EXPOSURE_AUTOMATIC_LIMITS	0xf0f00834
INQUIRY_FUNCTION_EXPOSURE_AUTOMATIC_LIMITS	0xf0f00534

Set the lower and upper limit for the automatic exposure algorithm. Values can go from 0 to 4095 (in 10 μs steps).
Note: Only supported by scanCONTROL 30xx series.
See *OpManPartB.html#exposureautomatic* for the specific type of sensor.

- **IDLE_TIME**

FEATURE_FUNCTION_IDLE_TIME FEATURE_FUNCTION_IDLETIME (deprecated)	0xf0f00800
INQUIRY_FUNCTION_IDLE_TIME INQUIRY_FUNCTION_IDLETIME (deprecated)	0xf0f00500

Query and control of the idle time in 10 μs steps. The value can be set between 1 and 4095. If the automatic exposure mode is activated, the idle time is automatically adjusted to match the intended profile frequency.
See *OpManPartB.html#idletime* for the specific type of sensor.

- **PROFILE_PROCESSING**

FEATURE_FUNCTION_PROFILE_PROCESSING FEATURE_FUNCTION_PROCESSING_PROFILEDATA (depr.)	0xf0f00804
INQUIRY_FUNCTION_PROFILE_PROCESSING INQUIRY_FUNCTION_PROCESSING_PROFILEDATA (depr.)	0xf0f00504

Query and control of profile processing parameter, like e.g. deactivation of the calibration, profile mirroring, measurement data post-processing, reflection determination or advanced exposure settings.
See *OpManPartB.html#profileprocessing* or *#processingprofile* for the specific type of sensor.

- **THRESHOLD**

FEATURE_FUNCTION_THRESHOLD	0xf0f00810
INQUIRY_FUNCTION_THRESHOLD	0xf0f00510

Query and control of the threshold for measurement data acquisition. For targets with multiple reflections, increasing the threshold can improve the raw data. Optionally the dynamic threshold can be activated in this context.

See *OpManPartB.html#threshold* for the specific type of sensor.

- **MAINTENANCE**

`FEATURE_FUNCTION_MAINTENANCE` `FEATURE_FUNCTION_MAINTENANCEFUNCTIONS (depr.)`	0xf0f0088c
`INQUIRY_FUNCTION_MAINTENANCE` `INQUIRY_FUNCTION_MAINTENANCEFUNCTIONS (depr.)`	0xf0f0058c

Query and control of internal maintenance settings (e.g. encoder counter).
See *OpManPartB.html#maintenance* for the specific type of sensor.

- **ANALOGFREQUENCY**

`FEATURE_FUNCTION_ANALOGFREQUENCY`	0xf0f00828
`INQUIRY_FUNCTION_ANALOGFREQUENCY`	0xf0f00528

Interrogating and setting of the frequency for the analogue output (only scanCONTROL 28xx). The frequency may be set between 0 and 150 whereat the counting value is equal to the frequency in kHz. At the setting of 0 kHz the analogue output will be turned off which is reasonable at profile frequencies higher than 500 Hz for avoiding an overflow in the analogue output.
Note: Only supported by scanCONTROL 28xx series.
See *OpManPartB.html#focus* for the specific type of sensor.

- **ANALOGOUTPUTMODES**

`FEATURE_FUNCTION_ANALOGOUTPUTMODES`	0xf0f00820
`INQUIRY_FUNCTION_ANALOGOUTPUTMODES`	0xf0f00520

Setting of the analogue output modes (only scanCONTROL 28xx). E.g. the voltage range and the polarity of the analogue outputs may be shifted.
Note: Only supported by scanCONTROL 28xx series.
See *OpManPartB.html#gain* for the specific type of sensor.

- **CMM_TRIGGER**

`FEATURE_FUNCTION_CMM_TRIGGER` `FEATURE_FUNCTION_CMMTRIGGER (deprecated)`	0xf0f00888
`INQUIRY_FUNCTION_CMM_TRIGGER` `INQUIRY_FUNCTION_CMMTRIGGER (deprecated)`	0xf0f00588

Configuration of the optional CMM triggers. The configuration of the CMM triggers consists of 4 instruction words. These instruction words have to be written successively. Only the last written instruction word can be read from the sensor.
Note: Not supported by scanCONTROL 25xx series.
See *OpManPartB.html#cmmtrigger* for the specific type of sensor.

- **PROFILE_REARRANGEMENT**

FEATURE_FUNCTION_PROFILE_REARRANGEMENT FEATURE_FUNCTION_REARRANGEMENT_PROFILE (depr.)	0xf0f0080c
INQUIRY_FUNCTION_PROFILE_REARRANGEMENT INQUIRY_FUNCTION_REARRANGEMENT_PROFILE (depr.)	0xf0f0050c

Parametrization of profile information to be transmitted in the transposed Container Mode.
See *OpManPartB.html#profilerearrangement* or *#rearrangementprofile* for the specific type of sensor.

- **PROFILE_FILTER**

FEATURE_FUNCTION_PROFILE_FILTER	0xf0f00818
INQUIRY_FUNCTION_PROFILE_FILTER	0xf0f00518

Applying Resampling, Median-Filter and/or Average-Filter.
See *OpManPartB.html#profilefilter* for the specific type of sensor.

- **DIGITAL_IO**

FEATURE_FUNCTION_DIGITAL_IO FEATURE_FUNCTION_RS422_INTERFACE_FUNCTION (depr.)	0xf0f008c0
INQUIRY_FUNCTION_DIGITAL_IO INQUIRY_FUNCTION_RS422_INTERFACE_FUNCTION (depr.)	0xf0f005c0

Parameter to configure the RS422 interface or digital inputs.
See *OpManPartB.html#ioconfig or #capturesize* for the specific type of sensor.

