Schnittstellenhandbuch linLLT

C/C++
Linux

Version:	1.2
linLLT-Version:	0.2.0
Datum:	06.11.2017
Autor:	Daniel Rauch

I. Über dieses Dokument

Dieses Dokument hat das Ziel, es dem Leser grundsätzlich zu ermöglichen einen Laserprofilscanner vom Typ scanCONTROL mittels C/C++ in eine eigene Linux-Anwendung einzubinden. Basis dazu ist das Wissen um die grundlegende Verwendung der von den Bibliotheken zur Verfügung gestellten Programmierschnittstelle.

Dazu werden zu Beginn, neben allgemeinen Worten zu den Bibliotheken selbst, die Prinzipien der Messung und die daraus resultierenden Messwerte beschrieben. Dies ist insofern nötig, um ein gewisses Verständnis für die in der Software nötigen Messdaten zu schaffen. Des Weiteren werden die verschiedenen verfügbaren Mess-/Profildatenformate und die Varianten zu deren Übertragung dargestellt. Eine Erläuterung der Einschränkungen in Hinsicht Messgeschwindigkeiten schließt den allgemeinen Teil ab.

In die eigentliche Programmierung wird mittels der Beschreibung von häufig vorkommenden Basistasks anhand von Beispielcode eingeführt. Ausführliche Programmbeispiele und die vollständige Auflistung der API sollen die eigentliche Implementierung unterstützen.

II. Versionshistorie

Version	Datum	Autor	Status
0.1	10.09.2015	DRa	Initialer Entwurf
1.0	20.10.2015	UEi, DRa	Überarbeitete Version 1
1.1	08.11.2015	DRa	Ergänzungen
1.2	06.11.2017	DRa	Aktualisierung für v0.2.0

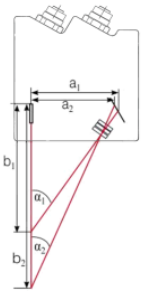
III. Inhalt

1 Einführung

1.1 Messprinzip und Messdaten

1.1.1 Prinzip der optischen Triangulation

Die scanCONTROL-Sensoren von Micro-Epsilon arbeiten, ähnlich den herkömmlichen Laserpunktsensoren, nach dem Prinzip der optischen Triangulation. Der Laserstrahl einer Laserdiode wird dabei mittels einer Spezialoptik aufgefächert und auf ein Messobjekt projiziert. Die Empfangsoptik fokussiert das diffus reflektierte Licht, welches schließlich von einem CMOS-Sensor detektiert wird. Um sicherzustellen, dass nur die Reflexion der projizierten Laserlinie ausgewertet wird, befindet sich vor dem Sensor ein Filter, der nur Licht im Wellenlängenbereich des Lasers passieren lässt.



Anhand der Position des detektierten Laserstrahls innerhalb einer Sensormatrixspalte kann nun mittels Triangulation der Abstand der Einzelmesspunkte von einer definierten Referenz im Sensor (z-Achse) bestimmt werden. In der Regel wird diese Referenz so gewählt, dass sich die Abstandswerte auf die Unterkante des Sensors beziehen. Die allgemeine Abstandsberechnung erfolgt über folgende Formel:

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

Die Messauflösung in z-Richtung ist durch die Pixelanzahl der Sensormatrix in der z-Achse festgelegt. Da Reflexionen von mehreren Pixeln detektiert werden, wird zur Bestimmung des z-Wertes der Reflexionsschwerpunkt dieser Pixel verwendet (subpixelgenaue Bestimmung).

Entsprechend der Position der Messpunkte innerhalb einer Zeile der Matrix wird ein Abstandswert einem korrespondierenden Punkt auf der x-Achse zugeordnet. Die Anzahl der Pixel der Sensormatrix in x-Richtung entscheidet dann darüber, wie viele Einzelmesspunkte es gibt.

Das direkte Messergebnis ist ein zweidimensionaler Profilverlauf, welcher auf eine Maßeinheit [mm] kalibriert ist. Dadurch ist sowohl eine referenzielle, als auch eine absolute Messung möglich. Eine 3D-Messung erfolgt über eine Bewegung des Sensors oder des Messobjekts in y-Richtung. Durch gleichförmige Bewegung bei definierter Profilfrequenz oder durch Verwenden eines Encoders, der die Bewegung abbildet, kann ein Gitternetz mit äquidistant verteilten Punkten generiert werden.

1.1.2 Aufgenommene Messwerte

Die von einem scanCONTROL-Sensor standardmäßig gesendeten Daten beinhalten neben den eigentlich detektierten Abstands- und Positionswerten auch weitere Rahmendaten der Messung, wie Intensität, Reflexionsbreite, Moment 0 und Moment 1. Außerdem wird der aktuell eingestellte Schwellwert übertragen. Die Werte sollen im Folgenden erläutert werden:

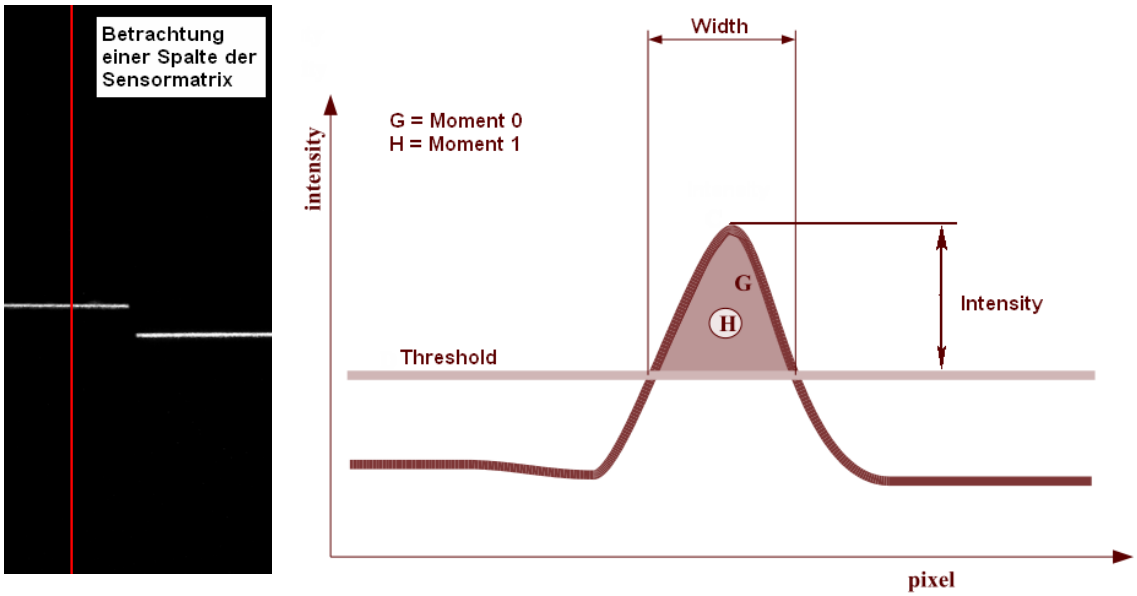


Abb. 1: Messdatenermittlung

- Abstand: Zur Ermittlung des Abstandes bzw. des z-Wertes eines Messpunktes wird der Gravitationsschwerpunkt der auf der CMOS-Sensorspalte detektierten Reflexion berechnet. Dieser wird im Sensor, anhand einer Kalibriertabelle, zu einer tatsächlichen Abstandskoordinate zurückgerechnet. Übertragen wird ein 16 Bit unsigned Integer Feld, das noch mit sensorabhängigen Skalierungsfaktoren verrechnet werden muss.

- Position: Die Position (x-Wert) korrespondiert mit der Pixelspalte des CMOS-Sensors. Pro Spalte wird ein Positionswert ermittelt. Mittels der auf dem Sensor gespeicherten Kalibriertabelle wird dieser auf die tatsächliche Position umgerechnet. Übertragen wird ebenfalls ein zu skalierendes 16 Bit unsigned Integer Feld.

- Intensität: Der übertragene Wert gibt die Differenz zwischen der maximal detektierten Intensität der aktuellen Reflexion und dem Threshold wieder. Intensität bedeutet hier, wie viel Laserlicht auf einen Pixel der Matrix gefallen ist. Voraussetzung für das Erkennen einer Reflexion ist, dass die Intensität über dem Threshold liegt. Übertragen wird ein 10 Bit unsigned Integer Feld.

- Reflexionsbreite: Die Reflexionsbreite sagt aus, über wie viele Pixel die aktuelle Reflexion zusammenhängend über dem Threshold war. Übertragen wird ein 10 Bit unsigned Integer Feld.

- Moment 0: Gibt die für die aktuelle Reflexion detektierte integrale Intensität („Fläche der Reflexion“) wieder. Das Moment ergibt sich somit aus dem Integral der über dem Schwellwert liegenden Intensität über die Reflexionsbreite; siehe Abb. 1 (G). Der Wert wird mit 32 Bit als unsigned Integer übertragen.

- Moment 1: Gibt den Schwerpunkt der Reflexion wieder, der als Basis für die Umrechnung in Abstands- und Positionswerte anhand der Kalibriertabelle verwendet wird. Der Schwerpunkt wird ebenfalls als 32 Bit unsigned Integer übertragen.

- Threshold: Der für diesen Messpunkt verwendete Schwellwert, der sich aus der eingestellten absoluten bzw. der dynamisch errechneten Schwelle und der ermittelten Fremdlichtunterdrückung zusammensetzt. Übertragen wird ein 10 Bit unsigned Integer Feld.

Es ist anzumerken, dass diese Werte bezogen auf die, im Sensor eingestellte, zu detektierende Reflexion gesendet werden. Zur Auswahl stehen die erste bzw. letzte auf der Spalte erkannte Reflexion über dem Schwellwert, die Reflexion mit der maximalen Intensität und die mit der größten integralen Intensität (siehe Abb. 2).

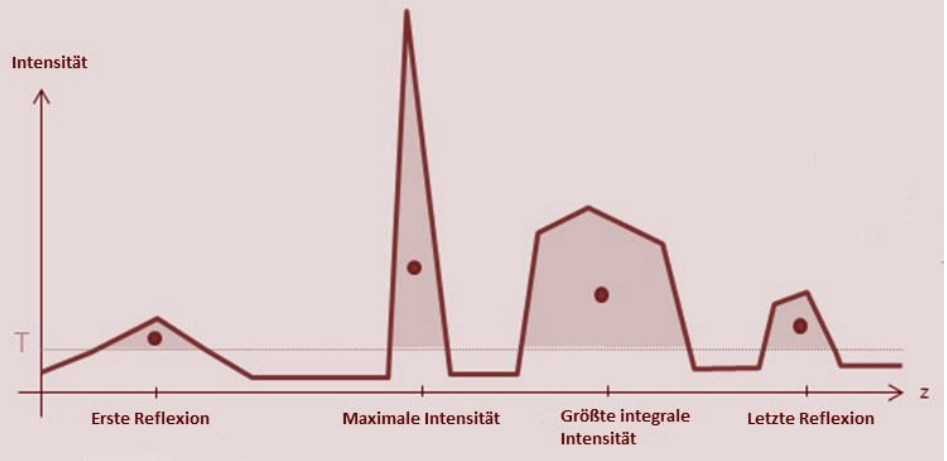


Abb. 2: Detektierbare Reflexionen

1.2 linLLT-Bibliotheken

1.2.1 Überblick

Die linLLT-Schnittstelle besteht aus zwei *Shared Object*-Bibliotheken. Konkret sind das die C-Hauptbibliothek libmescan und die C++-Wrapper-Bibliothek libLLT. Ein Shared Object ist eine Bibliothek, die automatisch bei Programmstart in das Projekt gelinkt wird. Sie muss auch zum Kompilierzeitpunkt verfügbar sein, kann aber ohne Rekompilation der Applikation ausgetauscht werden, solange sich das ABI nicht geändert hat.

