Interface documentation LLT.dll

C++

Document version:	1.3
C++ SDK Version	4.0
LLT.dll version:	3.9.0.2012
Date:	21.10.2019

I. About this document

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

To get a general overview, a general introduction to the LLT.dll 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

Version	Date	Author	State
0.1	19.01.2016	DRa	Initial draft
1.0	28.03.2017	DRa, UEi	Reviewed version
1.1	07.02.2018	DRa	Updated
1.2a	28.02.2019	Uei	Initial integration sC30xx
1.2	14.06.2019	Uei	Integration of sC30xx completed
1.3	21.10.2019	THl	Integration of sC25xx

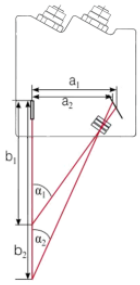
III. Content

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:

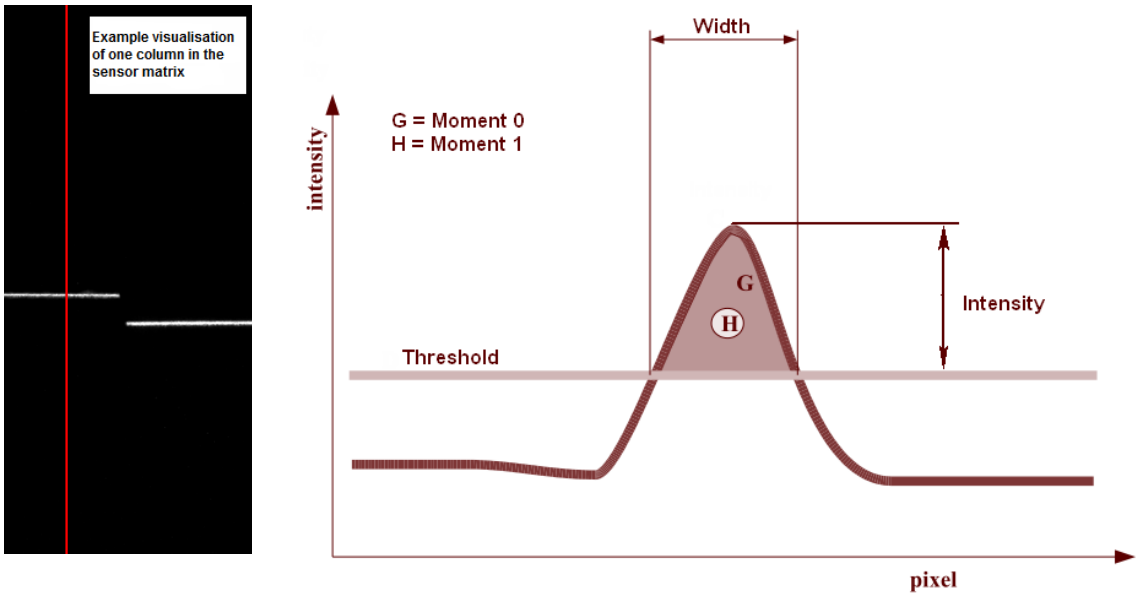


Fig. 1: How to get a measuring value from a reflection (evaluation of one sensor column)

- Distance: 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.

- Position: 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.

- Intensity: 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.

- Reflection width: 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.

- Moment 0: 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.

- Moment 1: 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.

- Threshold: 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).

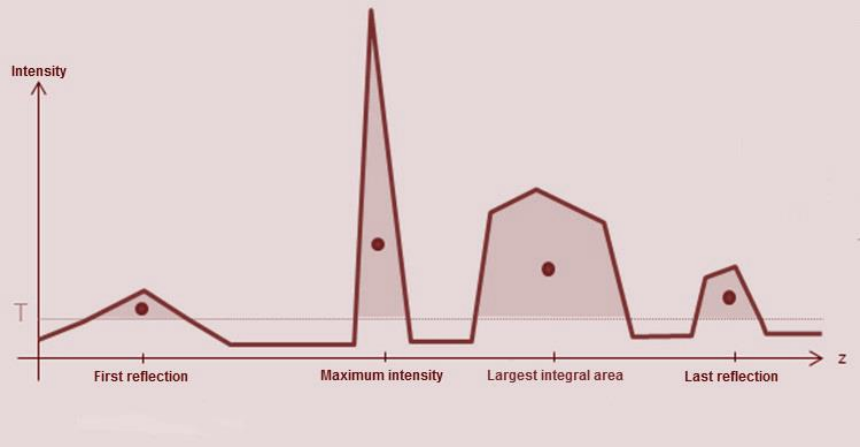


Fig. 2: Detected reflections

1.2 LLT.dll

The LLT.dll is a *Dynamic Link Library* (DLL) for the simple integration of scanCONTROL sensors into own applications. It provides an abstraction level above the direct responses of the scanCONTROL by Ethernet or the serial interface. The focus of the design of the DLL has been set on the simplicity of the interface and high performance.

The functionality includes the complete parametrization and control of the sensor, the transmission, saving and converting of measurement data in all different transmission modes. Additionally the connection monitoring between PC and scanCONTROL can be done via the DLL. Furthermore the integration of several sensors into one application is possible.

To make the DLL compatible with as many different application development systems and compilers as possible, the interface has been realized with pure C functions of the *cdecl* and *stdcall* call convention. Thus the DLL may be used under C, Delphi or other programming languages which provide compatibility to the used data types or if the implementation takes care of the differences (e.g. enums in Delphi). For C++ applications an additional wrapper exists, which maps the C functions to methods of an interface class. C# applications can be realized by wrapping the DLL with P/Invoke.

This documentation shows the DLL integration with the provided C++ interface class. The integration in C or another programming language may be derived from this documentation. For the implementation with C# another document is available, which comes with the C# wrapper.

1.3 Loading the DLL in C++

There are two different possibilities for loading a DLL. It may be loaded directly while the application is starting (static) or later if required (dynamic). The dynamic loading is more favorable, particularly because of the better error processing possibilities.

1.3.1 Static

For using the DLL statically in a C or C++ project, the corresponding *.lib-file is compiled into the project with the definitions provided by the DLL functions. Thereby it is important to use the .lib-file compatible to the used DLL-version. In the header files C_InterfaceLLT_2.h and S_InterfaceLLT_2.h the corresponding function declarations are to be found. In this context the prefix *s_* stands for *stdcall* and *c_* for *cdecl*.

For other programming languages the import function for the DLL may be generated with the C_InterfaceLLT_2.h or the S_InterfaceLLT_2.h.

In case of a new DLL-version the project has to be recompiled as the entry points in the DLL may change.

1.3.2 Dynamic

For the loading of the LLT.dll and the import of its functions in C++-projects the two classes *CInterfaceLLT* and *CDllLoader* are provided.

The DllLoader is responsible for the DLL handling (loading and respond of the function pointer). In the interface class (CInterfaceLLT) all functions exported by the LLT.dll are defined as function pointer. Therefore they simply can be called as methods of the interface class.

The outstanding advantage of this interface class is the dynamic call of the DLL functions. This means a new DLL version can be inserted without recompiling the project. Additionally it may be polled, if the requested function is available in the DLL. But this is only necessary for the functions added after the first release.
If new functions of the LLT.dll are to be used, the project has to be recompiled with the current interface class.

The constructor of the interface class allows passing the name of the DLL to be loaded (with corresponding path) and a pointer to a bool-variable, which signalizes loading errors.

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 30xx series provide a high resolution video mode that requires a bandwidth of 445 Mbps (25 Hz * 1088 * 2048 * 8 bit) and 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:

<table>
<tr><td>0..7</td><td>8..15</td><td>16..23</td><td>24..31</td></tr>
<tr><td>Res. (2 bit)</td><td>Reflection width (10 bit)</td><td>Max. Intensity (10 bit)</td><td>Threshold (10 bit)</td></tr>
<tr><td colspan="2">Position (16 bit)</td><td colspan="2">Distance (16 bit)</td></tr>
<tr><td colspan="4">Moment 0 (32 bit)</td></tr>
<tr><td colspan="4">Moment 1 (32 bit)</td></tr>
</table>

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.