- **PACKET_DELAY**

FEATURE_FUNCTION_PACKET_DELAY	0x00000d08

Ethernet packet delay to operate several scanners connected to a switch. The parameter is set in μs. The range is from 0 to 1000μs.

- **TEMPERATURE**

FEATURE_FUNCTION_TEMPERATURE	0xf0f0082c
INQUIRY_FUNCTION_TEMPERATURE	0xf0f0052c

Read sensor temperature in 0.1 K steps. Before the temperature can be read, the temperature measurement has to be triggered by writing 0x86000000 to the feature register.
See *OpManPartB.html#temperature* for the specific type of sensor.

- **EXTRA_PARAMETER**

`FEATURE_FUNCTION_EXTRA_PARAMETER` `FEATURE_FUNCTION_SHARPNESS (deprecated)`	0xf0f00808
`INQUIRY_FUNCTION_EXTRA_PARAMETER` `INQUIRY_FUNCTION_SHARPNESS (deprecated)`	0xf0f00508

Register for setting the peak filter, free measuring field and angle/offset calibration. The values are set via multiple write operations to the register. Only the last written value can be read back. Since DLL version 3.7 / sensor firmware v43, this register is mainly used to activate certain register settings.
See *OpManPartB.html#extraparameter or #sharpness* for the specific type of sensor.

- **PEAKFILTER**

`FEATURE_FUNCTION_PEAKFILTER_WIDTH`	0xf0b02000
`FEATURE_FUNCTION_PEAKFILTER_HEIGHT`	0xf0b02004

Set the minimum and maximum width/height of the peak filter. Values can go from 0 to 1023.
Requires: Firmware v43 or newer (with Firmware < v43 use EXTRA_PARAMETER register)

- **CALIBRATION**

`FEATURE_FUNCTION_CALIBRATION_0 - 7`	0xf0b02020 - 0xf0b0203c

Set the calibration offset and angle. To activate a 0 has to be written to FEATURE_FUNCTION_EXTRA_PARAMETER.
Requires: Firmware v43 or newer (with Firmware < v43 use EXTRA_PARAMETER register)

7.5 Special feature functions

7.5.1 Software Trigger

- **TriggerProfile ()**

```
int
CInterfaceLLT::TriggerProfile();
```

Execute a software trigger signal to trigger one profile.

Parameter
> *CInterfaceLLT* *LLT class*

Return value
> *General return values*

- **TriggerContainer ()**

```
int
CInterfaceLLT::TriggerContainer();
```

Execute a software trigger signal to trigger one container.

Parameter
>*CInterfaceLLT*   LLT class

Return value
>*General return values*

Requires: Firmware v46 or newer.
The sensor has to be in frametrigger mode (see Digital IO) to be able to use this function. Otherwise, the trigger container mode has to be activated with the subsequent function calls.

- **TriggerContainerEnable ()**
- **TriggerContainerDisable ()**

```
int
CInterfaceLLT::TriggerContainerEnable();

int
CInterfaceLLT::TriggerContainerDisable();
```

Activate/deactivate the trigger container mode to be able to use TriggerContainer() without set frametrigger mode (see Digital IO).

Parameter
>*CInterfaceLLT*   LLT class

Return value
>*General return values*

7.5.2   Profile configuration

- **GetProfileConfig ()**

```
int
CInterfaceLLT::GetProfileConfig(TProfileConfig *profileConf);
```

Query the current profile configuration.

Parameter
>*profileConf*   Profile configuration read from sensor

Return value

General return values

- **SetProfileConfig ()**

```
int
CInterfaceLLT::SetProfileConfig(TProfileConfig profileConf);
```

Profile configuration to be set.

Parameter
> *profileConf*   Profile configuration to be set

Return value
> *General return values*
> Specific return value:

ERROR_ SETGETFUNCTIONS_WRONG _PROFILE_CONFIG	-152	The requested profile configuration is not available

- **ProfileConfig**

Available ProfileConfig settings:

Value name	Value	Description
PROFILE	1	Profile data of all four stripes
VIDEO_IMAGE	2	Matrix video image
PARTIAL_PROFILE	5	Partial profile as set by SetPartialProfile
CONTAINER	6	Container data

7.5.3   Profile resolution / Points per profile

- **GetResolution ()**

```
int
CInterfaceLLT::GetResolution(unsigned int *value);
```

Query of the currently set profile resolution / measuring points per profile.

Parameter
> *value*   Profile resolution queried

Return value
> *General return values*

- **SetResolution ()**

```
int
CInterfaceLLT::SetResolution(unsigned int value);
```

Set the profile resolution / points per profile. For changing the resolution, the profile data transmission has to be stopped. After changing the resolution all partial profile settings are lost.

Parameter
> *value*   Profile resolution queried

Return value
> *General return values*
> *Specific return value:*

ERROR_ SETGETFUNCTIONS_NOT _SUPPORTED_RESOLUTION	-153	The requested resolution is not supported

- **GetResolutions ()**

```
int
CInterfaceLLT::GetResolutions(unsigned int *value, unsigned int size);
```

Query available profile resolutions.

Parameter
> *value*   Array with available profile resolutions
> *size*   Size of array

Return value
> *Number of available resolutions*
> *Specific return value:*

ERROR_ SETGETFUNCTIONS_NOT _SUPPORTED_RESOLUTION	-156	The size of the passed field is too small

7.5.4 Container size

- **GetProfileContainerSize ()**

```
int
CInterfaceLLT::GetProfileContainerSize(unsigned int *width,
                                                unsigned int *height);
```

Query of currently set container size.