Beide zur Verfügung gestellten Bibliotheken basieren auf der Open Source (LGPL) Schnittstelle aravis [9]. Aravis bildet die Grundlage für die Übertragung des Bildstroms über Ethernet, sowie für die Auswertung der GeniCam-XML zur Sensorkonfiguration. Getestete bzw. empfohlene Version ist die 0.5.9 (Stand: 06/2017). libmescan stellt hierbei Funktionen zur Verfügung, die zur Scannersuche, –steuerung und -initialisierung, sowie zur Messdatenübertragung bzw. -interpretation dienen und bietet daher die Möglichkeit eigene C-Applikationen zu erstellen. Aufrufkonvention ist Linux-typisch *cdecl*. libLLT wrappt aravis und libmescan und stellt eine C++-API zur Verfügung, die das einfache Erstellen von C++ Applikationen erlaubt. Diese ermöglicht einen weitgehend reibungslosen Übergang von der Windows SDK (LLT.dll), da die API ähnlich gehalten wurde.

1.2.2 Kompilation

Um eine Applikation mit linLLT kompilieren zu können, sind einige weitere Libraries am Rechner zur Verfügung zu stellen. Spezifisch sind das aravis-0.6 und deren Abhängigkeiten, wie libxml2 und glib. Ein Makefile steht augenblicklich nicht zur Verfügung. Konkret die nötigen Linkerflags zum Kompilieren einer Applikation mit linLLT:

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

Fehler! Kein gültiger Dateiname.

2 Format der Messdaten

2.1 Video Mode

Im Video Mode überträgt der Sensor das vom CMOS-Sensor direkt aufgenommene Bild als 8-Bit Graustufen Bitmap (Abb. 3). Dabei werden nur die reinen Bilddaten übertragen. Eventuelle Header oder das Spiegeln der Zeilen müssen extern realisiert werden. Dieses Datenformat besitzt keinen Zeitstempel. Die Messfrequenz sollte in diesem Modus 25 Hz nicht übersteigen und der Übertragungspuffer ist zwischen 3 und 5 zu wählen. Zu beachten ist außerdem, dass bei Ethernet-Scannern der 29xx-Serie eine Gigabit-Ethernet-Verbindung nötig ist, um die Bilddaten mit der empfohlenen Frequenz von 25 Hz zu übertragen. Der Scanner der 29xx-Serie benötigt beispielhaft eine Bandbreite von ca. 262 Mbit/s bei 25 Hz (25 Hz * 1024 * 1280 * 8 Bit).

Abb. 3: Beispiel Video Mode

2.2 Einzelprofilübertragung

2.2.1 Profildatenformat allgemein

Im Einzelprofilmodus wird standardmäßig pro Messung ein Datenfeld übertragen, das pro Messpunkt 64 Byte breit ist. Die Höhe des Datenfeldes wird von der Anzahl der Punkte pro Profil festgelegt. Die 64 Byte unterteilen sich in vier Streifen mit jeweils 16 Byte Breite. Jeder Streifen kann ein komplettes Profil enthalten; im Normalfall enthält nur der erste Streifen gültige Profildaten - sind Mehrfachreflexionen vorhanden, können aber auch die anderen Streifen Profildaten enthalten. Die letzten 16 Byte des Datenfeldes enthalten den Timestamp, was in der Standardeinstellung (*Full Set*) dem letzten Punkt des vierten Streifens entspricht.

Pro Streifen werden für jeden Messpunkt alle Informationen übertragen, die in den vorherigen Abschnitten beschrieben wurden. Diese sind jeweils in folgender Struktur angeordnet:

0..7	8..15	16..23	24..31
Res. (2 Bit)	Reflexionsbreite (10 Bit)	Max. Intensität (10 Bit)	Threshold (10 Bit)
Position (16 Bit)		Abstand (16 Bit)	
Moment 0 (32 Bit)			
Moment 1 (32 Bit)			

Die byteweise Daten-Formatierung der einzelnen Messwerte ist mit Big-Endian ausgeführt. Außerdem müssen, wie bereits angedeutet, die Positions- und Abstandsdaten (X/Z) noch mittels Skalierungsfaktoren umgerechnet werden. Diese sind für jeden Messbereich verschieden. Abb. 4 zeigt schematisch die Gesamtstruktur einer Übertragung.

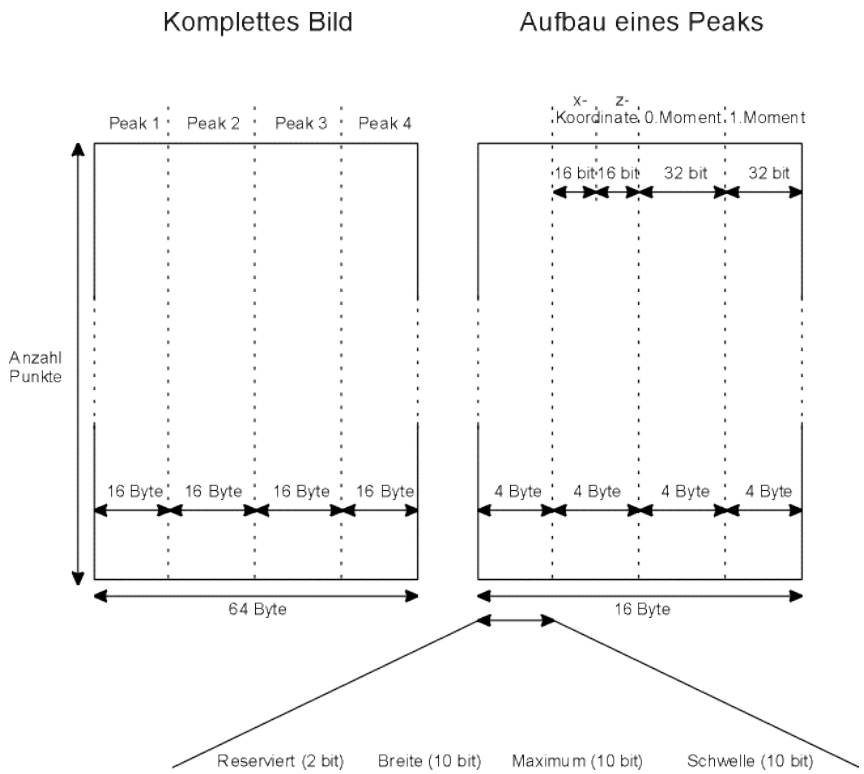


Abb. 4: Aufbau Profilübertragung

2.2.2 Timestamp-Informationen

Der in den letzten 16 Byte übertragene Timestamp beinhaltet folgende Rahmendaten eines gemessenen Profils (Datenformat: unsigned Integer; Big-Endian):

- Profilzähler: Inkrementeller Zähler mit dem sich Profile identifizieren lassen; wird bei jedem neuen Profil um eins erhöht. Für das Feld sind 24 Bit reserviert – es läuft damit nach 16777215 Profilen über.

- Startzeit Belichtung: Beinhaltet den absoluten Zeitpunkt bei dem die Belichtung gestartet wurde. Die interne Uhr hat dabei eine Periode von 128 Sekunden. Das 32-Bit breite Feld besteht aus einem Sekundenzähler, einem Zykluszähler und dem Zyklusoffset. Aus diesen Werten lässt sich der eigentliche Verschlussöffnungszeitpunkt berechnen.

- Flankenzähler: Hier wird je nach Scannereinstellung entweder die zweifache Anzahl der erkannten Encoderflanken oder der Status der Digitaleingänge übertragen. Das Feld ist 16-Bit groß.
- Endzeit Belichtung: Beinhaltet den absoluten Zeitpunkt bei dem die Belichtung beendet wurde. Ansonsten wie bei *Startzeit Belichtung* beschrieben.

Struktur der Timestamp-Daten ist wie folgt:

0..7	8..15	16..23	24..31
Flags (2 Bit)	Reserviert (6 Bit)	Profilzähler (24 Bit)	

Startzeit Belichtung (32 Bit)	
Flankenzähler bzw. DigIn (16 Bit)	Reserviert (16 Bit)
Endzeit Belichtung (32 Bit)	

Die beiden Zeitstempel für die Belichtung setzen sich folgendermaßen zusammen:

Sekundenzähler (7 Bit)	Zykluszähler (13 Bit)	Zyklusoffset (12 Bit)

Der eigentliche Zeitstempel folgt dann aus folgender Berechnungsvorschrift:

$$\text{Zeitstempel} = \text{Sekundenzähler} + \frac{\text{Zykluszähler}}{8000} + \frac{\text{Zyklusoffset}}{8000*3072}$$

(Bemerkung: Der Zykluszähler läuft bei 8000 und der Zyklusoffset bei 3072 über!)

2.2.3 CMM-Timestamp

Ist die Triggerung für eine *Coordinate Measuring Machine* (CMM; Koordinatenmessmaschine) aktiviert, ändert sich das Format des Timestamps. Statt dem Encoder-Flankenzähler und der reservierten 16-Bit werden folgende CMM-spezifische Informationen übertragen:

CMM-Flankenzähler (16 Bit)	CMM Trigger Flag (1 Bit)	CMM aktiv Flag (1 Bit)	CMM Triggerimpulszähler (14 Bit)

2.2.4 Alle Messdaten (Full Set, PROFILE)

Standardmäßig werden vom Sensor alle Messdaten wie in 1.1.2 beschrieben übertragen. Mit dem vordefinierten Format *Full Set* werden alle Daten aus der Übertragung extrahiert. Die Datenmenge pro Messung ist hier mit 64 Byte mal der Anzahl der Punkte pro Profil anzusetzen. Im Rahmen der DLL wird diese Konfiguration auch als Profilkonfiguration *PROFILE* bezeichnet. Ausgabeformat ist Big-Endian.

2.2.5 Ein Streifen (QUARTER_PROFILE)

Die Profilkonfiguration *QUARTER_PROFILE* ermöglicht es, nur einen Streifen aus der Übertragung zu extrahieren. Dies hat zur Folge, dass eine kleinere Datenmenge verarbeitet werden muss. Die Timestamp-Informationen werden an die Profildaten angehängt. Die Daten pro Messung beschränken sich daher hier auf 16 Byte mal der Punkte pro Profil plus 16 Byte Timestamp. Es ist zu beachten, dass die Profilkonfiguration per se nicht die übertragene Datenmenge reduziert, sondern nur der Teil des Puffers ausgewertet wird, der dem gewünschten Streifen entspricht. Es muss daher weiterhin die volle Datenmenge übertragen werden. Ausgabeformat ist Big-Endian.

2.2.6 X/Z-Daten (PURE_PROFILE)

Die Profilkonfiguration *PURE_PROFILE* extrahiert nur die Abstands- und Positionswerte (X/Z-Daten) aus dem aktuell ausgewählten Streifen (siehe Kapitel **Fehler! Verweisquelle konnte nicht efunden werden.**). Ansonsten verhält sie sich wie die *QUARTER_PROFILE* Konfiguration, d.h. die tatsächlich übertragene Datenmenge wird nicht reduziert. Die ausgewertete Datenmenge reduziert sich auf 4 Byte mal der Punkte pro Profil plus 16 Byte Timestamp. Zu beachten ist, dass hier im Little-Endian-Format übertragen wird.

2.2.7 Partielles Profil (PARTIAL_PROFILE)

Ein partielles Profil (*PARTIAL_PROFILE*) wird direkt im Scanner erzeugt und kann daher die zu übertragende Datenmenge stark reduzieren. Auch das Processing der Daten im Scanner wird

beschleunigt, was bei hohen Frequenzen wichtig werden kann. Die Größe des Profils kann mittels der folgenden vier Parameter definiert werden:

1. *StartPoint*: Erster Messpunkt der im Profil enthalten sein soll
2. *StartPointData*: Offset ab welchem Byte die Daten eines Punktes im Profil enthalten sein sollen
3. *PointCount*: Anzahl der Messpunkte ab dem *StartPoint*, die im Profil enthalten sein sollen
4. *PointDataWidth*: Anzahl an Bytes ab dem *StartPointData*-Offsets, die im Profil enthalten sein sollen

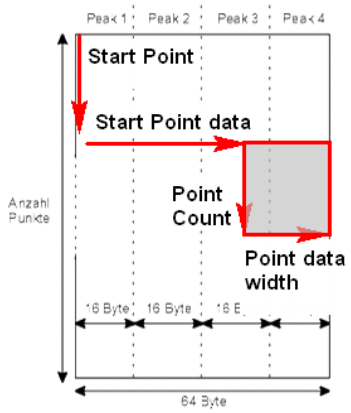


Abb. 5: Illustration Partial Profile

Alle Angaben haben als Bezugspunkt den linken oberen Punkt der Profildaten und der Basisindex ist jeweils 0. Die Angaben müssen durch die auf dem Sensor festgelegte Unitsize teilbar sein (meist 4). Am Ende des partiellen Profils befindet sich wieder der Timestamp, jedoch wird er nicht angehängt, sondern überschreibt die letzten 16 Bytes. Die übertragene Datenmenge reduziert sich somit auf *PointDataWidth* Bytes mal *PointCount.* Siehe Code-Beispiel Kapitel 5.13. Ausgabeformat ist Big-Endian.