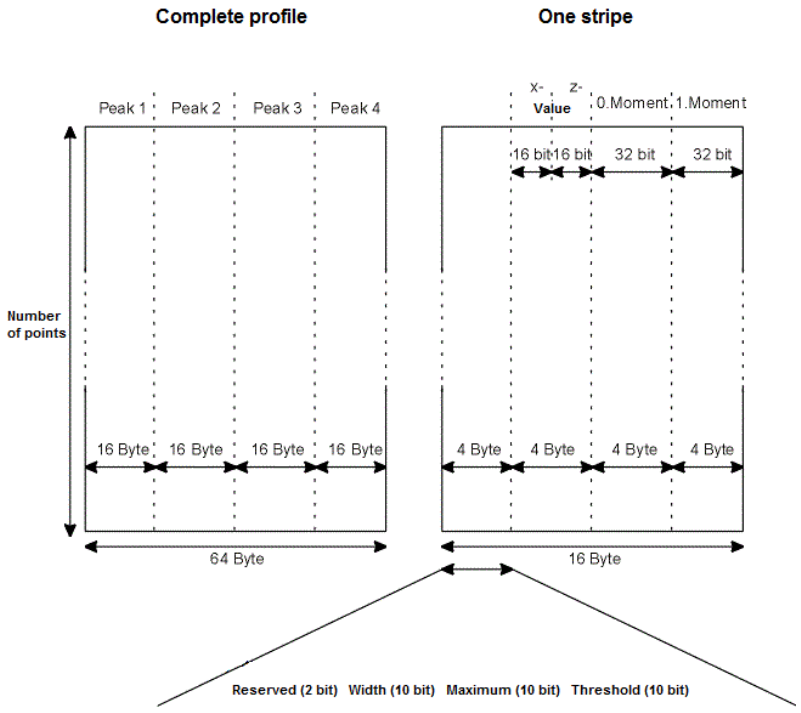


Fig. 4: Data arrangement single profile transmission

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:

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

(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

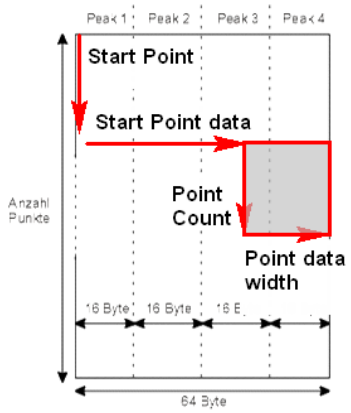


Fig. 5: Illustration Partial Profile

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.

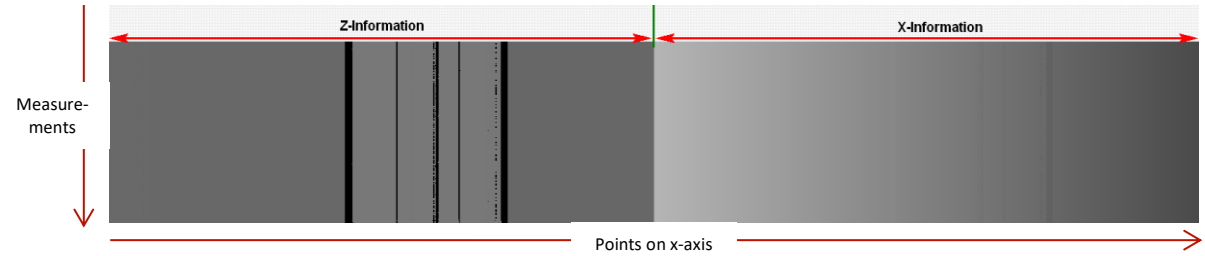


Fig. 6: Example image transposed container mode

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 5.1 the sensor has to be connected prior of code execution. In some examples, registers are set (*SetFeature(…)*) – note that register addresses are mapped onto variables 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 8.3).

6.1 Connect to the sensor

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

```
std::vector<unsigned int> Interfaces(5);
static CInterfaceLLT* pLLT = nullptr;
bool LoadError;

// Create device handle for an ethernet scanner
pLLT = new CInterfaceLLT("LLT.dll", &LoadError);

// Assign interface
pLLT->CreateLLTDevice(INTF_TYPE_ETHERNET);

// Search for scanners at the interface
pLLT->GetDeviceInterfacesFast(&Interfaces[0], (unsigned int)Interfaces.size());

// Set device handle to first detected scanner
pLLT->SetDeviceInterface(Interfaces[0], 0);

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

See API: CreateLLTDevice(), GetDeviceInterfaces(), GetDeviceInterfacesFast(), 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:

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

```
uint ExposureTime       = 1005;
uint IdleTime           = 8995;

// Set exposure 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. Please take a look at the SDK examples (sc30xx_HighSpeed).

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:

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

```
uint ExposureTime       = 100;
uint IdleTime           = 900;

// Set exposure 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.

```
unsigned int Resolution = resolutions[0];
TScannerType scanCONTROLType;

[…] // Connect

// 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(&scanCONTROLType);

// Lost profiles counter
unsigned int LostProfiles = 0;

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

```
// 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
pLLT->ConvertProfile2Values(&ProfileBuffer[0], Resolution,
    PROFILE, scanCONTROLType, 0, 1, NULL, NULL, NULL, ValueX, ValueZ, NULL, NULL);

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

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.

```
unsigned int Resolution;
unsigned int ProfileBufferSize;
TScannerType scanCONTROLType;

// Define callback handle
HANDLE hProfileEvent = CreateEvent(NULL, true, false, "ProfileEvent");

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

// Register callback
pLLT->RegisterCallback(STD_CALL, (void*)NewProfile, pLLT)

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

// Wait for event with timeout 1 second
if(WaitForSingleObject(hProfileEvent, 1000) != WAIT_OBJECT_0)
{
    // Handling timeout
}

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

// Callback function (copies one profile into the buffer and sets the event afterwards)
void __stdcall NewProfile(const unsigned char* pucData, unsigned int uiSize,
                                                              void* pUserData)
{
   memcpy(&ProfileBuffer[0], pucData, uiSize);
   SetEvent(hProfileEvent);
}
```

See API: RegisterCallback(), TransferProfiles()

6.6 Set profile filter

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

```
// 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.

```
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(), GetFeature(), 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.

```
// 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(), TriggerProfile(), FEATURE_FUNCTION_TRIGGER
See Operation Manual Part B: *OpManPartB.html#trigger*

Please take a look at the SDK examples (TriggerProfile).

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*

Requires: Firmware v46 or newer.
Please take a look at the SDK examples (TriggerContainer).

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 peak 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->SetFeature(FEATURE_FUNCTION_PEAKFILTER_WIDTH,
                                        (min_width << 16) + max_width);
pLLT->SetFeature(FEATURE_FUNCTION_PEAKFILTER_HEIGHT,
                                        (min_intensity << 16) + max_intensity);

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

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

Note for scanCONTROL 27xx and scanCONTROL 26xx/29xx (Firmware version < v43):
The peak filter has to be set via the EXTRA_PARAMETER register. Please take a look at the SDK examples (advanced/LLTPeakFilter).

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;
```

Please take a look at the SDK examples (SetROIs).

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.