Parameter
> *width*   Container width read from sensor
> *height*   Container height read from sensor

Return value
>   *General return values*

- **SetProfileContainerSize ()**

```
int
CInterfaceLLT::SetProfileContainerSize(unsigned int width, unsigned int height);
```

Set container size. The container width is set automatically if *SetFeature(FEATURE_FUNCTION_REARRANGEMENT_PROFILE)* is called. The height can be freely set from 0 to the maximum container height and determines how many profiles are transmitted within one container. The container height shouldn't be higher than three times the profile frequency. If the "connection of successive profiles" (see *OpManPartB.html#rearrangementprofile)* is activated, the height * width of an image has to be an integral multiple of 16384. If it is tried to set up another height value, the height will be adjusted automatically to the next matching value.

Parameter
>   *width*   Container width to be set
>   *height*   Container height to be set

Return value
>   *General return values*
>   *Specific return values:*

`ERROR_SETGETFUNCTIONS_WRONG _PROFILE_SIZE`	-157	The size for the container is wrong
`ERROR_SETGETFUNCTIONS_MOD_4`	-158	The container width is not divisible by 4

- **GetMaxProfileContainerSize ()**

```
int
CInterfaceLLT::GetMaxProfileContainerSize(unsigned int *maxWidth,
                                          unsigned int *maxHeight);
```

Query the maximal possible container size. If the maximum width is 64, the container mode is not supported by the connected scanCONTROL.

Parameter
>   *maxWidth*   Maximum container width
>   *maxHeight*   Maximum container height

Return value
>   *General return values*

7.5.5 Allocated buffer for profile polling

- **GetHoldBuffersForPolling ()**

```
int
CInterfaceLLT::GetHoldBuffersForPolling(unsigned int *holdBuffersForPolling);
```

Query the number of buffers allocated for polling profile with *GetActualProfile().*

Parameter
  holdBuffersForPolling   Buffer count queried

Return value
  General return values

- **SetHoldBuffersForPolling ()**

```
int
CInterfaceLLT::SetHoldBuffersForPolling(unsigned int holdBuffersForPolling);
```

Setting the profile count that the llt.dll may hold for *GetActualProfile()*; the buffer is set up as FIFO in case the set value is greater than 0. A larger count lets the LLT.dll hold more profiles before profiles will be dropped. This decreases the risk of profile loss. The count must not be higher than half the total buffer count. Default value: 1.

Parameter
  holdBuffersForPolling   Buffer count set

Return value
  General return values
  Specific return value:

`ERROR_SETGETFUNCTIONS_WRONG` `_BUFFER_COUNT`	-150	Buffer count is not in the range of >=2 and <= 200

7.5.6 Buffer count

A high buffer count is useful when using high profile frequencies, slow computing hardware and/or a PC with a lot of background activity. For container and video mode transmission maximum four buffers are recommended.

- **GetBufferCount ()**

```
int
CInterfaceLLT::GetBufferCount(unsigned int *value);
```

Query the set number of buffers in the driver for data transmission.

Parameter
  value   Buffer count

Return value
> *General return values*

- **SetBufferCount ()**

```
int
CInterfaceLLT::SetBufferCount(unsigned int value);
```

Set the number of buffers in the driver for data transmission.

Parameter
> *value*   Buffer count to be set

Return value
> *General return values*
> *Specific return value*:

ERROR_SETGETFUNCTIONS_WRONG _BUFFER_COUNT	-150	Count of the required buffer does not lie in the range of >=2 and <= 200

7.5.7   Packet size

scanCONTROL supports the packet sizes 128, 256, 512, 1024, 2048 and 4096 bytes. For Ethernet connections, packets larger than 1024 bytes require the support of jumbo frames by all devices, especially by the receiving network card.

- **GetPacketSize ()**

```
int
CInterfaceLLT::GetPacketSize(unsigned int *value);
```

Query the active packet size of the Ethernet streaming packets.

Parameter
> *value*   Packet size queried

Return value
> *General return values*

- **SetPacketSize ()**

```
int
CInterfaceLLT::SetPacketSize(unsigned int value);
```

Set the active packet size for the size of the Ethernet streaming packets.

Parameter
> *value*   Packet size to be set

Return value
> *General return values*
> *Specific return value*:

ERROR_SETGETFUNCTIONS_PACKET_SIZE	-151	The requested packet size is not supported

- **GetMinMaxPacketSize ()**

```
int
CInterfaceLLT::GetMinMaxPacketSize(unsigned int *minPacketSize,
                                              unsigned int *maxPacketSize);
```

Query the minimum and maximum packet sizes of the Ethernet streaming packets.

Parameter
> *minPacketSize*   Minimal packet size
> *maxPacketSize*   Maximal packet size

Return value
> *General return values*

7.5.8   Timeout for communication supervision to the sensor

Setting and reading the heartbeat timeout in milliseconds to monitor the connection between linllt and scanCONTROL device. This is the time between two monitoring packets. A connection abort will occur if the device does not receive packets after three times the set up value. Especially while debugging, a heartbeat timeout set too small is reason for connection loss.

- **GetEthernetHeartbeatTimeout ()**

```
int
CInterfaceLLT::GetEthernetHeartbeatTimeout(unsigned int *timeout);
```

Query the set heartbeat timeout.

Parameter
> *CInterfaceLLT*   LLT class
> *timeout*   Heartbeat timeout queried

Return value
> *General return values*

- **SetEthernetHeartbeatTimeout ()**

```
int
CInterfaceLLT::SetEthernetHeartbeatTimeout(unsigned int timeout);
```

Set the heartbeat timeout in ms. Values between 500 and 1.000.000.000 ms are allowed.