2.3 Container Mode

Prinzipiell erlaubt der Container Mode eine Zusammenfassung von Daten aus mehreren Profilen in einen großen Übertragungscontainer. Dies hat unter anderem den Vorteil, dass die nötige Reaktionszeiten der Software und der Daten-Overhead reduziert werden kann.

2.3.1 Standard Container Mode

Standardmäßig können im Container Mode mehrere komplette Profile in einen logischen Übertragungscontainer zusammengefasst werden. Dabei sammelt der Sensor so lange Profile, bis die gewünschte Anzahl erreicht ist und überträgt diese als Gesamtpaket. Die maximale Containergröße hängt vom Sensor ab (128 Mbyte bzw. 4096 Profile). Wie erwähnt, ist hier der Hauptvorteil, dass die korrespondierende Software in längeren Reaktionsintervallen arbeiten kann.

2.3.2 Rearranged Container Mode (Transponierter Container Mode)

Mit der sog. *Rearrangement*-Funktion können nur die Daten in den Container geschrieben werden, die tatsächlich nötig sind. Der Anwender kann dabei frei bestimmen, von welchem Streifen er welche Messwerte übertragen will. Diese Werte werden dann pro Profil hintereinander im Container abgespeichert. Für den Timestamp kann man ein zusätzliches Feld

definieren, falls gewünscht. Die gewählten Werte werden dann für eine gewünschte Anzahl von Profilen gesammelt und dann übertragen.

Ein häufig verwendeter Sonderfall ist der transponierte Container Mode, bei dem nur die Abstandsdaten (und optional Positionsdaten) extrahiert werden. Diese sind dann so angeordnet, dass sie ein 16-Bit Graustufen-Bitmap ergeben, was sich anschließend mittels Bildverarbeitungsalgorithmen analysieren lässt. Die Breite des Bitmaps wird hierbei von der Anzahl der Punkte pro Profil und die Höhe von der Anzahl der Profile im Container festgelegt. Standard-Datenformat ist Big-Endian, kann aber in Little-Endian geändert werden.

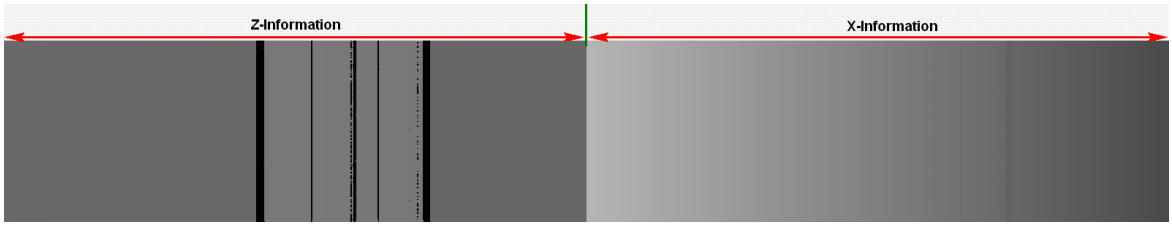


Abb. 6: Beispielbild transponierter Container Mode

3 Datenübertragung vom scanCONTROL Sensor

3.1 Datenübertragung

Die Datenübertragung wird softwareseitig gestartet und beendet. Ist die Übertragung aktiv, werden abhängig von der Profilfrequenz Daten in einen Empfangspuffer geschrieben. Konfiguriert werden kann die Übertragung entweder für eine kontinuierliche Datenextraktion aus dem Puffer (*NORMAL_TRANSFER*) oder für die Extraktion einer definierten Anzahl an Profilen (*SHOT_TRANSFER*).

3.2 Pollen von Messdaten

Für weniger zeitkritische Anwendungen bzw. Anwendungen, die nicht alle empfangenen Profile oder Videobilder verarbeiten müssen, gibt es die Möglichkeit Daten aktiv aus dem Empfangspuffer anzufordern. Bei jedem Poll wird das zuletzt empfangene Profil/Bild ausgelesen und in einen von der Anwendersoftware bereitgestellten Datenpuffer kopiert, der zur Weiterverarbeitung verwendet werden kann. Ist seit dem letzten Poll noch kein neues Profil angekommen, wird die Anwendung informiert.

3.3 Nutzen von Callbacks

Der Callback wird immer dann aufgerufen, wenn ein neues Profil oder ein neuer Container empfangen wurde. Bei vollständiger Abarbeitung der Callback-Routine ist ein Event zu setzen. Als Parameter beinhaltet dieser Callback einen Zeiger auf das soeben empfangene Profil bzw. den empfangenen Container. Das Profil wurde vorher schon in die aktuelle Profilkonfiguration gewandelt. Mit Hilfe dieses Zeigers kann die Callback-Funktion die empfangenen Daten in einen eigenen Puffer zur Weiterverarbeitung kopieren. Wichtig ist dabei, dass die Callback-Funktion sehr kurz ist und möglichst schnell wieder beendet wird, damit der nächste Puffer vom Treiber geholt werden kann.

4 Messgeschwindigkeit

Die maximal mögliche Messgeschwindigkeit hängt von mehreren Parametern ab und unterscheidet sich je nach Sensortyp. Zusätzlich hat die gegebene Netzwerkinfrastruktur eine

einschränkende Rolle. Eine Überschreitung dieser Maximalfrequenz führt zu verlorenen und/oder korrupten Daten und sollte komplett vermieden werden.

Um von vorneherein Performanceprobleme zu vermeiden, sollte die komplette Netzwerkinfrastruktur zwischen Sensor und PC Gigabit-Ethernet tauglich sein. Speziell falls Sensoren der 29xx-Serie einsetzt werden, kommen 100 Mbit/s-Netzwerke schnell an die physikalischen Grenzen. Außerdem sollte der Rechner, auf dem die Software ausgeführt wird, mit ausreichend performanter Hardware ausgestattet sein, da v.a. bei mehreren Sensoren eine große Datenmenge abzuarbeiten ist.

Häufigster Flaschenhals, nach der Netzwerkumgebung, ist das eingesetzte Messfeld. Das Messfeld beschreibt, welcher Teil der Sensormatrix tatsächlich ausgelesen wird. Je weniger Fläche der Sensormatrix ausgelesen wird, desto größer ist die Maximalfrequenz. Eine detaillierte Auflistung der je nach Messfeld möglichen Frequenzen, ist in der (der Sensordokumentation beiliegenden) *Quickreference* [5-8] gegeben. Grundsätzlich sollte bei hochfrequenten Messungen nur der für die Messung relevante Teil der Matrix ausgelesen werden.

Weiterhin beschränkt die Länge der Belichtungszeit die Messfrequenz. Die Maximalfrequenz ergibt sich hier aus dem Reziprokwert der Belichtungsdauer. Sind extensive Post-Processing Operationen auf dem Sensor konfiguriert worden (SMART- oder GAP-Sensoren), kann auch hier ein Flaschenhals entstehen. Die mit der aktuellen Post-Processing-Konfiguration mögliche Maximalfrequenz kann den *scanCONTROL Configuration Tools* entnommen werden. Bei höheren Frequenzen kann auch bei Sensoren ohne aufgespielten Processing die Berechnungszeit nicht ausreichen. Dies kann vermieden werden, indem man die auszuwertenden Punkte dem Messfeld anpasst und/oder nur einen Streifen bzw. die X/Z-Werte überträgt (via partiellem Profil). Vor allem bei der jeweiligen Maximalgeschwindigkeit ist auf die Minimalkonfiguration zurückzufallen. Dies hat keine Auswirkungen auf die Qualität der Messung, da alle weiteren Punkte außerhalb des Messfeldes liegen und nur invalide Werte liefern würden.

5 Typische Code-Beispiele mit Verweise auf das SDK

Im folgenden Abschnitt werden C++-Code-Minimalbeispiele für verschiedene Einbindungsschritte gezeigt. Komplettbeispiele mit Fehlerbehandlung etc. sind im Projektordner des SDKs zu finden. Bis auf Beispiel 5.1 ist vorausgesetzt, dass die Verbindung mit dem Sensor hergestellt ist. Bei einigen Beispielen werden Register beschrieben (*SetFeature(…)*). Die Registeradressen sind in der SDK auf Makros abgebildet (z.B. `FEATURE_FUNCTION_SHUTTERTIME` für die Belichtungszeit). Die detaillierten Registerbeschreibungen sind im Operation Manual Part B [1-4] zu finden; diese ist der Scannerdokumentation beiliegend.

5.1 Verbindung mit Sensor herstellen

Dieses Beispiel zeigt, wie ein Sensor gefunden und eine Verbindung hergestellt werden kann.

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

// Erzeugen eines Handles für einen Ethernet Scanner
CInterfaceLLT *pLLT = new CInterfaceLLT();  // (C) LLT *pLLT = create_llt_device();

// Suchen nach Scannern am Interface
CInterfaceLLT::GetDeviceInterfaces(&Interfaces[0], Interfaces.size());

// Setzen des Pfades zur Sensorparameterdatei (optional)
pLLT->SetPathtoDeviceProperties("./device_properties.dat");

// Zuordnen des ersten gefundenen Interfaces zum Handle
pLLT->SetDeviceInterface(Interfaces[0]);

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

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

5.2 Profilfrequenz und Belichtungszeit setzen

Dieses Beispiel zeigt die Vorgehensweise zum Ändern der Profilfrequenz und der Belichtungszeit. Die übergebenen Werte beschreiben die Zeit in 10 µs-Schritten. Die Profilfrequenz kann nicht direkt gesetzt werden. Sie setzt sich aus der Belichtungszeit (*ShutterTime*) und der Leerlaufzeit (*IdleTime*) zusammen. Sie berechnet sich aus:

$$Profilfrequenz = \frac{1}{(ShutterTime + IdleTime) * 10\ \mu s}$$

```
unsigned int ShutterTime= 100;
unsigned int IdleTime   = 900;

// Setzen der Belichtungszeit auf 1 ms (100*10 us)
pLLT->SetFeature(FEATURE_FUNCTION_SHUTTERTIME, ShutterTime);

// Setzen der Leerlaufzeit auf 9 ms (900*10 us)
pLLT->SetFeature(FEATURE_FUNCTION_IDLETIME, IdleTime);
```

Siehe API: SetFeature(), FEATURE_FUNCTION_SHUTTERTIME, FEATURE_FUNCTION_IDLETIME
Siehe Quickreference: *OpManPartB.html#shutter, OpManPartB.html#idletime*

Im Beispielcode wird also die Belichtungszeit auf 1 ms und die Profilfrequenz auf 100 Hz festgelegt.