```
int toggle = 0;

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

WriteCommand(0, 0); // Reset
WriteCommand(0, 0); // Initialization
WriteCommand(2, 8); // Navigate in register
WriteValue2Register(row_start);
WriteValue2Register(row_size);
WriteValue2Register(col_start);
WriteValue2Register(col_size);
WriteCommand(0, 0); // Stop writing process

// Write command for seq. register
static void WriteCommand(unsigned int command, unsigned int data)
{
   pLLT->SetFeature(FEATURE_FUNCTION_EXTRA_PARAMETER, (unsigned int)(command << 9)
                                                      + (toggle << 8) + data);

   if (toggle == 1) : toggle = 0 ? toggle = 1;
}

// Write value to register position
static void WriteValue2Register(unsigned short value)
{
   WriteCommand(1, (unsigned int)(value/256));
   WriteCommand(1, (unsigned int)(value%256));
}
```

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

See chapter 6.12 how to compute the correct matrix region values. Please take a look at the SDK examples (SetROIs).

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.

```
// Activate region of interest 1
pLLT->SetFeature(FEATURE_FUNCTION_ROI1_PRESET, 0x82000800);

// 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: SetFeature(), FEATURE_FUNCTION_ROI1_PRESET, FEATURE_FUNCTION_EXTRA_PARAMETER
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. Please take a look at the SDK examples (SetROIs).

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. The functions *SetCustomCalibration()* and *ResetCustomCalibration()* are not integrated into the DLL, but implemented in the example program for calibration.

```
// Sensor offset and scaling (depending on sensor type)
double offset = 95.0;
double scaling = 0.002;

// Rotation center and angle
double center_x = -9; // mm
double center_z = 86.7; // mm
double angle = -45; // °

// Shift rotation center
double shift_x = 0; // mm
double shift_z = 0; // mm

// Set calibration
SetCustomCalibration(center_x, center_z, angle, shift_x, shift_z, offset, scaling);

// Reset calibration
ResetCustomCalibration();
```

See example program: Calibration

Please take a look at the SDK examples (Calibration).

6.16 Use CMM trigger

With CMM triggering the scanCONTROL sensor can communicate the exact point of time at which the profile was generated. This happens via the sensor's RS422 interface.

Configuration of the CMM trigger signal can be done via the following parameters:

- Polarity: Polarity of trigger signal
- Divisor: Trigger divisor (0 deactivates CMM triggering)
- Mark-Space ratio: Mark space ratio (duty cycle) of the signal
- Skew Correction: Correction of the trigger signal offset (in 0.5 µs steps; must be smaller than half of the sensor cycle (1/profile frequency))

```
unsigned int Incounter = 0, CmmCount = 0;
int  CmmTrigger = 0, CmmActive = 0;

std::vector<unsigned char>Timestamp(16);

// Set RS422 interface for 26xx/29xx to CMM triggering
unsigned int Interface = MULTI_RS422_CMM;
pLLT->SetFeature(FEATURE_FUNCTION_DIGITAL_IO, Interface);

// Set mark space ratio
pLLT->SetFeature(FEATURE_FUNCTION_CMM_TRIGGER, 0x00000401);
// Set skew correction
pLLT->SetFeature(FEATURE_FUNCTION_CMM_TRIGGER, 0x00000801);
// Set output port
pLLT->SetFeature(FEATURE_FUNCTION_CMM_TRIGGER, 0x00000c00);
// Set trigger divisor
pLLT->SetFeature(FEATURE_FUNCTION_CMM_TRIGGER, 0x00000001);

[…] // Transmission

// Read CMM timestamp
Timestamp2CmmTriggerAndInCounter(Timestamp, &Incounter, &CmmTrigger,
                                                    &CmmActive, &CmmCount);
```

See API: SetFeature(), Timestamp2CmmAndInCounter(), FEATURE_FUNCTION_DIGITAL_IO, FEATURE_FUNCTION_CMM_TRIGGER
See Operation Manual Part B: *OpManPartB.html#cmmtrigger, OpManPartB.html#ioconfig*

Before activating the trigger by setting the divisor, the CMM trigger should be configured completely. After activation the CMM trigger must not be reconfigured. Additionally the CMM trigger is not saved in the user modes, thus it has to be configured by the connected software.

Note: only supported by the scanCONTROL 26xx/27xx/29xx/30xx series.

6.17 Save profile data

This example shows how to save profile data for subsequent offline evaluation. The profiles are saved as .AVI file.

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

// Starte saving data as .avi file
pLLT->SaveProfiles(AviFilename, AVI);

// Wait for the period of time the profiles should be saved (1 s)
Sleep(1000);

// Stop saving procedure
pLLT->SaveProfiles(NULL, AVI);

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

See API: TransferProfiles(), SaveProfiles()

6.18 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.

```
#include <math.h>

[…] // Setup and declaration of ContainerBuffer

// Set container size
unsigned int ProfileCount = 100; // Number of profiles in one image/container
unsigned int LostProfiles = 0;

// Flags für gerade verwendete Auflösung berechnen
double TempLog = 1.0 / log(2.0);
unsigned int ResBits = (unsigned int)floor((log((double)Resolution)*TempLog)+0.5);
// Set rearrangement parameters to extract z data (without timestamp) with the
// set resolution; the necessary container width is automatically set
pLLT->SetFeature(FEATURE_FUNCTION_PROFILE_REARRANGEMENT,
    (CONTAINER_DATA_Z | CONTAINER_STRIPE_1 | CONTAINER_DATA_LSBF | ResBits << 12));

// Set container size
pLLT->SetProfileContainerSize(0, ProfileCount);

// Allocate buffer (z value has 2 bytes)
std::vector<unsigned char>ProfileBuffer(Resolution * 2 * ProfileCount);

// Start profile transmission with container mode config
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, true);

// Poll a profile from receiving buffer
// Hint: If no new container 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(&ContainerBuffer[0], (unsigned int) ContainerBuffer.size(),
                                                    CONTAINER, &LostProfiles));

// Stop profile transmission with container mode config
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, false);
```

See API: SetFeature(), SetProfileContainerSize(), TransferProfiles(), GetActualProfile(), FEATURE_FUNCTION_PROFILE_REARRANGEMENT
See Operation Manual Part B: *OpManPartB.html#profilerearrangement*

6.19 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 to define partial profile
TPartialProfile PartialProfile;

[…] // Init

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

// Allocate profile buffer
std::vector<unsigned char>ProfileBuffer(PartialProfile.nPointCount *
                                                    PartialProfile.nPointDataWidth);

// Write partial profile settings
pLLT->SetPartialProfile(PartialProfile);

[…] // Normal profile transmission

// Convert buffer to real distance and position in mm
pLLT->ConvertPartProfile2Values(&ProfileBuffer[0], ProfileBuffer.size(),
     &PartialProfile, scanCONTROLType, 0, 1, NULL, NULL, NULL, &ValueX[0],
                                                          &ValueZ[0], NULL, NULL);
```

See API: SetPartialProfile(), TransferProfiles(), GetActualProfile(), 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.

6.20 Use several sensors in one application

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

```
std::vector<unsigned int> Interfaces(5);
static CInterfaceLLT* pLLT = nullptr;
static CInterfaceLLT* pLLT2 = nullptr;
HANDLE hProfileEvent = CreateEvent(NULL, true, false, "ProfileEvent");
bool LoadError;

// Create handle for every sensor
pLLT = new CInterfaceLLT("LLT.dll", &LoadError);
pLLT2 = new CInterfaceLLT("LLT.dll", &LoadError);

// Search available interfaces
pLLT->GetDeviceInterfaces(Interfaces, Interfaces.GetLength(0));

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

```
// Connect both sensors
pLLT->Connect();
pLLT2->Connect();

[…]

// Registrierung Callback Scanner 1
pLLT->RegisterCallback(STD_CALL, (void*)NewProfile, pLLT);

// Registrierung Callback Scanner 2
pLLT2->RegisterCallback(STD_CALL, (void*)NewProfile, pLLT2);

[…] // Start of transmission equivalent

// Callback function with separation of sensor data
void __stdcall NewProfile(const unsigned char* pucData, unsigned int uiSize,
                                                                    void* pUserData)
{
   if (pUserData == pLLT)
   {
      // Data sensor 1
   }

   if (pUserData == pLLT2)
   {
      // Data sensor 2
   }
}
```

See API: CreateLLTDevice(), GetDeviceInterfacesFast(), SetDeviceInterface(), Connect(), RegisterCallback(),

Please take a look at the SDK examples (MultiLLTs).