Parameter
   CInterfaceLLT   LLT class
   Value   Heartbeat timeout to be set

Return value
   General return values
   Specific return value:

ERROR_SETGETFUNCTIONS _HEARTBEAT_TOO_HIGH	-162	Parameter value too large

7.5.9   Loading and saving of user modes

Loading and storing the user mode. All settings of a scanCONTROL can be stored in a user mode so that all settings become active again immediately after a reset or restart. This is particularly expedient for post processing applications. Loading the user mode cannot be performed during an active profile / container transmission. User mode 0 can only be loaded as it contains the factory settings.

- **GetActualUserMode ()**

```
int
CInterfaceLLT::GetActualUserMode(unsigned int *actualUserMode,
                                            unsigned int *userModeCount);
```

Query the last loaded user mode / parameter queue. The scanCONTROL sensors of type 26xx/27xx/29xx/30xx support 16 user modes.

Parameter
   actualUserMode Currently loaded user mode
   userModeCount Available user modes

Return value
   General return values

- **ReadWriteUserModes ()**

```
int
CInterfaceLLT::ReadWriteUserModes(gboolean write, unsigned int userMode);
```

Loading or storing a user mode. If *write* is *false*, the user mode specified by *userMode* is loaded; otherwise the current settings are stored to this user mode. After loading of a user mode the sensor must be reconnected.

Parameter
   write   Loading (false) or saving (true) of an user mode
   userMode   User mode to be loaded or saved

Return value
   General return values

Specific return values:

ERROR_SETGETFUNCTIONS_USER _MODE_TOO_HIGH	-160	The specified user mode number is not available
ERROR_SETGETFUNCTIONS_USER _MODE_FACTORY_DEFAULT	-161	User mode 0 cannot be overwritten (factory settings)

7.6   Register functions

7.6.1   Register callback for profile reception

These callbacks are, after they have been registered, called after the receipt of a profile/ container and have as parameter a pointer to the profile/container data, the corresponding size of a data field and an userData parameter.

The callback is intended for the processing of profiles / containers with a high profile frequency. During the callback profiles / container may be copied in a buffer for a later or a processing synchrony or asynchrony to the callback. A processing during the callback is not recommendable as for the time the callback needs for processing the linLLT is not able to fetch new profiles / container from the driver. Possibly by this it may amount to profile/container failures.

The profile / container data in the buffer passed by the callback must not be changed.

- **RegisterBufferCallback ()**

```
int
 CInterfaceLLT::RegisterBufferCallback(gpointer *bufferCb, gpointer userData);
```

Registrieren callback for profile reception

Parameter
  bufferCb   Pointer to callback function
  userData   User defined data, available in callback

Return value
  General return values

7.6.2   Register error message for error handling

- **RegisterControlLostCallback ()**

```
int
CInterfaceLLT::RegisterControlLostCallback(gpointer *controlLostCb,
                                                  gpointer userData);
```

Register callback for the case the connection to a sensor is lost.
Parameter
  controlLostCb   Pointer to ControlLost function
  userData   User defined data, available in callback

Return value
  General return values

7.7   Profile transmission function

7.7.1   Start/stop profiles transmission

- **TransferProfiles ()**

```
int
CInterfaceLLT::TransferProfiles(TTransferProfileType transferProfileType,
                                                     gboolean enable);
```

Start or stop profile transmission. After the first start of a transfer it may take up to 100 ms before the first profiles/container arrive via callback. If a transfer is terminated, the function waits automatically till the driver has returned all buffers.

Parameter
  transferProfileType   Profile transfer type
  enable   Start (true) or stop (false) transmission

Return value
  General return values

- **TTransferProfileType**

Available TTransferProfileTypes:

Constant return value	Value	Description
NORMAL_TRANSFER	0	Activation of a continuous transfer of profiles
NORMAL_CONTAINER_MODE	2	Activation of a continuous transfer in the container mode

- **SetStreamNiceValue ()**

```
int
CInterfaceLLT::SetStreamNiceValue(unsigned int niceValue);
```

Sets nice value of stream thread (19 to -20). Smaller values set higher priorities, but may need advanced user privileges. (=arv_make_thread_high_priority)

Parameter
  niceValue   nice value

Return value
  General return values

- **GetStreamNiceValue ()**

```
int
CInterfaceLLT::GetStreamNiceValue(unsigned int *niceValue);
```

Reads the currently set nice value.

Parameter
        niceValue   nice value

Return value
        General return values

- **SetStreamPriority ()**

```
int
CInterfaceLLT::SetStreamPriority (unsigned int priority);
```

Sets the (realtime-)priority of the stream thread (0 to 99). Higher values set higher priorities, but may need advanced user privileges. (=arv_make_thread_realtime)

Parameter
        priority   Priority

Return value
        General return values

- **GetStreamPriority ()**

```
int
CInterfaceLLT::SetStreamPriority(unsigned int *priority);
```

Reads the currently set priority value.

Parameter
        priority   Priority

Return value
        General return values

- **GetStreamPriorityState ()**

```
int
CInterfaceLLT::GetStreamPriorityState (TStreamPriorityState *prioState);
```

Reads the current priority state of the stream thread.

Parameter
  prioState Priority state

Return value
  General return values

- **TStreamPriorityState**

Available TStreamPriorityStates:

Name	Value	Description
PRIO_NOT_SET	0	No priority set. Uses system default.
PRIO_SET_SUCCESS	1	Priority set successfully
PRIO_SET_RT_FAILED	2	Priority value could not be set. Check user privileges. (arv_make_thread_realtime)
PRIO_SET_NICE_FAILED	3	nice value could not be set. Check user privileges. (arv_make_thread_high_priority)
PRIO_SET_FAILED	4	Priority values could not be set. Check user privileges.