5.3 Pollen von Messwerten

Dieses Beispiel zeigt die Profildatenabholung über aktives Pollen. Dazu wird mittels der Funktion *GetActualProfile()* das zuletzt empfangene Profil, das sich im Empfangspuffer befindet, in einen Puffer der Anwendung zur Weiterverarbeitung kopiert. Anschließend wird aus den Pufferdaten die eigentliche X/Z-Information extrahiert (*ConvertProfile2Values()*). Diese Funktion rechnet schon die nötigen Skalierungsfaktoren für den Scannertyp ein; die Ausgabewerte sind Positions- und Abstandswerte in Millimeter.

```
unsigned int Resolution;
TScannerType LLTType;

[…] // Connect

pLLT->GetLLTType(&LLTType);

// Festlegen von Puffer für ein vollständiges Profil und X/Z-Werte
std::vector<unsigned char>ProfileBuffer(Resolution * 64);
std::vector<double>ValueX(Resolution);
std::vector<double>ValueZ(Resolution);

unsigned int LostProfiles = 0;

// Starten der kontinuierlichen Profilübertragung
pLLT->TransferProfiles(NORMAL_TRANSFER, true);

// Pollen eines Profiles und Abspeichern in Puffer
// Anm.: Falls noch kein neues Profil seit dem letzten Aufruf angekommen ist
// gibt die Funktion -104 zurück. Ggfls. in einer Schleife abfragen.
pLLT->GetActualProfile(&ProfileBuffer[0], ProfileBuffer.size(), PROFILE,
                                                                &LostProfiles));

// Konvertierung von Pufferdaten zu X/Z-Werten des Profils
CInterfaceLLT::ConvertProfile2Values(&ProfileBuffer[0], ProfileBuffer.size(),
     Resolution, PROFILE, LLTType, 0, NULL, NULL, NULL, &ValueX[0], &ValueZ[0],
                                                                NULL, NULL);

// Beenden der kontinuierlichen Profilübertragung
pLLT->TransferProfiles(NORMAL_TRANSFER, false);
```

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

5.4 Auslesen via Callback

Dieses Beispiel zeigt die Profildatenabholung mittels Callback. Dazu wird ein Callback registriert, der bei einem neu angekommenen Profil den Puffer kopiert und anschließend einen Event signalisiert. Nach Empfangen des Profils wird die Übertragung beendet.

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

[…] // Init, Connect und Auslesen von Resolution und TScannerType

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

event = CInterfaceLLT::CreateEvent();

// Registrieren der Callback-Funktion
pLLT->RegisterBufferCallback((gpointer)&NewProfile, NULL)

// Starten der kontinuierlichen Profilübertragung
pLLT->TransferProfiles(NORMAL_TRANSFER, true);

CInterfaceLLT::ResetEvent(event);

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

// Beenden der kontinuierlichen Profilübertragung
pLLT->TransferProfiles(NORMAL_TRANSFER, false);

// Konvertieren der Rohdaten in mm
CInterfaceLLT::ConvertProfile2Values(&ProfileBuffer[0], ProfileBuffer.size(),
Resolution, PROFILE, LLTType, 0, , NULL, NULL, &ValueX[0], &ValueZ[0], NULL, NULL);

CInterfaceLLT::FreeEvent(event);

// Callback-Funktion (kopiert ein Profil in den Puffer und signalisiert danach)
void NewProfile (const void *pucData, size_t uiSize, gpointer user_data)
{
   if (uiSize == ProfileBuffer.size())
   {
      memcpy(&ProfileBuffer[0], pucData, uiSize);
   }
   CInterfaceLLT::SetEvent(event);
}
```

Siehe API: RegisterCallback(), TransferProfiles(), ConvertProfiles2Values(), EventHandling

5.5 Profilfilter setzen

Dieses Beispiel zeigt das Setzen von Resampling-, Median- und Average-Filter.

```
// Setze Average-Filter auf 7 Taps
unsigned int ProfileFilter = FILTER_AVG_7;
// Setze Median-Filter auf 5 Taps
ProfileFilter |= FILTER_MEDIAN_5;
// Setze Resampling (Inter/Extrapolation von invaliden Punkten; Alle Infos; huge)
ProfileFilter |= FILTER_RESAMPLE_EXTRAPOLATE_POINTS |
                        FILTER_RESAMPLE_ALL_INFO | FILTER_RESAMPLE_HUGE;
// Setzen der eingestellten Filter
pLLT->SetFeature(FEATURE_FUNCTION_PROFILE_FILTER, ProfileFilter);
```

Siehe API: SetFeature(), FEATURE_FUNCTION_PROFILE_FILTER
Siehe Quickreference: *OpManPartB.html#profilefilter*

5.6 Encoder

Dieses Beispiel zeigt das Aktivieren der Encoder-Triggerung über digitale Eingänge.

```
unsigned int Encoder = 0;

// Setze Trigger input auf encoder
unsigned int Trigger = TRIG_MODE_ENCODER;
// Setze digitale Eingänge als Trigger input und aktiviere ext. Triggerung
Trigger |= TRIG_INPUT_DIGIN | TRIG_EXT_ACTIVE;
// Setzen der Triggereinstellungen
pLLT->SetFeature(FEATURE_FUNCTION_TRIGGER, Trigger);

// Setzen des Multifunktionsports auf bidirektionalen 24V HTL-Encoderbetrieb
unsigned int MultiPort = MULTI_DIGIN_ENC_INDEX | MULTI_LEVEL_24V
                                                | MULTI_ENCODER_BIDIRECT;
pLLT->SetFeature(FEATURE_FUNCTION_RS422_INTERFACE_FUNCTION, MultiPort);

// Lese Maintenance-Register und aktivere Encoder
pLLT->GetFeature(FEATURE_FUNCTION_MAINTENANCEFUNCTIONS, &Encoder);
Encoder |= MAINTENANCE_ENCODER_ACTIVE;
pLLT->SetFeature(FEATURE_FUNCTION_MAINTENANCEFUNCTIONS, Encoder);
```

Siehe API: SetFeature(), FEATURE_FUNCTION_TRIGGER, FEATURE_FUNCTION_MAINTENANCEFUNCTIONS
Siehe Quickreference: *OpManPartB.html#trigger, OpManPartB.html#maintenance*

5.7 Externe Triggerung

Dieses Beispiel zeigt das Aktivieren der externen Triggerung über digitale Eingänge.

```
// Setze Trigger input auf pos. pulse mode
unsigned int Trigger = TRIG_MODE_PULSE | TRIG_POLARITY_HIGH;
// Setze digitale Eingänge als Trigger input und aktiviere ext. Triggerung
Trigger |= TRIG_INPUT_DIGIN | TRIG_EXT_ACTIVE;
// Setzen der Triggereinstellungen
pLLT->SetFeature(FEATURE_FUNCTION_TRIGGER, Trigger);

// Setzen des Multifunktionsports auf 5V TTL-DigIn-Trigger
unsigned int MultiPort = MULTI_DIGIN_TRIG_ONLY | MULTI_LEVEL_5V;
pLLT->SetFeature(FEATURE_FUNCTION_RS422_INTERFACE_FUNCTION, MultiPort);
```

Siehe API: SetFeature(), FEATURE_FUNCTION_TRIGGER
Siehe Quickreference: *OpManPartB.html#trigger*

5.8 Software-Trigger

Dieses Beispiel zeigt das Aktivieren der externen Triggerung über den Software-Trigger.

```
// Aktiviere ext. Triggerung
unsigned int Trigger = TRIG_EXT_ACTIVE;

// Setzen der Triggereinstellungen
pLLT->SetFeature(FEATURE_FUNCTION_TRIGGER, Trigger);

// Software-Triggerung; Löst Aufnahme eines Profils aus
pLLT->TriggerProfile();
```

Siehe API: SetFeature(), FEATURE_FUNCTION_TRIGGER, TriggerProfile()
Siehe Quickreference: *OpManPartB.html#trigger*

5.9 Peak-Filter setzen

Dieses Beispiel zeigt, wie die sog. Peak-Filter des Scanners gesetzt werden können. Diese ermöglichen es, Punkte außerhalb eines gewissen Intensitäts- und/oder Reflexionsbreitenbereich auszusortieren. Seit scanCONTROL Firmware-Version v43 können die Peak-Filter mittels SetFeature/GetFeature gesetzt/gelesen werden, z.B. SetFeature(FEATURE_FUNCTION_PEAKFILTER_WIDTH, (max_width << 16) + min_width). Um die Einstellungen zu aktivieren, muss eine 0 in das FEATURE_FUNCTION_SHARPNESS-Register geschrieben werden.

```
// Setze die gewünschten Peak-Werte
unsigned short min_width       = 2;   // Werte <2 erhöhen Rauschen immens
unsigned short max_width       = 1023;
unsigned short min_intensity   = 0;
unsigned short max_intensity   = 1023;

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

Siehe API: SetPeakFilter()

5.10 Frei definierbares Messfeld setzen

Dieses Beispiel zeigt, wie ein frei definierbares Messfeld gesetzt werden kann. Dies ermöglicht eine flexible Definition der Messfeldgröße. Seit scanCONTROL Firmware-Version v43 kann das freie Messfeld mittels SetFeature/ GetFeature gesetzt/gelesen werden, z.B. SetFeature(FEATURE_FUNCTION_PEAKFILTER_WIDTH, (size_z << 16) + start_z). Um die Einstellungen zu aktivieren, muss eine 0 in das FEATURE_FUNCTION_SHARPNESS-Register geschrieben werden.

```
// Aktiviere freies Messfeld
pLLT->SetFeature(FEATURE_FUNCTION_MEASURINGFIELD, MEASFIELD_ACTIVATE_FREE);

// Messfeldgröße setzen
unsigned short start_z = 20000;
unsigned short size_z = 25000;
unsigned short start_x = 20000;
unsigned short size_x = 25000;

pLLT->SetFreeMeasuringField(start_x, size_x, start_z, size_z);
```

Siehe API: SetFreeMeasuringField()

Alle Startwerte und Größen werden prozentual vom maximalen Wert 65535 angegeben. Bsp: *start_z* = 20000 → 20000/65535 * 100 % = 30,52 %; *start_z* liegt somit bei 30,52% der Matrixhöhe. Bei einem scanCONTROL 29xx mit 1024 Pixeln Matrixhöhe ist dies etwa Pixel 312. Es ist auf die Matrixrotation des jeweiligen Sensortyps zu achten (Überprüfung mit DeveloperDemo möglich).

5.11 Einbaulagenkalibrierung auf den Sensor spielen

Dieses Beispiel zeigt, wie man die Einbaulage kalibriert. Dies ist nützlich, wenn man eine konstant schiefe Einbaulage hat, man das Profil aber gerade ausgeben lassen will. Es kann der Winkel und der Offset kalibriert werden.

```
// Rotationszentrum und Winkel
double center_x = -9; // mm
double center_z = 86.7; // mm
double angle = -45; // °

// Verschiebung Rotationszentrum
double shift_x = 0; // mm
double shift_z = 0; // mm

// Setze Kalibrierung
pLLT->SetCustomCalibration(center_x, center_z, angle, shift_x, shift_z);

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

Siehe API: SetCustomCalibration(), ResetCustomCalibration()

5.12 Containermode zur Weiterverarbeitung mit BV-Tools

Dieses Beispiel zeigt, wie man die Übertragung so konfiguriert, dass Standard-Bildverarbeitungstools direkt mit dem übertragenen Format arbeiten können.

```
#include <math.h>

[…] // Setup und ContainerBuf definieren

unsigned int ProfileCount = 500; // Anzahl der Profile in einem Bild/Container

// Flags für gerade verwendete Auflösung berechnen
unsigned int Res = (unsigned int)floor((log((double)Resolution)*1.0/log(2.0))+0.5);
unsigned int Rearrangement = CONTAINER_DATA_Z | CONTAINER_STRIPE_1 |
                                        CONTAINER_DATA_LSBF | Res << 12);

// Setzen des Rearrangement-Parameters zur Extrahierung von Z-Daten
// (ohne Timestamp) mit der eingestellten Auflösung;
pLLT->SetFeature(FEATURE_FUNCTION_REARRANGEMENT_PROFILE, Rearrangement);

// Containergröße setzen
pLLT->SetProfileContainerSize(Resolution, ProfileCount);

// Puffer reservieren (Z-KOO hat 2 byte)
ContainerBuf.resize(Resolution * 2 * ProfileCount);

// Starten der Profilübertragung
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, true);

[…] // Übertragung

// Stoppen der Profilübertragung
pLLT->TransferProfiles(NORMAL_CONTAINER_MODE, false);
```

```
// Extrahiere/Konvertiere alle Rohdaten zu mm Werten
CInterfaceLLT::ConvertRearrangedContainer2Values(&ContainerBuf[0],
  ContainerBuf.size(), Rearrangement, ProfileCount, LLTType, 0, NULL, NULL, NULL,
                                                    &ValueX[0],  &ValueZ[0]);
```

Siehe API: SetFeature(), SetProfileContainerSize(), TransferProfiles(), FEATURE_FUNCTION_REARRANGEMENT_PROFILE, ConvertRearrangedContainer2Values()
Siehe Operational Manual Part B: *OpManPartB.html#rearrangementprofile*

5.13 Übertragung von partiellen Profilen

Dieses Beispiel zeigt das Einrichten einer Übertragung von partiellen Profilen. Das übertragene Profil entspricht hier der Profilkonfiguration *PURE_PROFILE* mit eingeschränkter Punktezahl, d.h. es werden nur X/Z-Werte von einem definierten Punktebereich übertragen.