6.21 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 Error Message (depending on application)
pLLT->RegisterErrorMsg(/*UINT */Msg, /*HWND */hWnd, /*WPARAM*/ WParam);

BOOL CMessageWnd::OnWndMsg(unsigned int message, WPARAM wParam,
                                          LPARAM lParam, LRESULT* pResult)
{
  if (message == WM_USER)
  {
    if (lParam == ERROR_CONNECTIONLOST)
      // Error handling
  }

  return CWnd::OnWndMsg(message, wParam, lParam, pResult);
}
```

See API: RegisterErrorMsg()

6.22 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;

// Prior to temperature reading 0x86000000 has to be written to the register
pLLT->SetFeature(FEATURE_FUNCTION_TEMPERATURE, TEMP_PREPARE_VALUE);

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

See API: SetFeature(), FEATURE_FUNCTION_TEMPERATURE
See : *OpManPartB.html#temperature*

6.23 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):

```
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.

7.1 Instance functions

- **CreateLLTDevice ()**

```
unsigned int
CInterfaceLLT::CreateLLTDevice(iInterfaceType);
```

Set and return a device handle (sensor instance) in the DLL for scanCONTROL communication – depending on the connection interface.

Parameter

CInterfaceLLT LLT class
iInterfaceType Type of interface (InterfaceType)

Return value

New device handle (0x0 or 0xFFFFFFFF → Error: No device handle created)
General error codes

- **GetInterfaceType ()**

```
int
CInterfaceLLT::GetInterfaceType();
```

Query the currently assigned interface type for a sensor instance.

Parameter

CInterfaceLLT LLT class

Return value

Value InterfaceType (0x0 or 0xFFFFFFFF → Error)
General error codes

- **InterfaceType**

Available interface types:

InterfaceType	Value	Description
INTF_TYPE_UNKNOWN	0	Error value if GetInterfaceType () fails; not to be set with CreateLLTDevice()
INTF_TYPE_SERIAL	1	Connection via serial interface
INTF_TYPE_FIREWIRE	2	Connection via Firewire bus (deprecated)

INTF_TYPE_ETHERNET	3	Connection via Ethernet interface

- **DelDevice ()**

```
int
CInterfaceLLT::DelDevice();
```

Delete sensor instance prior to unloading of the DLL. All currently set parameters are preserved on the sensor, except the driver parameters *Packet size*, *Buffer count* and *Profile config*.

Parameter

CInterfaceLLT LLT class

Return value

General return values

7.2 Interface functions

- **GetDeviceInterfaces () / GetDeviceInterfacesFast ()**

```
int
CInterfaceLLT::GetDeviceInterfaces(unsigned int[] pInterfaces, int nSize);

int
CInterfaceLLT::GetDeviceInterfacesFast(unsigned int[] pInterfaces, int nSize);
```

Query the available scanCONTROL device interfaces at the PCs interface cards. The returned device interfaces are IP addresses of the connected sensors. *GetDeviceInterfacesFast()* (Ethernet interface only) is significantly faster in small Ethernet networks.

Parameter

CInterfaceLLT LLT class
pInterfaces Array for available interfaces (IP; Node ID)
nSize Size of array

Return value

Number of device interfaces found
General error codes
Specific return values:

ERROR_GETDEVINTERFACES_WIN_NOT_SUPPORTED	-250	Function is only available for Windows 2000 or higher
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

- **SetDeviceInterface ()**

```
int
CInterfaceLLT::SetDeviceInterface(unsigned int nInterface, int nAdditional);
```

Assign a scanCONTROL device interface to a sensor instance in the DLL. The additional parameter may optionally include which host IP address to use. This is useful if more than one network interface card is available.

Parameter

CInterfaceLLT	LLT class
nInterface	Interface of scanCONTROL to be connected
nAdditional	IP address of host (optional)

Return value

General return values

Specific return value:

ERROR_GETDEVINTERFACES_CONNECTED	-252	The scanCONTROL is connected, call *Disconnect()*

- **SetDiscoveryBroadcastTarget ()**

```
int
CInterfaceLLT::SetDiscoveryBroadcastTarget(unsigned int nNetworkAddress,
                                           unsigned int nSubnetMask);
```

Set the local sender address that is used for discovery packets in context of the Ethernet broadcast. Useful if more than one network interface card is available.

Parameter

CInterfaceLLT	LLT class
nNetworkAddress	Sender IP address
nSubnetMask	Sender subnet mask

Return value

General return values

7.3 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()*).

Parameter

CInterfaceLLT	LLT class

Return value

General return values

Specific return values:

ERROR_CONNECT_LLT_COUNT	-300	There is no scanCONTROL connected to the computer or the driver is not installed correctly.
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_CONNECT_LLT_NUMBER_ALREADY_USED	-303	The requested scanCONTROL is already used by another instance -> choose another scanCONTROL with *SetDeviceInterface()*
ERROR_CONNECT_SERIAL_CONNECTION	-304	The scanCONTROL by serial interface could not be connected -> choose another scanCONTROL with *SetDeviceInterface()*

- **Disconnect ()**

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

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

Parameter

CInterfaceLLT LLT class

Return value

General return values

7.4 Identification functions

- **GetDeviceName ()**

```
int
CInterfaceLLT::GetDeviceName(char * DevName, unsigned int DevNameSize,
                             char * VenName, unsigned int VenNameSize);
```

Query device name and vendor name of scanCONTROL sensor.

Parameter

CInterfaceLLT	LLT class
sbDevName	Device name array
sbDevNameSize	Size of DevName buffer
sbVenName	Vendor name array
sbVenNameSize	Size of VenName buffer

Return value

General return values

Specific return values:

ERROR_GETDEVICENAME_SIZE_TOO_LOW	-1	The size of a buffer is too small
ERROR_GETDEVICENAME_NO_BUFFER	-2	No buffer has been passed

- **GetLLTVersions ()**

```
int
CInterfaceLLT::GetLLTVersions(unsigned int * uiDSP, unsigned int * uiFPGA1,
                                                    unsigned int * uiFPGA2);
```

Query Firmware version of scanCONTROL sensor.

Parameter

CInterfaceLLT	LLT class
uiDSP	Firmware version DSP
uiFPGA1	Firmware version FPGA1
uiFPGA2	Firmware version FPGA2

Return value

General return values

- **GetLLTType ()**

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

Query measuring range and type of scanCONTROL sensor.

Parameter

CInterfaceLLT	LLT class
ScannerType	Scanner type and measuring range

Return value

General return values

- **ScannerType**

TScannerType	Value	scanCONTROL type	Measuring range
StandardType	-1	-	-
scanCONTROL27xx_25	1000	27xx	25 mm
scanCONTROL27xx_100	1001	27xx	100 mm
scanCONTROL27xx_50	1002	27xx	50 mm
scanCONTROL26xx_25	2000	26xx	25 mm

scanCONTROL26xx_50	2002	26xx	50 mm
scanCONTROL26xx_100	2001	26xx	100 mm
scanCONTROL29xx_25	3000	29xx	25 mm
scanCONTROL29xx_50	3002	29xx	50 mm
scanCONTROL29xx_100	3001	29xx	100 mm
scanCONTROL29xx_10	3003	29xx	10 mm
scanCONTROL30xx_25	4000	30xx	25mm
scanCONTROL30xx_50	4001	30xx	50mm
scanCONTROL25xx_25	5000	25xx	25mm
scanCONTROL25xx_50	5002	25xx	50mm
scanCONTROL25xx_100	5001	25xx	100mm

7.5 Feature functions

7.5.1 Set-/get functions

- **GetFeature ()**

```
int
CInterfaceLLT::GetFeature(unsigned int Function, unsigned int * pValue);
```

Query currently set parameter value / Check availability of a feature according to the table in chapter 7.5.2.