- **GetStreamStatistics ()**

```
int
GetLLTOffsetAndScaling (unsigned long *completedBuffer,
                        unsigned long *failures, unsigned long underruns);
```

Queries the current transmission statistics. A transmission must be active during function call.

Parameter
  completedBuffer Successfully completed buffers
  failures Buffer failures
  underruns Buffer underruns

Return value
  General return values

7.7.2 Fetch current profile / container / video image

- **GetActualProfile ()**

```
int
CInterfaceLLT::GetActualProfile(unsigned char *buffer, int bufferSize,
              TProfileConfig profileConfig, unsigned int lostProfiles);
```

Fetching the active profile/container/video image from holding buffer.

Parameter
> buffer   Transmission buffer
> buffersize   Size of transmission buffer
> profileConfig   Profile configuration of transmission
> lostProfiles   Lost profiles

Return value
> *Number of bytes copied into buffer*
> *General return values*
> *Specific return values:*

`ERROR_PROFTRANS_WRONG_PROFILE_CONFIG`	-102	Not able to convert the loaded profile into the requested profile configuration
`ERROR_PROFTRANS_FILE_EOF`	-103	The end-of-file during the loading of profiles has been reached
`ERROR_PROFTRANS_NO_NEW_PROFILE`	-104	Since the last call of *GetActualProfile()* no new profile has been received
`ERROR_PROFTRANS_BUFFER_SIZE_TOO_LOW`	-105	The buffer size of the passed buffer is too small
`ERROR_PROFTRANS_NO_PROFILE_TRANSFER`	-106	The profile transfer has not been started and no file is loaded

7.7.3 Convert profile data

- **ConvertProfile2Values ()**

```
static int
CInterfaceLLT::ConvertProfile2Values(const unsigned char *buffer,
unsigned int bufferSize, unsigned int resolution, TProfileConfig profileConfig,
 TScannerType scannerType, unsigned int reflection, unsigned short *width,
    unsigned short *intensity, unsigned short *threshold, double *x, double *z,
        unsigned int *m0, unsigned int *m1);
```

Conversion of profile data in coordinates and further measuring point information. The size of the data arrays must correspond to the profile resolution.

Parameter
> buffer   Profile buffer
> bufferSize   Size of profile buffer
> resolution   Points per profile
> profileConfig   Profile Config (must be PROFILE)
> scannerType   Sensor type
> reflection   Profile stripe to be evaluated
> width   Array for reflection width
> maximum   Array for maximum intensity
> threshold   Array for threshold values
> x   Array for position values
> z   Array for distance values
> m0   Array for moment 0
> m1   Array for moment 1

Return value
> *General return values*
> *Additional return values in case of success*
> *Specific return value*:

ERROR_PROFTRANS_REFLECTION _NUMBER_TOO_HIGH	-110	The count of the requested stripes is bigger than 3

- **ConvertPartProfile2Values ()**

```
static int
CInterfaceLLT::ConvertPartProfile2Values(const unsigned char buffer,
    unsigned int bufferSize, TPartialProfile partialProfile,
        TScannerType scannerType, unsigned int reflection,
            unsigned short *width, unsigned short *intensity,
                    unsigned short *threshold, double *x, double *z,
                                    unsigned int *m0, unsigned int *m1);
```

Conversion of partial profile data in coordinates and further measuring point information. The size of the data arrays must correspond to the PointCount of the PARTIAL_PROFILE parameter.

Parameter
> *buffer*   Profile buffer
> *bufferSize*   Size of profile buffer
> partialProfile   Partial profile
> *scannerType*   Sensor type
> *reflection*   Profile stripe to be evaluated
> *width*   Array for reflection width
> *maximum*   Array for maximum intensity
> *threshold*   Array for threshold values
> *x*   Array for position values
> *z*   Array for distance values
> *m0*   Array for moment 0
> *m1*   Array for moment 1

Return value
> *General return values*
> *Additional return values in case of success*
> *Specific return value:*

ERROR_PROFTRANS_REFLECTION _NUMBER_TOO_HIGH	-110	The count of the requested stripes is bigger than 3

- **ConvertRearrangendContainer2Values ()**

```
static int
CInterfaceLLT::ConvertRearrangendContainer2Values (const unsigned char *buffer,
unsigned int bufferSize, unsigned int rearrangement,
 unsigned int numberProfiles,  TScannerType scannerType,
   unsigned int reflection, unsigned short *width,
     unsigned short *intensity, unsigned short *threshold, double *x, double *z);
```

Extracts and converts rearranged container raw data to coordinates and advanced profile information. The whole container is converted, thus more than one profile. This means the arrays for X, Z, … must have a size of resolution*number of profiles.

Parameter

buffer	Profile buffer
bufferSize	Size of profile buffer
rearrangement	Value of rearrangement parameter
numberProfiles	Number of profiles in container
scannerType	Sensor type
reflection	Profile stripe to be evaluated
width	Array for reflection width
maximum	Array for maximum intensity
threshold	Array for threshold values
x	Array for position values
z	Array for distance values

Return value

General return values
Additional return values in case of success
Specific return value:

ERROR_PROFTRANS_REFLECTION _NUMBER_TOO_HIGH	-110	The count of the requested stripes is bigger than 3

- **Return values in case of success**

If the return value was >0, the set bits of the value describe which arrays have been filled:

Bit	Constant	Description
8	CONVERT_WIDTH	The array for the reflection width has been filled with data
9	CONVERT_MAXIMUM	The array for the maximum intensity has been filled with data
10	CONVERT_THRESHOLD	The array for the threshold has been filled with data
11	CONVERT_X	The array for the position coordinates has been filled with data
12	CONVERT_Z	The array for the distance coordinates has been filled with data
13	CONVERT_M0	The array for the M0 has been filled with data
14	CONVERT_M1	The array for the M1 has been filled with data

7.8   Functions for transmission of partial profiles

The scanCONTROL offers the possibility to restrict the transferred profile. The advantage of this procedure is the reduced size of the transferred data. Furthermore, the unused ranges of a profile may be cut out already directly in the scanCONTROL.