```
// Struct zum Definieren des partiellen Profils
TPartialProfile PartialProfile;

[…] // Init

// Setzen des partiellen Profils
PartialProfile.nStartPoint = 20; // Offset 20 -> Startpunkt = Punkt 21
PartialProfile.nStartPointData = 4; // Datenoffset 4 Bytes -> Beginn X-Daten
PartialProfile.nPointCount = m_uiResolution / 2; // Halbe Auflösung
PartialProfile.nPointDataWidth = 4; // 4 Bytes -> X und Z (je 2 Bytes)

// Reservieren des Profilpuffers
ProfileBuffer.resize(PartialProfile.nPointCount * PartialProfile.nPointDataWidth);

// Übergeben des partiellen Profils
pLLT->SetPartialProfile(&PartialProfile);

[…] // Normale Übertragung mit Callback

// Konvertieren des Pufferinhalts in reale Koordinaten
CInterfaceLLT::ConvertPartProfile2Values(&ProfileBuffer[0], ProfileBuffer.size(),
&PartialProfile, LLTType, 0, NULL, NULL, NULL, &ValueX[0], &ValueZ[0], NULL, NULL);
```

Siehe API: SetPartialProfile(), TransferProfiles(), GetActualProfile(), ConvertPartProfile2Values()

Eine Änderung der Profilauflösung mit *setResolution()* muss immer vor dem Setzen der PartialProfile-Konfiguration erfolgen, da der Aufruf von *setResolution()* die PartialProfile-Einstellung zurücksetzt.

5.14 Betrieb von mehreren Sensoren

Dieses Beispiel zeigt, wie in einem Programm mit mehreren Sensoren gearbeitet werden kann.

```
// Erstellen eines Handles für jeden Sensor
CInterfaceLLT pLLT  = new CInterfaceLLT();
CInterfaceLLT pLLT2 = new CInterfaceLLT();

// Suche verfügbare Interfaces
CInterfaceLLT::GetDeviceInterfaces(Interfaces, Interfaces.GetLength(0));

// Setzen der Interfaces
pLLT->SetDeviceInterface(Interfaces[0]);
pLLT2->SetDeviceInterface(Interfaces[1]);
```

```
// Verbinden mit beiden Scannern
pLLT->Connect();
pLLT2->Connect();

[…]

int userData1 = 1;
int userData2 = 2;

// Registrierung Callback Scanner 1
pLLT->RegisterBufferCallback((gpointer)&NewProfile, &userData1);

// Registrierung Callback Scanner 2
pLLT2->RegisterBufferCallback((gpointer)&NewProfile, &userData2);

[…] // Start der Übertragungen äquivalent

// Callback-Funktion mit Unterscheidung der Sensordaten
void NewProfile(const void *pucData, size_t uiSize, gpointer userData)
{
   if (*(int *)userData == 1)
   {
      // Daten Sensor 1
   }

   if (*(int *)userData == 2)
   {
      // Daten Sensor 2
   }
   // Event
}
```

Siehe API: CreateLLTDevice(), GetDeviceInterfacesFast(), SetDeviceInterface(), Connect(), RegisterBufferCallback()

5.15 Fehlermeldungen bei Verbindungsverlust

Dieses Beispiel zeigt, wie ein Callback zur Fehlerbehandlung bei Verbindungsverlust in einer Anwendung zu registrieren ist.

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

void ControlLostCallback(ArvGvDevice *mydevice)
{
   if (mydevice)
   {
      /* Beispielverhalten nach verlorener Verbindung */
      cout << “Control lost!” << endl;
      exit(0);
   }
}
```

Siehe API: RegisterControlLostCallback()

5.16 Temperatur auslesen

Dieses Beispiel zeigt, wie die aktuelle Innentemperatur des Sensors ausgelesen werden kann.

```
unsigned int Temperature = 0;

// Vor Auslesevorgang muss TEMP_PREPARE_VALUE auf das Register geschrieben werden
pLLT->SetFeature(FEATURE_FUNCTION_TEMPERATURE, TEMP_PREPARE_VALUE);

// Auslesen der Temperatur
pLLT->GetFeature(FEATURE_FUNCTION_TEMPERATURE, &Temperature);
```

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

5.17 Packet Delay berechnen und setzen

Dieses Beispiel zeigt die Bestimmung des minimale bzw. maximalen Packet Delays für einen Sensor in einen Netzwerk mit mehreren Sensoren an einem Switch. Der Packet Delay ist abhängig von folgenden Faktoren: Der eingestellten Paketgröße, dem gegebenen Netzwerk, die zu übertragende Datenmenge und die Anzahl der Sensoren. Die Datenmenge setzt sich dabei aus dem eingestellten Übertragungsmodus und der Profilfrequenz zusammen. Der minimale Delay berechnet sich aus:

$$PD_{min} = (Anzahl\ der\ Sensoren - 1) * \frac{Paketgröße}{Netzwerkübertragungsrate}$$

Der maximale Delay ergibt sich aus:

$$PD_{max} = \left(1000 * \frac{\frac{1000}{Profilfrequenz}}{\frac{KByte\ pro\ Profil * 1024}{Paketgröße} + 1} - \frac{Paketgröße}{Netzwerkübertragungsrate} \right) * 0{,}8$$

Der zu konfigurierende Wert kann zwischen diesen Schranken liegen. Jedem betroffenen Sensor muss ein Packet Delay gesetzt werden. Die Scanner testen dann Übertragungsslots an und senden bei einem freien Slot nun die jeweils verzögerten Pakete ab.

Das Setzen des Wertes geschieht wie folgt (hier wird ein Wert von 50 µs eingestellt):

```
unsigned int PacketDelay = 50;

// Setzen des Packet Delays in us
pLLT->SetFeature(FEATURE_FUNCTION_PACKET_DELAY, PacketDelay);
```

Siehe API: SetFeature(), FEATURE_FUNCTION_PACKET_DELAY

6 API

Im Folgenden wird das vollständige API (*Application Program Interface*) aufgelistet. Jede Funktion ist mit ihren Rückgabe- und Parameterwerten beschrieben. Es werden hier größtenteils die Funktionen der C++ API (CInterfaceLLT Klasse) aufgeführt. Für eine C Implementierung ist folgendes zu beachten: Die korrespondierenden Funktionen der C API folgen dem Muster llt_handle->*CppInterfaceFunc([args]*); → *c_api_func(llt_handle, [args]); z.B. hllt->SetFeature(([args]);* → *set_feature(hllt, ([args]);*. In C ist das llt_handle mit *create_llt_device()* zu erstellen (siehe Kommentar in Code Kapitel 5.1) und kann mittels *del_device()* wieder gelöscht werden.

6.1 Auswahl und Initiierungs-Funktionen

- **GetDeviceInterfaces ()**

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

Abrufen der am Rechner verfügbaren scanCONTROL device interfaces. Device interfaces sind die verschiedenen Identifikationsnamen der Sensoren.

Parameter

interfaces	Array für verfügbare Interfaces
size	Größe des Arrays

Rückgabewert

Anzahl der gefundenen device interfaces
Standardfehlerwerte
Spezifische Rückgabewerte:

ERROR_GETDEVINTERFACES_REQUEST_COUNT	-251	Die Größe des übergebenen Feldes ist zu klein
ERROR_GETDEVINTERFACES_INTERNAL	-253	Bei der Abfrage der angeschlossenen scanCONTROL ist ein Fehler aufgetreten

- **InitDevice ()**

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

Initialisieren eines Scanners. Wird bei Aufruf von Connect() schon automatisch durchgeführt.

Parameter

camName	Identifikator des Sensors
devData	Device data des Sensors
pathDevProp	Pfad zu device_properties.dat

Rückgabewert

Standardfehlerwerte
Spezifische Rückgabewerte:

ERROR_DEVPROP_NOT_FOUND	-1300	device_properties.dat (Datei) nicht

		gefunden
ERROR_DEVPROP_DECODE	-1301	Dekodieren der Datei fehlgeschlagen
ERROR_DEVPROP_DEPRECATED	-1302	Dateiversion veraltet oder fehlerhaft ausgelesen
ERROR_DEVPROP_READ_FAILURE	-1303	Lesen der Datei fehlgeschlagen

- **SetPathtoDeviceProperties ()**

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

Setzen des Pfades für die device_properties.dat. Optionaler Aufruf vor Connect; falls gesetzt stehen erweiterte Informationen in MEDeviceData zur Verfügung.

Parameter

pathDevProp Pfad zu device_properties.dat

Rückgabewert

Standardfehlerwerte

Spezifische Rückgabewerte:

ERROR_DEV_PROP_NOT_FOUND	-1300	device_properties.dat (Datei) nicht gefunden

- **SetDeviceInterface ()**

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

Zuweisen eines scanCONTROL device interfaces zu einer Sensorinstanz in der Bibliothek. Ein Interface Identifikator besteht aus dem Sensornamen und der Seriennummer.

Parameter

interface Interface des zu verbindenden scanCONTROL

Rückgabewert

Standardrückgabewerte

Spezifischer Rückgabewert:

ERROR_GETDEVINTERFACES_CONNECTED	-252	scanCONTROL ist verbunden, Disconnect(); aufrufen

6.2 Verbindungs-Funktionen

- **Connect ()**

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

Verbinden mit dem ausgewählten scanCONTROL Sensor. Nur möglich, falls mit *SetDeviceInterface()* ein gültiges device interface zugeordnet wurde. Falls mit

SetPathtoDeviceProperties() eine gültige device_properties.dat geladen wurde, stehen alle Dateiinhalte in MEDeviceData zur Verfügung. Ruft *InitDevice()* auf.

Rückgabewert

Standardrückgabewerte
Alle Rückgabewerte von InitDevice()
Spezifische Rückgabewerte:

ERROR_CONNECT_SELECTED_LLT	-301	Das gewählte Interface ist nicht verfügbar -> ein neues Interface mit *SetDeviceInterface()* wählen
ERROR_CONNECT_ALREADY_CONNECTED	-302	Mit dieser ID ist schon ein scanCONTROL verbunden
ERROR_DEVPROP_NOT_AVAILABLE	-999	Die device_properties.dat ist nicht geladen worden

- **Disconnect ()**

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

Trennen der Verbindung zum scanCONTROL Sensor. Alle eingestellten Parameter bleiben auf dem Scanner erhalten. Die Treibereinstellungen wie *Packetsize*, *Buffer* count und *Profile config* bleiben nicht erhalten.

Rückgabewert

Standardrückgabewerte

6.3 Identifikations-Funktionen

- **GetDeviceName ()**

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

Abfrage des Geräte- und Herstellernamens des scanCONTROL Sensors, sowie die Seriennummer.

Parameter

devName Pointer Namen des Devices
venName Pointer Namen des Herstellers

Rückgabewert

Standardrückgabewerte

- **GetLLTVersion ()**

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

Abfrage der aktuell aufgespielten Firmware version.

Parameter

devVersion Pointer Firmware

Return value

General return values

- **GetLLTType ()**

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

Abfrage des Messbereichs und des Typs des scanCONTROL Sensors.

Parameter

scannerType Scannertyp

Rückgabewert

Standardrückgabewerte

- **GetLLTTypeByName ()**

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

Abfrage des Messbereichs und des Typs des scanCONTROL Sensors mittels dessen Device Name.