Parameter

CInterfaceLLT	LLT class
Function	Register address of function (FEATURE or INQUIRY)
pValue	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 Function, unsigned int Value);
```

Set feature parameter.

Parameter

CInterfaceLLT	LLT class
Function	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

7.5.2 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	0xfffff0f00524	0xfffff0f00824	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 $0\ |_{\mathrm{dec}}$ corresponds with laser off, $1\ |_{\mathrm{dec}}$ with reduced and $2\ |_{\mathrm{dec}}$ 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.
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)

7.6 Special feature functions

7.6.1 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

7.6.2 Profile configuration

- **GetProfileConfig ()**

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

Query of the current profile configuration.

Parameter

CInterfaceLLT	LLT class
pValue	Profile configuration read from sensor

Return value

General return values

- **SetProfileConfig ()**

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

Profile configuration to be set for data transmission.

Parameter

CInterfaceLLT	LLT class
Value	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
PURE_PROFILE	2	Reduced profile data of one stripe (only position and distance values)
QUARTER_PROFILE	3	Profile data of one stripe
PARTIAL_PROFILE	5	Partial profile which has been restricted by *SetPartialProfile()*
CONTAINER	1	Container data
VIDEO_IMAGE	1	Video image of the scanCONTROL

7.6.3 Profile resolution / Points per profile

- **GetResolution ()**

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

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

Parameter

CInterfaceLLT LLT class
pValue 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

CInterfaceLLT	LLT class
Value	Profile resolution to be set

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[] pValue, int nSize);
```

Query of available profile resolutions.

Parameter

CInterfaceLLT	LLT class
pValue	Array with available profile resolutions
nSize	Size of array

Return value

Number of available resolutions

General return values

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

7.6.4 Container size

- **GetProfileContainerSize ()**

```
int
CInterfaceLLT::GetProfileContainerSize(unsigned int * pWidth,
                                       unsigned int * pHeight);
```

Query of currently set container size.

Parameter

CInterfaceLLT	LLT class
pWidth	Container width queried
pHeight	Container height queried

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_PROFILE_REARRANGEMENT)* 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* or *OpManPartB.html#profilerearrangement)* 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. Additionally the error value GENERAL_FUNCTION_CONTAINER_MODE_HEIGHT_CHANGED will be returned for indicating the changes.

Parameter

CInterfaceLLT	LLT class
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 * pMaxWidth,
                                          unsigned int * pMaxHeight);
```

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

Parameter

CInterfaceLLT	LLT class
pMaxWidth	Maximum container width
pMaxHeight	Maximum container height

Return value

General return values

7.6.5 Main reflection

- **GetMainReflection ()**

```
int
CInterfaceLLT::GetMainReflection(unsigned int * pValue);
```

Query the currently set main reflection, to be extracted by the profile configurations *PURE_PROFILE* or *QUARTER_PROFILE*.

Parameter

CInterfaceLLT	LLT class
pValue	Value of main reflection queried

Return value

General return values

- **SetMainReflection ()**

```
int
CInterfaceLLT::SetMainReflection(unsigned int Value);
```

Set the main reflection (stripe), from which the profile data is extracted if *PURE_PROFILE* or *QUARTER_PROFILE* is used. The index of the stripe to be used ranges from 0 for the 1st stripe to 3 for the 4th stripe.

Parameter

CInterfaceLLT	LLT class
Value	Main reflection to be set

Return value

General return values

Specific return value:

ERROR_SETGETFUNCTIONS_REFLECTION_NUMBER_TOO_HIGH	-154	The index of the stripe to be put out is greater than 3

7.6.6 Number of buffers

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 * pValue);
```

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

Parameter

CInterfaceLLT	LLT class
Value	Buffer count queried

Return value

General return values

- **SetBufferCount ()**

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

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

Parameter

CInterfaceLLT	LLT class
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

7.6.7 Allocated buffer for profile polling

- **GetHoldBuffersForPolling ()**

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

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

Parameter

CInterfaceLLT	LLT class
puiHoldBuffersForPolling	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

CInterfaceLLT	LLT class
uiHoldBuffersForPolling	Buffer count set

Return value

General return values

Specific return value:

ERROR_SETGETFUNCTIONS_WRONG_BUFFER_COUNT	-150	Bufer count is not in the range of >=2 and <= 200

7.6.8 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 * pValue);
```

Query the active packet size of the Ethernet streaming packets.

Parameter

CInterfaceLLT	LLT class
pValue	Packet size queried

Return value

General return values

- **SetPacketSize ()**

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

Set the active packet size of the Ethernet streaming packets. This packet size has to be between the minimum and maximum packet size.

Parameter

CInterfaceLLT	LLT class
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 long * pMinPacketSize,
                                   unsigned long * pMaxPacketSize);
```

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

Parameter

CInterfaceLLT	LLT class
pMinPacketSize	Minimal packet size settable
pMaxPacketSize	Maximal packet size settable

Return value

General return values

7.6.9 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 * pActualUserMode,
                                 unsigned int * pUserModeCount);
```

Query the last loaded user mode / parameter queue. The scanCONTROL 25xx/26xx/27xx/29xx/30xx series support 16 user modes.

Parameter

CInterfaceLLT	LLT class
pActualUserMode	Currently loaded user mode
pUserModeCount	Available user modes

Return value

General return values

- **ReadWriteUserModes ()**

```
int
CInterfaceLLT::ReadWriteUserModes(int nWrite, unsigned int nUserMode);
```

Loading or storing a user mode. If nWrite is 0, the user mode specified by nUserMode is loaded; otherwise the current settings are stored to this user mode. After loading of a user mode the sensor must be reconnected.

Parameter

CInterfaceLLT	LLT class
nWrite	Loading (0) or saving (other than 0) of an user mode
nUserMode	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)

7.6.10 Timeout for communication supervision to the sensor

Setting and reading the heartbeat timeout in milliseconds to monitor the connection between LLT.dll 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 * pValue);
```

Query the set heartbeat timeout.

Parameter

CInterfaceLLT	LLT class
pValue	Heartbeat timeout queried

Return value

General return values

- **SetEthernetHeartbeatTimeout ()**

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

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

7.6.11 Set file size for saving data

- **GetMaxFileSize ()**

```
int
CInterfaceLLT::GetMaxFileSize(unsigned int * pValue);
```

Query the maximum file size for saving profile data in byte.

Parameter

CInterfaceLLT	LLT class
pValue	Maximum file size queried

Return value

General return values

- **SetMaxFileSize ()**

```
int
CInterfaceLLT::SetMaxFileSize(unsigned int Value);
```

Set maximum file size for saving profile data in byte. If this size is reached the saving process is stopped.

Parameter

CInterfaceLLT	LLT class
Value	Maximum file size to be set

Return value

General return values

7.7 Register functions

7.7.1 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 pUserData 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 LLT.dll 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.

- **RegisterCallback ()**

```
int
CInterfaceLLT::RegisterCallback(TCallbackType tCallbackType,
                        void * tReceiveProfiles, unsigned int pUserData);
```

Register the callback which is called every time a profile is received.

Parameter

CInterfaceLLT	LLT class
tCallbackType	Call convention (0: stdcall; 1: c_decl)
tReceiveProfiles	Callback function to be registered
pUserData	Userdata for distinguishing sensors

Return value

General return values

- **CallbackType**

Callback Type	Value	Description
STD_CALL	0	The callback is working with stdcall (TNewProfile_s)
C_DECL	1	The callback is working with cdecl (TNewProfile_c)

7.7.2 Register error message for error handling

- **RegisterErrorMsg ()**

```
int
CInterfaceLLT::RegisterErrorMsg(UINT Msg, HWND hWnd, WPARAM WParam);
```

Register an error message.