- **GetPartialProfile ()**

```
int
CInterfaceLLT::GetPartialProfile(TPartialProfile *partialProfile);
```

Query the current partial profile setting set on the scanCONTROL.

Parameter
    partialProfile   Reference to partial profile structure

Return value
    General return values
    Specific return values:

ERROR_PARTPROFILE_NO_PART_PROF	-350	The profile configuration is not set to PARTIAL_PROFILE -> call SetProfileConfig(PARTIAL_PROFILE);

- **SetPartialProfile ()**

```
int
CInterfaceLLT::SetPartialProfile(TPartialProfile *partialProfile);
```

By using this function the partial profile transfer of the scanCONTROL can be adjusted. Before setting the partial profile parameters, the profile configuration has to be set to PARTIAL_PROFILE. All parameter of the *SetPartialProfile()* function always have to be a multiple of the respective unit size of the function *GetPartialProfileUnitSize()*.

Parameter
    partialProfile   Reference to partial profile structure

Return value
    General return values
    Specific return values:

ERROR_PARTPROFILE_NO_PART_PROF	-350	The profile configuration is not set to PARTIAL_PROFILE -> call SetProfileConfig(PARTIAL_PROFILE);
ERROR_PARTPROFILE_TOO_MUCH_BYTES	-351	The count of bytes per point is too high -> change nStartPointData or nPointDataWidth
ERROR_PARTPROFILE_TOO_MUCH_POINTS	-352	The count of points is too high -> change nStartPoint or nPointCount
ERROR_PARTPROFILE_NO_POINT_COUNT	-353	nPointCount or nPointDataWidth is 0
ERROR_PARTPROFILE_NOT_MOD_UNITSIZE_POINT	-354	nStartPoint or nPointCount are not a multiple of nUnitSizePoint
ERROR_PARTPROFILE_NOT_MOD_UNITSIZE_DATA	-355	nStartPointData or nPointDataWidth are not a multiple of nUnitSizePointData

- **GetPartialProfileUnitSize ()**

```
int
CInterfaceLLT::GetPartialProfileUnitSize(unsigned int *unitSizePoint,
                                         unsigned int *unitSizePointData);
```

This function returns the increments for adjusting the partial profile.

Parameter
> *unitSizePoint*   Unit size of point increments
> *unitSizePointData*   Unit size of data increments

Return value
> *General return values*

7.9   Timestamp extraction functions

- **Timestamp2TimeAndCount ()**

```
static int
CInterfaceLLT::Timestamp2TimeAndCount(unsigned char *buffer,
 double *shutterOpen, double *shutterClose, unsigned int *profileCount,
                                      unsigned short encTimes2OrDigIn);
```

This function evaluates the whole timestamp of a profile. It returns the internal timestamp of the beginning and the end of the shutter interval and the consecutive profile numbers.

Parameter
> *buffer*   Reference to timestamp bytes of profile buffer
> *shutterOpen*   Timestamp shutter open
> *shutterClosed*   Timestamp shutter closed
> *profileCount*   Profile count
> *encTimes2OrDigIn* 2x value encoder counter or state digital inputs (binary)

Return value
> *General return values*

7.10 Calibration

- **SetCustomCalibration ()**

```
int
CInterfaceLLT::SetCustomCalibration(double cX, double cZ, double angle,
                                    double sX, double sZ);
```

Calibration of sensor mounting position through rotation and shift of a profile. For permanent saving of the calibration the user mode has to be saved. The calibration is valid for every user mode.

Parameter

cX	Rotational centrum x in mm
cZ	Rotational centrum z in mm
angle	Rotation angle in degree
sX	Shift of rotational centrum x in mm
sZ	Shift of rotational centrum z in mm

Return value
   General return values

`ERROR_SETGETFUNCTIONS_WRONG _FEATURE_ADRESS`	-155	The address of the selected property is wrong

- **ResetCustomCalibration ()**

```
int
CInterfaceLLT::ResetCustomCalibration();
```

Reset calibration of sensor mounting position. For permanent saving of the calibration the user mode has to be saved. The calibration is valid for every user mode.

Return value
   General return values

`ERROR_SETGETFUNCTIONS_WRONG _FEATURE_ADRESS`	-155	The address of the selected property is wrong

7.11 Post processing functions

In context of post processing the scanCONTROL may apply several modules to the profiles. These modules are only available in the SMART / GAP options of the sensor.

- **ReadPostProcessingParameter ()**

```
int
CInterfaceLLT::ReadPostProcessingParameter(unsigned int *parameter,
                                           unsigned int size);
```

Read post processing parameters.

Parameter

CInterfaceLLT	LLT class
parameter	Pointer to post processing parameter array
size	Size of post processing parameter (max. 1024 DWORDs)

Return value
   General return values

- **WritePostProcessingParameter ()**

```
int
CInterfaceLLT::WritePostProcessingParameter(unsigned int *parameter,
                                            unsigned int size);
```

Write post processing parameters.