Parameter

modelName Name des Sensors (*devName)*
scannerType Scannertyp

Rückgabewert

Standardrückgabewerte

- **ScannerType**

TScannerType	Wert	scanCONTROL Type	Messbereich
StandardType	-1	-	-
scanCONTROL27xx_25	1000	27xx	25 mm
scanCONTROL27xx_100	1001	27xx	100 mm
scanCONTROL27xx_50	1002	27xx	50 mm
scanCONTROL27xx_xxx	1999	27xx	-

scanCONTROL26xx_25	2000	26xx	25 mm
scanCONTROL26xx_50	2002	26xx	50 mm
scanCONTROL26xx_100	2001	26xx	100 mm
scanCONTROL26xx_xxx	2999	26xx	-
scanCONTROL29xx_25	3000	29xx	25 mm
scanCONTROL29xx_50	3002	29xx	50 mm
scanCONTROL29xx_100	3001	29xx	100 mm
scanCONTROL29xx_10	3003	29xx	10 mm
scanCONTROL29xx_xxx	3999	29xx	-

- **GetLLTScalingAndOffset ()**

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

Abfrage des Skalierungs- und des Offsetfaktors des verbundenen scanCONTROL Sensors.

Parameter

scaling Skalierungsfaktor des Sensors
offset Offsetfaktor des Sensors

Rückgabewert

Standardrückgabewerte

- **GetLLTScalingAndOffsetByType ()**

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

Abfrage des Skalierungs- und des Offsetfaktors eines bestimmten Sensortyps.

Parameter

scannerType Scannertyp
scaling Skalierungsfaktor des Sensors
offset Offsetfaktor des Sensors

Rückgabewert

Standardrückgabewerte

6.4 Eigenschafts-Funktionen

6.4.1 Set-/Get-Funktionen

- **GetFeature ()**

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

Auslesen des aktuellen Parameterwertes / Überprüfen der Verfügbarkeit einer Eigenschaft anhand Tabelle in Kapitel 6.4.2.

Parameter

register	Registeradresse der Funktion (FEATURE oder INQUIRY)
value	Ausgelesener Wert

Rückgabewert

Standardrückgabewerte

Spezifischer Rückgabewert:

ERROR_SETGETFUNCTIONS_WRONG_FEATURE_ADRESS	-155	Die Adresse der gewählten Eigenschaft ist falsch

- **SetFeature ()**

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

Setzen des Parameters einer Eigenschaft.

Parameter

register	Registeradresse der Funktion (FEATURE)
value	Zu schreibender Wert

Rückgabewert

Standardrückgabewerte

Spezifischer Rückgabewert:

ERROR_SETGETFUNCTIONS_WRONG_FEATURE_ADRESS	-155	Die Adresse der gewählten Eigenschaft ist falsch

6.4.2 Eigenschaften / Parameter

Im Folgenden werden die Eigenschaftsregister erläutert. INQUIRY-Register dienen zur Überprüfung, ob die jeweilige Funktion vorhanden ist. FEATURE-Register dienen zur Abfrage und Einstellung der Werte. Beide Register definieren das LSB bei Bit 0.

Mithilfe des ausgelesenen Wertes eines **INQUIRY**-Registers kann das Setzen eines Features klassifiziert werden. Dazu liefert das Register den minimal und maximal einstellbaren Wert, ob eine automatische Regelung verfügbar ist und ob die Eigenschaft verfügbar ist:

31	30..26	25	24	23..12	11..0

Eigenschaft verfügbar (1 Bit)	Res. (5 Bit)	Auto (1 Bit)	Res. (1 Bit)	Min Wert (12 Bit)	Max Wert (12 Bit)

Der Wert des **FEATURE**-Registers ist mithilfe des Operation Manual Part B zu interpretieren.

Beispiel: Laserleistung

Feature name	Inquiry address	Status and control address	Default setting
Laser Power	0xfffff0f00524	0xfffff0f00824	0x82000002

Bit	Function
1..0	**Bits 1..0 Laser Power** **0** OFF **1** reduced power **2** full power
11	enable laser pulse mode: the laser is switched on only in the first half of the measurement interval. Additionally the Trigger-Out 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 shutter time < idle time.

Abb. 7: Auszug aus dem Operation Manual Part B

Daraus ergibt sich, dass das für die Laserleistung zu beschreibende Register die Adresse 0xf0f00824 besitzt und mittel den Bits 0 und 1 die Laserleistung geregelt werden kann. Dezimal 0 beschreibt die Einstellung *Laser aus*, $1\,|_{\mathrm{dec}}$ die reduzierte und $2\,|_{\mathrm{dec}}$ die volle Laserleistung. Bit 11 aktiviert den Laserpulsmodus.

- **SERIAL**

CLLTI.FEATURE_FUNCTION_SERIAL	0xf0000410

Auslesen der Seriennummer des verbundenen Sensors. Dieses Register kann nur ausgelesen werden.

- **LASERPOWER**

CLLTI.FEATURE_FUNCTION_LASERPOWER	0xf0f00824
CLLTI.INQUIRY_FUNCTION_LASERPOWER	0xf0f00524

Steuern und Auslesen der Laserleistung: 0 (aus), 1 (reduziert), 2 (voll). Je nach Gerätetyp kann auch die Polarität der externen Laser-Schutzabschaltung oder der Laserpulsmodus eingestellt werden. Das direkt nach der Laserumschaltung übertragene Profil kann korrupt sein. Siehe *OpManPartB.html#laserpower* für den verwendeten Sensortyp.

- **MEASURINGFIELD**

CLLTI.FEATURE_FUNCTION_MEASURINGFIELD	0xf0f00880
CLLTI.INQUIRY_FUNCTION_MEASURINGFIELD	0xf0f00580

Setzen oder Auslesen eines vordefinierten Messfeldes oder Aktivierung der erweiterten Messfeldkonfiguration. Das direkt nach der Messfeldänderung übertragene Profil kann korrupt sein. Siehe *OpManPartB.html#zoom* für den verwendeten Sensortyp. Eine Übersicht über die vordefinierten Messfelder und die damit möglichen maximalen Profilfrequenzen liefert QuickReference.html für den verwendeten Sensortyp.

- **TRIGGER**

`CLLTI.FEATURE_FUNCTION_TRIGGER`	0xf0f00830
`CLLTI.INQUIRY_FUNCTION_TRIGGER`	0xf0f00530

Setzen und Auslesen der Triggerung. Das direkt nach der Änderung der Triggerkonfiguration übertragene Profil kann korrupt sein. Zusammen mit der Triggerfunktion muss meist auch die Trigger-Schnittstelle parametriert werden (siehe *RS422_INTERFACE_FUNCTION*). Durch Änderung der Triggereinstellung wird auch der Profilzähler zurückgesetzt. Siehe *OpManPartB.html#trigger* für den verwendeten Sensortyp.

- **SHUTTERTIME**

`CLLTI.FEATURE_FUNCTION_SHUTTERTIME`	0xf0f0081c
`CLLTI.INQUIRY_FUNCTION_SHUTTERTIME`	0xf0f0051c

Setzen und Auslesen der Belichtungszeit in 10 µs-Schritten. Der Wert kann zwischen 1 und 4095 liegen. Optional kann hier auch die automatische Belichtungsregelung eingestellt werden. Siehe *OpManPartB.html#shutter* für den verwendeten Sensortyp.

- **IDLETIME**

`CLLTI.FEATURE_FUNCTION_IDLETIME`	0xf0f00800
`CLLTI.INQUIRY_FUNCTION_IDLETIME`	0xf0f00500

Setzen und Auslesen der Totzeit zwischen den Belichtungsintervallen in 10 µs-Schritten. Der Wert kann zwischen 1 und 4095 liegen. Ist die automatische Belichtungsregelung aktiv, wird die Totzeit automatisch so angepasst, dass die Profilfrequenz stabil bleibt (*OpManPartB.html#idletime).*

- **PROCESSING_PROFILEDATA**

`CLLTI.FEATURE_FUNCTION_PROCESSING_PROFILEDATA`	0xf0f00804
`CLLTI.INQUIRY_FUNCTION_PROCESSING_PROFILEDATA`	0xf0f00504

Abfragen und Setzen der Einstellung für die Profilverarbeitung, wie z.B. Deaktivieren der Kalibrierung, Spiegelung des Profils, Messdatenverarbeitung (Post-Processing), Reflexionsauswahl oder erweiterte Belichtungseinstellung. Siehe *OpManPartB.html#processingprofile* für den verwendeten Sensortyp.

- **THRESHOLD**

CLLTI.FEATURE_FUNCTION_THRESHOLD	0xf0f00810
CLLTI.INQUIRY_FUNCTION_THRESHOLD	0xf0f00510

Setzen und Auslesen des Schwellwerts für die Messdatenaufnahme. Bei Targets mit mehreren Reflektionen kann das Erhöhen der Schwelle zu besseren Ergebnissen führen. Optional kann hier auch die dynamische Threshold-Regelung aktiviert werden. Siehe *OpManPartB.html#threshold* für den verwendeten Sensortyp.

- **MAINTENANCEFUNCTIONS**

CLLTI.FEATURE_FUNCTION_MAINTENANCEFUNCTIONS	0xf0f0088c
CLLTI.INQUIRY_FUNCTION_ MAINTENANCEFUNCTIONS	0xf0f0058c

Abfragen und Setzen interner Einstellungen, wie z.B. dem Encoderzähler. Siehe *OpManPartB.html#maintenance* für den verwendeten Sensortyp.

- **ANALOGFREQUENCY**

CLLTI.FEATURE_FUNCTION_ANALOGFREQUENCY	0xf0f00828
CLLTI.INQUIRY_FUNCTION_ANALOGFREQUENCY	0xf0f00528

Analogfrequenz für die Analogausgänge der scanCONTROL 28xx-Serie. Die Frequenz kann zwischen 0 und 150 eingestellt werden, wobei der Zählwert der Frequenz in kHz entspricht. Bei einer Einstellung von 0 kHz wird der Analogausgang abgeschaltet, was bei Profilfrequenzen größer 500 Hz empfehlenswert ist, um einen Überlauf bei der Analogausgabe zu vermeiden. Siehe *OpManPartB.html#focus* für den scanCONTROL 28xx.

- **ANALOGOUTPUTMODES**

CLLTI.FEATURE_FUNCTION_ANALOGOUTPUTMODES	0xf0f00820
CLLTI.INQUIRY_FUNCTION_ANALOGOUTPUTMODES	0xf0f00520

Modes für die Analogausgänge der scanCONTROL 28xx-Serie. Einstellen der Analog output modes. Es können z.B. die Spannungsbereiche und die Polarität der analogen Ausgänge umgeschaltet werden. Siehe *OpManPartB.html#gain* für den scanCONTROL 28xx.

- **CMMTRIGGER**

CLLTI.FEATURE_FUNCTION_CMMTRIGGER	0xf0f00888
CLLTI.INQUIRY_FUNCTION_CMMTRIGGER	0xf0f00588

Konfiguration der optionalen CMM-Trigger-Funktionen. Die Konfiguration des CMM-Triggers erfolgt durch mehrere Schreibzugriffe auf dieses Register. Zurückgelesen werden kann nur der zuletzt geschriebene Wert. Siehe *OpManPartB.html#cmmtrigger* für den verwendeten Sensortyp.

- **REARRANGEMENT_PROFILE**

CLLTI.FEATURE_FUNCTION_REARRANGEMENT_PROFILE	0xf0f0080c
CLLTI.INQUIRY_FUNCTION_REARRANGEMENT_PROFILE	0xf0f0050c

Parametrierung der übertragenen Profilinformationen im Container-Mode. Siehe *OpManPartB.html#rearrangementprofile* für den verwendeten Sensortyp.

- **PROFILE_FILTER**

CLLTI.FEATURE_FUNCTION_PROFILE_FILTER	0xf0f00818
CLLTI.INQUIRY_FUNCTION_PROFILE_FILTER	0xf0f00518

Anwendung von Resampling, Median-Filter und/oder Average-Filter. Siehe *OpManPartB.html#profilefilter* für den verwendeten Sensortyp.

- **RS422_INTERFACE _FUNCTION**

CLLTI.FEATURE_FUNCTION_RS422_INTERFACE_FUNCTION	0xf0f008c0
CLLTI.INQUIRY_FUNCTION_RS422_INTERFACE_FUNCTION	0xf0f005c0

Parameter für die Modi-Einstellung der RS422-Schnittstelle bzw. den digitalen Schnittstellen. Siehe *OpManPartB.html#ioconfig bzw. OpManPartB.html#capturesize* für den verwendeten Sensortyp.