Parameter

CInterfaceLLT	LLT class
Msg	Message ID
hWnd	Handle (e.g. Window)
WParam	Identification parameter

Return value

General return values

- **WParam (error ID)**

Available error parameter:

Constant of return value	Value	Description
ERROR_SERIAL_COMM	1	Error during the serial data transfer. Possibly the profile frequency is too high.
ERROR_SERIAL_LLT	7	ScanCONTROL could not interpret the commando or a parameter out of the validity area has been send.
ERROR_CONNECTIONLOST	10	The connection to the scanCONTROL has been interrupted (scanCONTROL has been switched off, reset or the Ethernet cable has been removed). Please send a *Disconnect()* for being able to reconnect. This message is only send at a connection by Ethernet
ERROR_STOPSAVING	100	The saving of the profiles is finished (maximum data size reached)

7.8 Profile transmission functions

7.8.1 Start/stop profile transmission

- **TransferProfiles ()**

```
int
CInterfaceLLT::TransferProfiles(TTransferProfileType TransferProfileType,
                                int nEnable);
```

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 or can be fetched via *GetActualProfile()*. If a transfer is terminated, the function waits automatically till the driver has returned all buffers.

Parameter

CInterfaceLLT	LLT class

TransferProfileType Profile transfer type
nEnable Start (1) or stop (0) transmission

Return value

Size of profile/container
General return values
Specific return value:

ERROR_PROFTRANS_PACKET_SIZE_TOO_HIGH	-107	Packet size is too high
ERROR_PROFTRANS_CREATE_BUFFERS	-108	Buffers could not be reserved → restart PC
ERROR_PROFTRANS_WRONG_PACKET_SIZE_FOR_CONTAINER	-109	Packet size does not match container settings, see *setPacketSize()*

- **TransferProfileType**

Available TransferProfileTypes:

Constant return value	Value	Description
NORMAL_TRANSFER	0	Activation of a continuous transfer of profiles
SHOT_TRANSFER	1	Activation of a demand-based transfer of profiles (the transfer is always activated by *MultiShot()*)
NORMAL_CONTAINER_MODE	2	Activation of a continuous transfer in the container mode
SHOT_CONTAINER_MODE	3	Activation of a demand-based transfer in the container mode (the transfer is always activated by *MultiShot()*)

7.8.2 Transmission of sensor matrix image / start/stop Video Mode

- **TransferVideoStream ()**

```
int
CInterfaceLLT::TransferVideoStream(TTransferVideoType videoType, int nEnable,
                                   unsigned int * pWidth, unsigned int * pHeight);
```

Start or stop transmission of video images of the sensor matrix (Video Mode). Maximum video frame rate is 25 pictures per second. Video pictures are fetched per *GetActualProfile()*. The Callback cannot be used for video image transmission.

Parameter

CInterfaceLLT LLT class
TTransferVideoType Profile transmission type
nEnable Start (1) or stop (0) transmission
pWidth Width of received image
pHeight Height of received image

Return value

General return values
Specific return values:

ERROR_PROFTRANS_PACKET_SIZE_TOO_HIGH	-107	The packet size is bigger than the available one -> adjust a smaller packet size by *SetPacketSize()*
ERROR_PROFTRANS_CREATE_BUFFERS	-108	The buffer for the driver could not be created duly -> possibly restart the PC

- **TransferVideoType**

Available TransferVideoTypes:

Constant return value	Value	Description
VIDEO_MODE_0	0	Low resolution image of matrix
VIDEO_MODE_1	1	Full size image of matrix

7.8.3 Transmission of a specified number of profiles / container

- **MultiShot ()**

```
int
CInterfaceLLT::MultiShot(unsigned int nCount);
```

Request the transmission of a specified number of profiles / containers. The amount of profiles / container is passed in the parameter nCount. You can fetch between 1 and 65535 profiles / container.
Note: MultiShot() does not trigger the sensor to deliver a profile or a container. It only passes the most recent profile(s)/container(s) in the buffer to the application so that GetActualProfile or the Callback deliver only the number of specified profile(s)/container(s) instead of a continuous sequence of profile(s)/container(s).

Parameter

CInterfaceLLT	LLT class
nCount	Number of requested profiles/container

Return value

General return values
Specific return values:

ERROR_PROFTRANS_SHOTS_NOT_ACTIVE	-100	The SHOT_TRANSFER mode or the SHOT_CONTAINER_MODE mode is not activated -> restart profile transfer
ERROR_PROFTRANS_SHOTS_COUNT_TOO_HIGH	-101	The number of requested profiles / container is bigger than 65535
ERROR_PROFTRANS_MULTIPLE_SHOTS_ACTIVE	-111	A MultiShot request is active -> call *MultiShot(0)* for abort

7.8.4 Transmission of profiles via serial interface

- **GetProfile ()**

```
int
CInterfaceLLT::GetProfile();
```

Profile transfer via serial interface. This function is only available for the serial interface.

Parameter

CInterfaceLLT LLT class

Return value

General return values

7.8.5 Fetch current profile / container / video image

- **GetActualProfile ()**

```
int
CInterfaceLLT::GetActualProfile(unsigned char[] pBuffer, int nBuffersize,
                TProfileConfig ProfileConfig, unsigned int * pLostProfiles);
```

Fetching the active profile/container/video image from the LLT.dll.

Parameter

CInterfaceLLT	LLT class
pBuffer	Transmission buffer
nBuffersize	Size of transmission buffer
ProfileConfig	Profile configuration of transmission
pLostProfiles	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

7.8.6 Convert profile data

- **ConvertProfile2Values ()**

```
int
CInterfaceLLT::ConvertProfile2Values(const unsigned char[] pProfile,
unsigned int nResolution, TProfileConfig ProfileConfig, ScannerType ScannerType,
      unsigned int nReflection, int nConvertToMM, unsigned short[] pWidth,
           unsigned short[] pMaximum, unsigned short[] pThreshold, double[] pX,
                 double[] pZ, unsigned int[] pM0, unsigned int[] pM1);
```

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

Parameter

CInterfaceLLT	LLT class
pProfile	Profile buffer
nResolution	Points per profile
ProfileConfig	Profile configuration of transmission
ScannerType	Scanner type
nReflection	Profile stripe to be evaluated
nConvertToMM	Convert x/z data to millimeters
pWidth	Array for reflection width
pMaximum	Array for maximum intensity
pThreshold	Array for threshold setting
pX	Array for position values
pZ	Array for distance values
pM0	Array for moment 0
pM1	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 ()**

```
int
CInterfaceLLT::ConvertPartProfile2Values(const unsigned char * pProfile,
    TPartialProfile * ProfileConfig, ScannerType ScannerType,
       unsigned int nReflection, int nConvertToMM, ushort[] pWidth,
         ushort[] pMaximum, ushort[] pThreshold, double[] pX, double[] pZ,
                                                    uint[] pM0, uint[] pM1);
```

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

CInterfaceLLT	LLT class
pProfile	Profile buffer
PartialProfile	Partial profile
ProfileConfig	Profile configuration of transmission
ScannerType	Scanner type

nReflection	Profile stripe to be evaluated
nConvertToMM	Convert x/z data to millimeters
pWidth	Array for reflection width
pMaximum	Array for maximum intensity
pThreshold	Array for threshold setting
pX	Array for position values
pZ	Array for distance values
pM0	Array for moment 0
pM1	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

- **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

7.9 Is functions

- **IsInterfaceType ()**

```
int
CInterfaceLLT::IsInterfaceType(int iInterfaceType);
```

Query of currently used interface.

Parameter

CInterfaceLLT	LLT class
iInterfaceType	Integer value of interface type

Return value

Specific return values:

INTF_TYPE_SERIAL	1	Connection via serial interface
INTF_TYPE_ETHERNET	3	Connection via Ethernet interface

- **IsTransferingProfiles()**

```
int
CInterfaceLLT::IsTransferingProfiles();
```

Query of transmission state, i.e. if the transmission is active or not.