Parameter

CInterfaceLLT	LLT class
parameter	Pointer to Post processing parameter array
size	Size of post processing parameter (max. 1024 DWORDs)

Return value
General return values

7.12 MISC

- **SetPeakFilter ()**

```
int
CInterfaceLLT::SetPeakFilter(unsigned short minWidth, unsigned short maxWidth,
                             unsigned short minIntensity, unsigned short maxIntensity);
```

Sets the peak filter, which allows limiting the characteristics for valid points.

Parameter

minWidth	Min. valid reflection width
maxWidth	Max. valid reflection width
minIntensity	Min. valid intensity
maxIntensity	Max. valid intensity

Return value
General return values
Like SetFeature()

- **SetFreeMeasuringField ()**

```
int
CInterfaceLLT::SetFreeMeasuringField(unsigned short startX, unsigned short sizeX,
                                     unsigned short startZ, unsigned short sizeZ);
```

Sets a custom measuring field with specific size. The *start* and *size* values are to be seen in reference to 65535. Example: *start_z* = 20000 → 20000/65535 * 100 % = 30.52 %; *start_z* is therefore at 30.52 % of the sensor matrix height. For a scanCONTROL 2900 (height: 1024 pixel) this means approx. at pixel 312. For correct results the matrix rotation of the current sensor type has to be considered (check with Developer Tool).

Parameter

startX	Start value X
sizeX	Size of X
startZ	Start value Z

>   *sizeZ*   Size of Z

> Return value
>   *General return values*
>   *Like SetFeature()*

- **SetDynamicMeasuringFieldTracking ()**

```
int
CInterfaceLLT:: SetDynamicMeasuringFieldTracking (unsigned short divX,
            unsigned short divZ, unsigned short shiftX, unsigned short shiftZ);
```

Sets the dynamic, Encoder-controlled measuring field.

> Parameter
>   *divX*   Steps in X
>   *divZ*   Steps in Z
>   *shiftX*   Shift in X
>   *shiftZ*   Shift Z

> Return value
>   *General return values*
>   *Like SetFeature()*

7.13   Event handling (for Win compatibility)

- **CreateEvent ()**

```
static EHANDLE*
CInterfaceLLT::CreateEvent();
```

Creates event handle.

> Return value
>   *Event handle or error*

- **FreeEvent ()**

```
static void
CInterfaceLLT::FreeEvent(EHANDLE *eventHandle);
```

Frees resources of event.

> Parameter
>   *eventHandle*   Event handle

- **SetEvent ()**

```
static void
CInterfaceLLT::CreateEvent(EHANDLE *eventHandle);
```

Sets event.

<u>Parameter</u>
  eventHandle   Event handle

- **ResetEvent ()**

```
static void
CInterfaceLLT::ResetEvent(EHANDLE *eventHandle);
```

Resets event.

<u>Parameter</u>
  eventHandle   Event handle

- **WaitForSingleObject ()**

```
static int
CInterfaceLLT::WaitForSingleObject(EHANDLE *eventHandle, unsigned int timeout);
```

Blocks until event is set or timeout is expired.

<u>Parameter</u>
  eventHandle   Event handle
  timeout   Timeout in ms

<u>Return value</u>
  0 if successful

7.14 Save/Export configuration

- **ExportLLTConfig ()**

```
int
CInterfaceLLT::ExportLLTConfig(const char *fileName);
```

Exporting the current configuration of the scanCONTROL. This configuration file contains all relevant parameters and is primarily intended for post processing applications. The file format complies with the communications protocol for the serial connection with the scanCONTROL. The configuration files created thus can be transmitted to the scanCONTROL without changes via the serial port using a terminal program or via ImportLLTConfig.

Parameter
    CInterfaceLLT   LLT class
    fileName   Name of file to be exported

Return value
    General return values
    Specific return value:
    Return values of GetFeature()

`ERROR_READWRITECONFIG_CANT _CREATE_FILE`	-500	The specified file cannot be created

- **ExportLLTConfigString ()**

```
int
CInterfaceLLT::ExportLLTConfigString(const char *configData, int size);
```

Exporting the current configuration of the scanCONTROL. This configuration string contains all relevant parameters and is primarily intended for post processing applications. The string format complies with the communications protocol for the serial connection with the scanCONTROL. The configuration files created thus can be transmitted to the scanCONTROL without changes via the serial port using a terminal program or via ImportLLTConfigString.

Parameter
    CInterfaceLLT   LLT class
    configData   char array for config data
    size   char array size

Return value
    General return values
    Specific return value:
    Return values of GetFeature()

`ERROR_READWRITECONFIG_QUEUE_TO _SMALL`	-502	Data array to small

- **ImportLLTConfig ()**

```
int
CInterfaceLLT::ImportLLTConfig(const char *fileName, bool ignoreCalibration);
```

Reads and sets the sensor parameters exported by ExportLLTConfig and is also able to read .sc1-files as long as they've been saved with scanCONTROL Configuration Tools Version 5.2 or newer. The ignore calibration flag specifies if the custom calibration of the sensor is also imported from the file. After loading of a configuration file the sensor must be reconnected.

Parameter
    CInterfaceLLT   LLT class
    fileName   Path/name of file to be imported
    ignoreCalibration   if true, do not import calibration data from file

Return value
    General return values
    Specific return value:
    Return values of SetFeature()

`ERROR_READWRITECONFIG_CANT_OPEN_FILE`	-502	The specified file cannot be opened
`ERROR_READWRITECONFIG_FILE_EMPTY`	-503	The specified file is empty
`ERROR_READWRITE_UNKNOWN_FILE`	-504	The imported data has not the expected format

- **ImportLLTConfigString ()**

```
int
CInterfaceLLT::ImportLLTConfigString(const char *configData, int size,
                                                bool ignoreCalibration);
```

Reads and sets the sensor parameters exported by ExportLLTConfigString. The ignore calibration flag specifies if the custom calibration of the sensor is also imported from the string.