- **PACKET_DELAY**

CLLTI.FEATURE_FUNCTION_PACKET_DELAY	0x00000d08

Ethernet-Paketverzögerung für den Betrieb mehrerer Sensoren an einem Switch in µs. Der einzustellende Wert kann zwischen 0 und 1000 µs liegen.

- **TEMPERATURE**

CLLTI.FEATURE_FUNCTION_TEMPERATURE	0xf0f0082c
CLLTI.INQUIRY_FUNCTION_TEMPERATURE	0xf0f0052c

Auslesen der Sensortemperatur in 0,1 K-Schritten. Bevor die aktuelle Temperatur ausgelesen werden kann, muss erst 0x86000000 auf das Feature-Register geschrieben werden. (*OpManPartB.html#temperature*)

- **SHARPNESS**

CLLTI.FEATURE_FUNCTION_SHARPNESS	0xf0f00808
CLLTI.INQUIRY_FUNCTION_SHARPNESS	0xf0f00508

Einstellungen für Peak-Filter, frei definierbares Messfeld und Einbaulagenkalibrierung. Die Konfiguration erfolgt durch mehrere Schreibzugriffe auf dieses Register. Zurückgelesen werden kann nur der zuletzt geschriebene Wert. Seit DLL Version 3.7 / Sensor-Firmware v43, hat dieses Register hauptsächlich die Funktion einige gesetzte Registerwerte zu aktivieren. Siehe *OpManPartB.html#extraparameter* für den verwendeten Sensortyp.

- **FEATURE_FUNCTION_FREE_MEASURINGFIELD**

CLLTI.FEATURE_FUNCTION_FREE_MEASURINGFIELD_X	0xf0b0200c
CLLTI.FEATURE_FUNCTION_FREE_MEASURINGFIELD_Z	0xf0b02008

Setzt Start und Größe in X und Z des freien Messfeldes. Die möglichen Werte reichen von 0 bis 65535. Die Matrixrotation der Senoren ist dabei zu beachten. Aktiviert wird die Einstellung mittels des Sharpness-Registers (FEATURE_FUNCTION_SHARPNESS).

- **FEATURE_FUNCTION_PEAKFILTER**

CLLTI.FEATURE_FUNCTION_PEAKFILTER_WIDTH	0xf0b02000
CLLTI.FEATURE_FUNCTION_PEAKFILTER_HEIGHT	0xf0b02004

Setzt die minimal und maximal für eine Reflexion zulässige Intensität bzw. Reflexionsweite. Die Werte reichen von 0 bis 1023.

- **FEATURE_FUNCTION_DYNAMIC_TRACK**

CLLTI.FEATURE_FUNCTION_DYNAMIC_TRACK_DIVISOR	0xf0b02010

CLLTI.FEATURE_FUNCTION_DYNAMIC_TRACK_FACTOR	0xf0b02014

Setzt die encoderbasierte Messfeldnachverfolgung.

- **FEATURE_FUNCTION_CALIBRATION**

CLLTI.FEATURE_FUNCTION_CALIBRATION_0 - 7	0xf0b02020 - 0xf0b0203c

Setzt Parameter der Einbaulagenkalibrierung.

6.5 Spezielle Eigenschafts-Funktionen

6.5.1 Software Trigger

- **TriggerProfile ()**

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

Ausführen einer Software Triggerung bei Aufruf.

Rückgabewert
Standardrückgabewerte

6.5.2 Profilkonfiguration

- **GetProfileConfig ()**

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

Abfrage der aktuellen Profilkonfiguration.

Parameter
profileConf Ausgelesene eingestellte Profilkonfiguration

Rückgabewert
Standardrückgabewerte

- **SetProfileConfig ()**

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

Setzen der Profilkonfiguration.

Parameter

profileConf Zu setzende Profilkonfiguration

Rückgabewert

Standardrückgabewerte

Spezifischer *Rückgabewert*:

ERROR_ SETGETFUNCTIONS_WRONG _PROFILE_CONFIG	-152	Die gewünschte Profilkonfiguration steht nicht zur Verfügung

- **ProfileConfig**

Zur Verfügung stehende *ProfileConfig*-Einstellungen.

Konstante für den Rückgabewert	Wert	Beschreibung
PROFILE	1	Profildaten aller vier Streifen
VIDEO_IMAGE	2	Matrix Videobild
PARTIAL_PROFILE	5	Partielles Profil welches per SetPartialProfile eingeschränkt wurde
CONTAINER	6	Container-Daten

6.5.3 Profilauflösung / Punkte pro Profil

- **GetResolution ()**

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

Abfrage der aktuellen Profilauflösung bzw. Messpunkte pro Profil.

Parameter

value Ausgelesene eingestellte Profilauflösung

Rückgabewert

Standardrückgabewerte

- **SetResolution ()**

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

Setzen der Profilauflösung bzw. Messpunkte pro Profil. Die Auflösung kann nur dann geändert werden, wenn keine Profile übertragen werden. Außerdem werden bei *SetResolution()* alle Einstellungen für das *PartialProfile* gelöscht.

Parameter

value Zu setzende Profilauflösung

Rückgabewert

Standardrückgabewerte

Spezifischer Rückgabewert:

ERROR_ SETGETFUNCTIONS_NOT _SUPPORTED_RESOLUTION	-153	Die gewünschte Auflösung wird nicht unterstützt

- **GetResolutions ()**

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

Abfrage der zur Verfügung stehenden Profilauflösungen.

Parameter

value	Array mit verfügbaren Profilauflösungen
size	Größe des übergebenen Arrays

Rückgabewert

Anzahl der verfügbaren Auslösungen

Spezifischer Rückgabewert:

ERROR_ SETGETFUNCTIONS_NOT _SUPPORTED_RESOLUTION	-156	Die Größe des übergebenen Feldes ist zu klein

6.5.4 Container-Größe

- **GetProfileContainerSize ()**

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

Abfrage der aktuellen Container-Größe.

Parameter

width	Ausgelesene eingestellte Containerbreite
height	Ausgelesene eingestellte Containerhöhe

Rückgabewert

Standardrückgabewerte

- **SetProfileContainerSize ()**

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

Setzen der Container-Größe. Die Höhe kann frei zwischen 0 und der maximal möglichen Höhe gewählt werden und entspricht der Anzahl von Profilen, die in dem Container übertragen werden. Die Container-Höhe sollte nicht höher als die dreifache Profilrate sein. Ist „Verbinden von aufeinanderfolgenden Profilen“ aktiviert, muss die Höhe * Breite eines

Bildes ein ganzzahliges Vielfaches von 16384 sein. Wird versucht einen anderen Höhenwert einzustellen, wird die Höhe automatisch auf den nächsten passenden Wert gesetzt.

Parameter

width Zu setzende Containerbreite
height Zu setzende Containerhöhe

Rückgabewert

Standardrückgabewerte
Spezifische Rückgabewerte:

ERROR_SETGETFUNCTIONS_WRONG_PROFILE_SIZE	-157	Die Größe für den Container ist falsch
ERROR_SETGETFUNCTIONS_MOD_4	-158	Die Container-Breite ist nicht durch 4 teilbar

- **GetMaxProfileContainerSize ()**

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

Abfrage der maximal möglichen Container-Größe. Ist die maximale Breite 64, so wird der Container-Mode nicht von dem scanCONTROL unterstützt.

Parameter

maxWidth Maximale einstellbare Containerbreite
maxHeight Maximale einstellbare Containerhöhe

Rückgabewert

Standardrückgabewerte

6.5.5 Vorgehaltene Puffer für das Profile-Polling

- **GetHoldBuffersForPolling ()**

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

Abfrage der Anzahl der vorgehaltenen Puffer für das Abholen mit *GetActualProfile()*.

Parameter

holdBuffersForPolling Zu setzende Pufferanzahl für Polling

Rückgabewert

Standardrückgabewerte

- **SetHoldBuffersForPolling ()**

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

Setzen der Anzahl der vorgehaltenen Puffer für das Abholen von Profilen/Containern mit *GetActualProfile().* Der Puffer arbeitet nach dem FIFO-Prinzip. Je größer die Anzahl ist, desto mehr Profile werden zwischengespeichert und die Häufigkeit von Profilausfällen beim Abholen mit *GetActualProfile()* wird verringert. Die Anzahl kann maximal halb so groß wie die Anzahl der Puffer im Treiber sein.

Parameter

holdBuffersForPolling Ausgelesene Pufferanzahl für Polling

Rückgabewert

Standardrückgabewerte

Spezifischer Rückgabewert:

ERROR_SETGETFUNCTIONS_WRONG_BUFFER_COUNT	-150	Die Anzahl der gewünschten Puffer liegt nicht im Bereich >=2 und <= 200

6.5.6 Anzahl der Puffer

Eine hohe Pufferanzahl ist bei sehr hohen Profilfrequenzen, langsamen Rechnern und/oder Rechnern bei denen mehrere Programme im Hintergrund laufen sinnvoll. Bei Container-Mode- oder Videobildübertragungen sind max. 4 Puffer sinnvoll.

- **GetBufferCount ()**

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

Abfrage der Anzahl der Puffer im Treiber für die Datenübertragung.

Parameter

value Ausgelesene Pufferanzahl

Rückgabewert

Standardrückgabewerte

- **SetBufferCount ()**

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

Setzen der Anzahl der Puffer im Treiber für die Datenübertragung.

Parameter

value Zu setzende Pufferanzahl

Rückgabewert

Standardrückgabewerte
Spezifischer Rückgabewert:

ERROR_SETGETFUNCTIONS_WRONG_BUFFER_COUNT	-150	Die Anzahl der gewünschten Puffer liegt nicht im Bereich >=2 und <= 200

6.5.7 Paketgröße

Vom scanCONTROL werden die Paketgrößen 128, 256, 512, 1024, 2048 und 4096 Bytes unterstützt. Pakete größer als 1024 Bytes erfordern bei Ethernet die Unterstützung von Jumbo Frames durch die gesamte Übertragungsstrecke, insbesondere der empfangenden Netzwerkkarte.

- **GetPacketSize ()**

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

Abfrage der aktuellen Paketgröße für die Größe der Ethernet-Streaming-Pakete.

Parameter
value Ausgelesene Paketgröße

Rückgabewert
Standardrückgabewerte

- **SetPacketSize ()**

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

Setzen der aktuellen Paketgröße für die Größe der Ethernet Streaming Pakete. Diese Paketgröße muss zwischen der minimalen und maximalen Paketgröße liegen.

Parameter
value Zu setzende Paketgröße

Rückgabewert
Standardrückgabewerte
Spezifischer Rückgabewert:

ERROR_SETGETFUNCTIONS_PACKET_SIZE	-151	Die gewünschte Paketgröße wird nicht unterstützt

- **GetMinMaxPacketSize ()**

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

Abfragen der minimalen und maximalen Paketgröße für die isochrone Profilübertragung der Ethernet Streaming Pakete.

Parameter

minPacketSize	Minimal einstellbare Paketgröße
maxPacketSize	Maximal einstellbare Paketgröße

Rückgabewert

Standardrückgabewerte

6.5.8 Timeout für die Kommunikationsüberwachung zum Sensor

Setzen und Auslesen des Heartbeat Timeouts in Millisekunden zur Überwachung der Kommunikations-Schnittstelle zwischen linllt und dem scanCONTROL. Der eigentliche Timeout-Wert liegt dreimal höher als der eingestellte Heartbeat Timeout. Läuft der Timeout ohne den Heartbeat ab, wird die Kommunikation automatisch vom Sensor aus abgebrochen. Beim Debuggen einer programmierten Anwendung ist oftmals ein zu klein gesetzter Heartbeat-Timeout die Ursache für Verbindungsabbrüche.

- **GetEthernetHeartbeatTimeout ()**

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

Abfrage des eingestellten Verbindungs-Timeouts.