Parameter

CInterfaceLLT LLT class

Return value

Specific return values:

IS_FUNC_YES	1	Interrogated state or connection is active
IS_FUNC_NO	0	Interrogated state or connection is not active

7.10 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 * pPartialProfile);
```

Query the current partial profile setting set on the scanCONTROL.

Parameter

CInterfaceLLT LLT class
pPartialProfile Reference to partial profile structure

Return value

General return values

- **SetPartialProfile ()**

```
int
CInterfaceLLT::SetPartialProfile(TPartialProfile pPartialProfile);
```

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 nUnitSize of the function *GetPartialProfileUnitSize()*.

Parameter

CInterfaceLLT	LLT class
pPartialProfile	Reference to partial profile structure to be set

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 * pUnitSizePoint,
                                         unsigned int * pUnitSizePointData);
```

This function returns the increments for adjusting the partial profile.

Parameter

CInterfaceLLT	LLT class
pUnitSizePoint	UnitSizePoint size
pUnitSizePointData	UnitSizePointData size

Return value

General return values

7.11 Timestamp extraction functions

- **Timestamp2TimeAndCount ()**

```
int
Timestamp2TimeAndCount(const unsigned char * pBuffer,
   double * dTimeShutterOpen, double * dTimeShutterClose,
                                          unsigned int * uiProfileCount);
```

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

pBuffer	Reference to timestamp bytes of profile buffer
dTimeShutterOpen	Timestamp shutter open
dTimeShutterClosed	Timestamp shutter closed
uiProfileCount	Profile count

Return value

General return values

- **Timestamp2CmmTriggerAndInCounter ()**

```
int
Timestamp2CmmTriggerAndInCounter(const unsigned char * pBuffer,
   unsigned int * pInCounter, int * pCmmTrigger, int * pCmmActive,
                                                  unsigned int * pCmmCount);
```

This function evaluates only the optional part of the timestamp of a profile. It returns the reading of the internal counter, the CmmTrigger and CmmActive flags as well as the CMM trigger counter.

Parameter

pBuffer	Reference to timestamp bytes of profile buffer
pInCounter	Internal counter
pCmmTrigger	Flag CMM trigger *active*
pCmmActive	*Flag CMM active*
pCmmCount	Trigger count

Return value

General return values

7.12 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.

7.12.1 Read and write Post-Processing parameters

- **ReadPostProcessingParameter ()**

```
int
CInterfaceLLT::ReadPostProcessingParameter(unsigned int * pParameter,
                                                    unsigned int nSize);
```

Read post processing parameters.

Parameter

CInterfaceLLT	LLT class
pParameter	Post processing parameter array
nSize	Size of post processing parameter

Return value

General return values

- **WritePostProcessingParameter ()**

```
int
CInterfaceLLT::WritePostProcessingParameter(unsigned int * pParameter,
                                                    unsigned int nSize);
```

Write post processing parameters.

Parameter

CInterfaceLLT	LLT class
pParameter	Post processing parameter array
nSize	Size of post processing parameter (max. 1024 DWORDs)

Return value

General return values

7.12.2 Extract post processing results

- **ConvertProfile2ModuleResult ()**

```
int
CInterfaceLLT::ConvertProfile2ModuleResults(const unsigned char * pProfileBuffer,
    unsigned int nProfileBufferSize, unsigned char * pModuleResultBuffer,
           unsigned int nResultBufferSize, TPartialProfile * pPartialProfile);
```

Extract post processing results from profile data. These overwrite the position and distance coordinates of the fourth stripe.

Parameter

CInterfaceLLT	LLT class
pProfileBuffer	Profile buffer
nProfileBufferSize	Size of profile buffer
pModuleResultBuffer	Result buffer
nResultBufferSize	Size of result buffer
pPartialProfile	Reference to partial profile (optional)

Return value

Number of bytes copied into buffer
General error codes
Specific return values:

ERROR_POSTPROCESSING_NO_PROF _BUFFER	-200	No profile buffer has been passed
ERROR_POSTPROCESSING_MOD_4	-201	The parameter nStartPointData or nPointDataWidth is not divisible by 4
ERROR_POSTPROCESSING_NO_RESULT	-202	No result block could be found in the profile
ERROR_POSTPROCESSING_LOW _BUFFERSIZE	-203	The buffer size for the result is too small
ERROR_POSTPROCESSING_WRONG _RESULT_SIZE	-204	The size of the result block in the profile is not correct

7.13 Functions for loading and saving profile data

7.13.1 Save profile data

- **SaveProfiles ()**

```
int
CInterfaceLLT::SaveProfiles(const char * pFilename, FileType FileType);
```

Saving of profiles. The profiles thereby are saved with the active profile configuration. The file name has to be indicated including extension. For finishing the saving *SaveProfiles(NULL, 0)* has to be called. If the maximum file size is reached during saving, an error message with the ERROR_STOPSAVING (see RegisterErrorMsg) value is sent.

Parameter

CInterfaceLLT	LLT class
pFileName	Name of file to be saved
FileType	File type

Return value

General return values

Specific return values:

ERROR_LOADSAVE_WRITING_LAST _BUFFER	-50	Error in the deactivation of the saving the last profile of the file could be damaged or not all have been saved
ERROR_LOADSAVE_AVI_NOT _SUPPORTED	-58	The operating system don't support the AVI format, please use windows 2000 or better
ERROR_LOADSAVE_WRONG_PROFILE _CONFIG	-60	The profile configuration or the file type mismatch to the transferred profiles/containers/video-pictures
ERROR_LOADSAVE_NOT_TRANSFERING	-61	The profile transfer is not active

- **FileType**

Available FileTypes. It is recommended to use the AVI-file format, this format is supported by the scanCONTROL Software from Micro-Epsilon:

FileType	Value	Description
AVI	0	AVI file
CSV	1	CSV file (profile data only)
BMP	2	BMP file (video image data only)
CSV_NEG	3	CSV file (profile data only) with mirrored z axis

7.13.2 Load profile data

- **LoadProfiles ()**

```
int
CInterfaceLLT::LoadProfiles(const char * pFilename,
          TPartialProfile pPartialProfile, TProfileConfig pProfileConfig,
                TScannerType pScannerType, unsigned int * pRearrangementProfile);
```

Loading of profiles from an *.AVI file. All *.AVI files can be loaded which are saved with the LLT.dll or the scanCONTROL programs from Micro-Epsilon. After loading the file, the single profiles in the file may be read out by the function *GetActualProfile()*. For terminating the file load LoadProfiles(NULL, NULL, NULL, NULL, NULL) has to be called.

The profile configuration of the loaded profile should always correspond to the profile configuration of the LoadProfiles function. Additionally at saved profile configuration of PROFILE also QUARTER_PROFILE and PURE_PROFILE or at QUARTER_PROFILE also PURE_PROFILE may be read out.

Parameter

CInterfaceLLT	LLT class
pFileName	Name of file to be loaded
pPartialProfile	Partial profile (optional)
pProfileConfig	Profile configuration
pScannerType	Scanner type
pRearrangementProfile	Rearrangement register value

Return value

Number of loaded profiles/container
General error codes
Specific return values:

ERROR_LOADSAVE_WHILE_SAVE_PROFILE	-51	File cannot be loaded, as saving is active
ERROR_LOADSAVE_NO_PROFILELENGTH_POINTER	-52	No pointer for the profile length has been passed
ERROR_LOADSAVE_NO_LOAD_PROFILE	-53	The filename is NULL, but no file is currently loaded
ERROR_LOADSAVE_STOP_ALREADY_LOAD	-54	On file has already been loaded, the loading has been stopped

ERROR_LOADSAVE_CANT_OPEN_FILE	-55	Cannot open file
ERROR_LOADSAVE_INVALID_FILE_HEADER	-56	The file header of the file to be loaded is wrong
ERROR_LOADSAVE_AVI_NOT_SUPPORTED	-58	The operating system don't support the AVI format, please use windows 2000 or better
ERROR_LOADSAVE_NO_REARRANGEMENT_POINTER	-59	The reference to pRearrangementProfile is NULL

7.13.3 Navigation in a loaded file

- **LoadProfilesGetPos ()**

```
int
CInterfaceLLT::LoadProfilesGetPos(unsigned int * pActualPosition,
                                  unsigned int * pMaxPosition);
```

Query the number of profiles in a file and the current reading position.