Parameter
    CInterfaceLLT   LLT class
    configData   char array with config data
    size   char array size
    ignoreCalibration   if true, do not import calibration data from string

Return value
    General return values
    Specific return value:
    Return values of SetFeature()

`ERROR_READWRITE_UNKNOWN_FILE`	-504	The imported data has not the expected format

- **SaveGlobalParameter ()**

```
int
CInterfaceLLT::SaveGlobalParameter();
```

Save IP configuration and calibration independent of user mode.

Parameter
    CInterfaceLLT   LLT class

Return value
    General return values

8   Appendix

8.1   General return values

All functions of the interface return an int value as return value. If the return value of a function is greater than or equal to GENERAL_FUNCTION_OK respectively '1', the function has been successful; if the return value is GENERAL_FUNCTION_NOT_AVAILABLE respectively '0' or negative, an error occurred.

For the differentiation of the single return values several constants are available. In the following table all general return values which may be returned by functions are listed. For the single functional groups additionally there may also be special return / error values.

Constant return value	Value	Description
GENERAL_FUNCTION_OK	1	Function successfully executed
GENERAL_FUNCTION_NOT_AVAILABLE	0	This function is not available, possibly using a new DLL or switching to the Ethernet mode
ERROR_GENERAL_NOT_CONNECTED	-1001	There is no connection to the scanCONTROL -> call *Connect()*
ERROR_GENERAL_DEVICE_BUSY	-1002	The connection to the scanCONTROL is interfered or disconnected -> reconnect and check interface of the scanCONTROL
ERROR_GENERAL_WHILE_LOAD_PROFILE _OR_GET_PROFILES	-1003	Function could not be executed as either the loading of profiles or the profile transfer is active
ERROR_GENERAL_WHILE_GET_PROFILES	-1004	Function could not be executed as the profile transfer is active
ERROR_GENERAL_GET_SET_ADDRESS	-1005	The address could not be read or written. Possibly a too old firmware is used
ERROR_GENERAL_POINTER_MISSING	-1006	A required pointer is set NULL
ERROR_GENERAL_SECOND_CONNECTION _TO_LLT	-1008	A second instance is connected to this scanCONTROL Ethernet port. Please close the second instance

8.2   SDK example overview

The sample programs in the project directory are intended as examples for the integration of the scanCONTROL in own projects. They are available as executable projects with the complete source code.

Name	Description
GetProfilesCallback	Transfer of profiles to the linLLT and pick up of profiles with a callback
GetProfilesPoll	Transfer of profiles to the linLLT and pick up of profiles via polling mode
VideoMode	Transfer and save matrix image

PartialProfile	Transfer of partial profiles
ContainerMode	Transfer and save profile containers respectively grey scale maps
PartialProfiles_MultiLLT	Usage of more than one scanCONTROL in one program with partial profile configuration
LLTPeakFilter	Set peak filter, free measuring field and dynamic, encoder defined measuring field
Calibration	Set sensor calibration (angle, offset)

8.3   Limitations

Scanner functionality not available in the linLLT in comparison to the Win SDK (LLT.dll):

-   SHOT transfer
-   Load and Save profiles
-   CMM triggering

On certain embedded boards (e.g. Raspberry Pi) only the partial profile mode can be used, due to interface limitations.

8.4   Supporting documentation

[1] Operation Manual PartB 2600: Interface Specification for scanCONTROL 2600 Device Family; Ethernet and Serial Port; Supplement B to the scanCONTROL 2600 Manual; MICRO-EPSILON Optronic GmbH;

[2] Operation Manual PartB 2700: Interface Specification for scanCONTROL 2700 Device Family; Firewire (IEEE 1394) Bus, Ethernet and Serial Port; Supplement B to the scanCONTROL 2700 Manual; MICRO-EPSILON Optronic GmbH; Revision:

[3] Operation Manual PartB 2900: Interface Specification for scanCONTROL 2900 Device Family; Ethernet and Serial Port; Supplement B to the scanCONTROL 2900 Manual; MICRO-EPSILON Optronic GmbH;

[4] scanCONTROL 2600 Quick Reference; Brief Introduction to scanCONTROL 2600 Device Family; MICRO-EPSILON Optronic GmbH;

[5] scanCONTROL 2700 Quick Reference; Brief Introduction to scanCONTROL 2700 Device Family; MICRO-EPSILON Optronic GmbH;

[6] scanCONTROL 2900 Quick Reference; Brief Introduction to scanCONTROL 2900 Device Family; MICRO-EPSILON Optronic GmbH;

[7] aravis; https://github.com/AravisProject/aravis/tree/ARAVIS_0_5_9; Date: 2017/06/23

[8] Operation Manual PartB 3000: Interface Specification for scanCONTROL 3000 Device Family; Ethernet and Serial Port; Supplement B to the scanCONTROL 3000 Manual; MICRO-EPSILON Optronic GmbH;

[9] scanCONTROL 3000 Quick Reference; Brief Introduction to scanCONTROL 3000 Device Family;

MICRO-EPSILON Optronic GmbH;

[10] Operation Manual PartB 2500: Interface Specification for scanCONTROL 2500 Device
Family; Ethernet and Serial Port; Supplement B to the scanCONTROL 2500 Manual; MICRO-
EPSILON Optronic GmbH;

[11] scanCONTROL 2500 Quick Reference; Brief Introduction to scanCONTROL 2500 Device Family;
MICRO-EPSILON Optronic GmbH;