Parameter

CInterfaceLLT	LLT class
pActualPosition	Current reading position
pMaxPosition	Maximum reading position

Return value

General return values

- **LoadProfilesSetPos ()**

```
int
CInterfaceLLT::LoadProfilesSetPos(unsigned int pNewPosition);
```

Set reading position in a loaded file.

Parameter

CInterfaceLLT	LLT class
pNewPosition	Position from which should be read (0 sets the first profile)

Return value

General return values

Specific return value:

ERROR_LOADSAVE_FILE_POSITION_TOO_HIGH	-57	The requested position is bigger than or equal to the maximum position

7.14 Special CMM trigger functions

The special CMM trigger functions simplify the starting and terminating of the profile transfer with activated CMM trigger. Additionally the profiles with active CMM trigger may be saved in a file. The CMM trigger is only available for certain options of the scanCONTROL.

Note: only supported by the scanCONTROL 26xx/27xx/29xx/30xx series

- **StartTransmissionAndCmmTrigger ()**

```
int
CInterfaceLLT::StartTransmissionAndCmmTrigger(unsigned int nCmmTrigger,
   TransferProfileType TransferProfileType, unsigned int nProfilesForerun,
      const unsigned char *  pFilename, TFileType FileType, unsigned int nTimeout);
```

Start profile transmission with activated CMM trigger.

The StartTransmissionAndCmmTrigger function starts first profile transfer without forwarding the profiles thereby by callback. If the transmission was successful, i.e. the requested count of profiles has been transferred without failure, the first CMM trigger command with the divisor is sent to the scanCONTROL. Afterwards it is waiting for the first profile with active CMM trigger flag. From this profile on all further profiles are forwarded by callback. Additionally, if a file name has been passed, the saving of the profiles with the passed file name is started.

If a timeout occurs during the waiting for the profile, the function will be aborted. It is recommended to set nProfilesForerun to half the profile rate (e.g. 500 profiles at 1000 Hz) and the nTimeout to 3000 ms.

Parameter

CInterfaceLLT	LLT class
nCmmTrigger	First indicator word of the CMM trigger, which contains the divisor and the polarity
TransferProfileType	Profile type of transmission
nProfilesForerun	Count of the continuous received profiles from which a stable data transfer is adopted
pFileName	File name for the file to be saved
pFileType	File format of the saved file
nTimeout	Timeout in ms

Return value

General return values

Specific return values:

ERROR_CMMTRIGGER_NO_DIVISOR	-400	Divisor has to be > 0
ERROR_CMMTRIGGER_TIMEOUT_AFTER_TRANSFERPROFILES	-401	After *TransferProfiles()* no profiles have been received
ERROR_CMMTRIGGER_TIMEOUT_AFTER_SETCMMTRIGGER	-402	After setting of the CMM trigger not enough profiles with active CMM trigger have been received

- **StopTransmissionAndCmmTrigger ()**

```
int
CInterfaceLLT::StopTransmissionAndCmmTrigger(int nCmmTriggerPolarity,
                                             unsigned int nTimeout);
```

Stop profile transmission with activated CMM trigger.
The *StopTransmissionAndCmmTrigger()* function first stops the CMM trigger, by setting the divisor to 0 (but thereby taking into account the passed polarity). Afterwards it is waiting for the first profile without active CMM trigger flag. This profile and all the following will not be forwarded by callback, the profile transfer will be stopped and in case of active saving the saving will be terminated.

If a timeout occurs during waiting for the first profile without active CMM trigger flag, the function will be aborted. It is reasonable to indicate a time between 100 and 500 ms for the nTimeout.

Parameter

CInterfaceLLT	LLT class
nCmmTriggerPolarity	Polarity of CMM triggers (0 = low active, 1 = high active)
nTimeout	Timeout in ms

Return value

General return values
Specific return value:

ERROR_CMMTRIGGER_TIMEOUT_AFTER_SETCMMTRIGGER	- 402	After setting the CMM trigger no profile with deactivated CMM trigger has been received

7.15 Error value conversion function

- **TranslateErrorValue ()**

```
int
CInterfaceLLT::TranslateErrorValue(int ErrorValue,
                                   char * pString, int nStringSize);
```

Conversion of an error value to an error text.

Parameter

CInterfaceLLT	LLT class
ErrorValue	Error value
pString	Buffer for output string
nStringSize	String size

Return value

Number of characters copied to the buffer
General error codes
Specific return values:

ERROR_TRANSERRORVALUE_WRONG_ERROR_VALUE	-450	A wrong error value has been passed

ERROR_TRANSERRORVALUE_BUFFER_SIZE_TO_LOW	-451	The size of the passed buffer is too small for the string

7.16 Save configuration

- **ExportLLTConfig ()**

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

Exports 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);
```

Exports 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.

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
Return values of SetFeature()
Specific return value:

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.
Some functions may also return GENERAL_FUNCTION_CONTAINER_MODE_HEIGHT_CHANGED respectively ’2’. If this return value appears, the size of the image in the container mode has changed.

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_CONTAINER_MODE_HEIGHT_CHANGED	2	Function successfully executed, but the image size for the container mode has changed
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_WHILE_LOAD_PROFILE	-1000	Function could not be executed as the loading of profiles is active
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_WHILE_SAVE_PROFILES	-1007	Function could not be executed as the saving of profiles is active
ERROR_GENERAL_SECOND_CONNECTION_TO_LLT	-1008	A second instance is connected to this scanCONTROL via Ethernet or the serial port. Please close the second instance

8.2 SDK examples 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
Calibration	Use the position calibration feature of the sensor
GetProfiles_Poll	Transfer of profiles to the LLT.dll and pick up of profiles in polling mode
GetProfiles_Callback	Transfer of profiles to the LLT.dll and pick up of profiles with a callback
MultiShot	Transfer of a defined count of profiles from the scanCONTROL
PartialProfile	Transfer of partial profiles
LoadSave	Loading and saving of profiles
ContainerMode	Transfer of profile containers respectively grey scale maps
VideoMode	Transfer of video images from the sensor matrix
MultiLLT	Usage of more than one scanCONTROL in one program
CMMTrigger	Usage of the optional programmable trigger (CMM trigger)
TriggerProfile	Example with Software profile trigger
TriggerContainer	Example with Software container trigger
LLTInfo	Information about connected sensor
SetROIs	Set Regions of interest on the sensor array.
sC30xx_HighSpeed	Shows the configuration to use scanCONTROL 30xx in High Speed mode

`ReadPPPResults`	Shows how to read out post processing results from profiles

8.3 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;

[3] Operation Manual PartB 2800: Interface Specification for scanCONTROL 2800 Device Family; Firewire (IEEE 1394) Bus and Serial Port; Supplement B to the scanCONTROL 2800 Manual; MICRO-EPSILON Optronic GmbH;

[4] 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;

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

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

[7] scanCONTROL 2800 Quick Reference; Brief Introduction to scanCONTROL 2800 Device Family; MICRO-EPSILON Optronic GmbH;

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

[9] Interface documentation for LLT-DLL; MICRO-EPSILON Optronic GmbH;

[10] 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;

[11] scanCONTROL 3000 Quick Reference; Brief Introduction to scanCONTROL 3000 Device Family; MICRO-EPSILON Optronic GmbH;

[12] 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;

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