Parameter

CInterfaceLLT	LLT-Klasse
pValue	Ausgelesener Heartbeat Timeout

Rückgabewert

Standardrückgabewerte

- **SetEthernetHeartbeatTimeout ()**

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

Setzen des Verbindungs-Timeouts in ms. Der Heartbeat-Timeout kann zwischen 500 und 1.000.000.000 ms liegen.

Parameter

CInterfaceLLT	LLT-Klasse
timeout	Zu setzender Heartbeat Timeout

Rückgabewert

Standardrückgabewerte
Spezifischer Rückgabewert:

ERROR_SETGETFUNCTIONS_HEARTBEAT_TOO_HIGH	-162	Der Parameter für den Heartbeat Timeout ist zu groß

6.5.9 Laden und Speichern von Parametersätzen

In einem Usermode können alle Einstellungen eines scanCONTROL gespeichert werden, so dass nach einem Reset oder Neustart sofort alle Einstellungen wieder aktiv sind. Dies ist vor allem bei Postprocessing-Anwendungen sinnvoll. Das Laden der Usermodes kann nicht während einer aktiven Profil/Container-Übertragung durchgeführt werden. Usermode 0 kann nur geladen (und damit nicht beschrieben) werden, da er die Standardeinstellungen enthält.

- **GetActualUserMode ()**

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

Abfrage des zuletzt geladenen User-Modes/Parametersatzes. Die scanCONTROL 27xx-, 26xx- und 29xx-Serien unterstützen 16 Usermodes.

Parameter

actualUserMode	Momentan geladener Usermode
userModeCount	Insgesamt verfügbare Usermodes

Rückgabewert

Standardrückgabewerte

- **ReadWriteUserModes ()**

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

Laden oder Speichern eines User-Modes/Parametersatzes. Ist *write* *false*, wird der mit *userMode* angegebene Usermode geladen, ansonsten werden die aktuellen Einstellungen unter diesem Usermode gespeichert. Nach dem Laden eines User Modes wird ein Reconnect mit dem Sensor benötigt.

Parameter

write	Laden (false) oder Schreiben (true) eines Usermodes
userMode	Zu ladender bzw. schreibender Usermode

Rückgabewert

Standardrückgabewerte
Spezifische Rückgabewerte:

ERROR_SETGETFUNCTIONS_USER_MODE_TOO_HIGH	-160	Die angegebene Usermode-Nummer steht nicht zur Verfügung
ERROR_SETGETFUNCTIONS_USER_MODE_FACTORY_DEFAULT	-161	Usermode 0 kann nicht überschrieben werden (Standardeinstellungen)

6.6 Registrierungs-Funktionen

6.6.1 Registrieren des Callbacks für Profilübertragung

Nach der Registrierung eines Callbacks wird dieser beim Empfang eines Profils/Containers aufgerufen. Die Callback-Funktion selbst besitzt als Parameter einen Pointer auf die Profil-/Container-Daten, die dazugehörige Größe des Datenfeldes und einen userData-Parameter.
Der Callback ist für die Verarbeitung von Profilen/Containern mit einer hohen Profilfrequenz gedacht. Innerhalb des Callback können die Profile/Container in einen Puffer für eine spätere oder zum Callback synchrone oder asynchrone Verarbeitung kopiert werden. Eine Verarbeitung innerhalb des Callbacks ist nicht zu empfehlen, da für die Zeit, die der Callback zur Verarbeitung benötigt, der Treiber keine neuen Profile/Container abholen kann. Unter Umständen kann es dadurch zu Profil-/Container-Ausfällen kommen.
Die Profil-/Container-Daten in dem vom Callback übergebenen Puffer dürfen nicht verändert werden.

- **RegisterBufferCallback ()**

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

Registrieren des Callback, der bei Profilankunft aufgerufen wird.

Parameter

bufferCb	Pointer zur Callback-Funktion
userData	Beliebige Daten, die im Callback zur Verfügung stehen sollen

Rückgabewert

Standardrückgabewerte

6.6.2 Registrieren einer Fehlermeldung, die bei Fehlern gesendet wird

- **RegisterControlLostCallback ()**

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

Registrieren einer Funktion zur Behandlung eines Verbindungsverlustes.

Parameter

controlLostCb	Pointer zur ControlLost-Funktion
userData	Beliebige Daten, die im Callback zur Verfügung stehen sollen

Rückgabewert

Standardrückgabewerte

6.7 Profilübertragungs-Funktionen

6.7.1 Profilübertragung

- **TransferProfiles ()**

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

Starten oder stoppen der Profilübertragung. Nach dem Starten einer Übertragung kann es bis zu 100 ms dauern, ehe die ersten Profile/Container per Callback abgeholt werden können. Wird eine Übertragung beendet, wartet die Funktion automatisch, bis der Treiber alle Puffer zurückgegeben hat.

Parameter

transferProfileType	Profilübertragungstyp
enable	Starten (true) oder Stoppen (false) der Übertragung

Rückgabewert

Standardrückgabewerte

- **TTransferProfileType**

Zur Verfügung stehende TTransferProfileTypes:

Name	Wert	Beschreibung
NORMAL_TRANSFER	0	Aktivieren einer kontinuierlichen Übertragung von Profilen
NORMAL_CONTAINER_MODE	2	Aktivieren einer kontinuierlichen Übertragung im Container-Mode

- **SetStreamNiceValue ()**

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

Setzt den gewünschten nice Wert des stream threads (19 bis -20). Kleinere Werte geben dem Thread eine höhere Priorität, erfordern aber ggfls. erweiterte Benutzerrechte. (=arv_make_thread_high_priority)

Parameter

niceValue	nice Wert

Rückgabewert

Standardrückgabewerte

- **GetStreamNiceValue ()**

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

Liest den aktuell gesetzten nice Wert zurück.

Parameter
niceValue nice Wert

Rückgabewert
Standardrückgabewerte

- **SetStreamPriority ()**

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

Setzt den gewünschten (Realtime-)Prioritätswert des stream threads (0 bis 99). Höhere Werte geben dem Thread eine höhere Priorität, erfordern aber ggfls. erweiterte Benutzerrechte. (=arv_make_thread_realtime)

Parameter
priority Priorität

Rückgabewert
Standardrückgabewerte

- **GetStreamPriority ()**

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

Liest den aktuell gesetzten (Realtime-)Prioritätswert zurück.

Parameter
priority Priorität

Rückgabewert
Standardrückgabewerte

- **GetStreamPriorityState ()**

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

Liest den aktuell gesetzten Prioritätswert zurück.

Parameter

prioState Prioritätsstatus

Rückgabewert

Standardrückgabewerte

- **TStreamPriorityState**

Zur Verfügung stehende TStreamPriorityStates:

Name	Wert	Beschreibung
PRIO_NOT_SET	0	Keine Priorität gesetzt. Nutze Systemstandard.
PRIO_SET_SUCCESS	1	Gewünschte Priorität gesetzt
PRIO_SET_RT_FAILED	2	Prioritätswert konnte nicht gesetzt werden. Nutzerrechte überprüfen. (arv_make_thread_realtime)
PRIO_SET_NICE_FAILED	3	Nicewert konnte nicht gesetzt werden. Nutzerrechte überprüfen. (arv_make_thread_high_priority)
PRIO_SET_FAILED	4	Prioritätswerte konnte nicht gesetzt werden. Nutzerrechte überprüfen.

- **GetStreamStatistics ()**

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

Abfrage der Übertragungsstatistik. Die Übertragung muss beim Abfragezeitpunkt laufen.

Parameter

completedBuffer Erfolgreich komplettierte Puffer
failures Fehlerhafte Übertragung
underruns Buffer underruns

Rückgabewert

Standardrückgabewerte

6.7.2 Abholen des aktuellen Profils/Containers/Video-Bildes

- **GetActualProfile ()**

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

Abholen des aktuellen Profils/Containers/Video-Bildes vom holding buffer.

Parameter

buffer	Übertragungspuffer
bufferSize	Übertragungspuffergröße
profileConfig	Profilkonfiguration der Übertragung
lostProfiles	Verlorene Profile

Rückgabewert

Anzahl der in den Puffer kopierten Bytes
Standardfehlerwerte
Spezifische Rückgabewerte:

ERROR_PROFTRANS_WRONG_PROFILE_CONFIG	-102	Das geladene Profil kann nicht in die gewünschte Profilkonfiguration konvertieren werden
ERROR_PROFTRANS_NO_NEW_PROFILE	-104	Es ist seit dem letzten Aufruf von GetActualProfile kein neues Profil angekommen
ERROR_PROFTRANS_BUFFER_SIZE_TOO_LOW	-105	Die Puffergröße des übergebenen Puffers ist zu klein
ERROR_PROFTRANS_NO_PROFILE_TRANSFER	-106	Die Profilübertragung ist nicht gestartet und es wird keine Datei geladen

6.7.3 Konvertieren von Profildaten

- **ConvertProfile2Values ()**

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

Extrahieren und Konvertieren von Profil-Daten in Koordinaten und erweiterte Punktinformationen. Die übergebenen Arrays müssen mindestens die Größe der Auflösung (Punkte pro Profil) besitzen.

Parameter

buffer	Profilpuffer
bufferSize	Größe des Profilpuffers
resolution	Aktuelle Profilauflösung
profileConfig	Profile Config (muss PROFILE sein)
scannerType	Sensortyp
reflection	Auszuwertender Profilstreifen
width	Array für ausgelesene Punktweiten
intensity	Array für ausgelesene Maximalintensitäten
threshold	Array für ausgelesene Thresholds
x	Array für ausgelesene Positionswerte
z	Array für ausgelesene Abstandswerte
m0	Array für ausgelesenes Moment 0
m1	Array für ausgelesenes Moment 1

Rückgabewert

Standardrückgabewerte
Zusätzliche Rückgabewerte bei Erfolg

Spezifischer Rückgabewert:

ERROR_PROFTRANS_REFLECTION_NUMBER_TOO_HIGH	-110	Die Anzahl der gewünschten Streifen ist größer 3

- **ConvertPartProfile2Values ()**

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

Extrahieren und Konvertieren von partiellen Profil-Daten in Koordinaten und erweiterte Punktinformationen. Die übergebenen Arrays müssen mindestens die Größe des PointCounts bei *PARTIAL_PROFILE* besitzen.

Parameter

buffer	Profilpuffer
bufferSize	Größe des Profilpuffers
partialProfile	Partielles Profil
scannerType	Sensortyp
reflection	Auszuwertender Profilstreifen
width	Array für ausgelesene Punktweiten
intensity	Array für ausgelesene Maximalintensitäten
threshold	Array für ausgelesene Thresholds
x	Array für ausgelesene Positionswerte
z	Array für ausgelesene Abstandswerte
m0	Array für ausgelesenes Moment 0
m1	Array für ausgelesenes Moment 1

Rückgabewert

Standardrückgabewerte

Zusätzliche Rückgabewerte bei Erfolg

Spezifischer Rückgabewert:

ERROR_PROFTRANS_REFLECTION_NUMBER_TOO_HIGH	-110	Die Nummer Anzahl der gewünschten Streifen ist größer 3

- **ConvertRearrangendContainer2Values ()**

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

Extrahieren und Konvertieren von rearranged Containerdaten in Koordinaten und erweiterte Punktinformationen. Dabei wird der komplette Container in die entsprechenden Daten

umgewandelt, d.h. mehr als ein Profil. Die übergebenen Arrays für X, Z, … müssen Auflösung*Anzahl der Profile groß sein.

Parameter

buffer	Profilpuffer
bufferSize	Größe des Profilpuffers
rearrangement	Wert des Rearrangement Registers
numberProfiles	Anzahl der Profile im Container
scannerType	Sensortyp
reflection	Auszuwertender Profilstreifen
width	Array für ausgelesene Punktweiten
intensity	Array für ausgelesene Maximalintensitäten
threshold	Array für ausgelesene Thresholds
x	Array für ausgelesene Positionswerte
z	Array für ausgelesene Abstandswerte

Rückgabewert

Standardrückgabewerte
Zusätzliche Rückgabewerte bei Erfolg
Spezifischer Rückgabewert:

ERROR_PROFTRANS_REFLECTION_NUMBER_TOO_HIGH	-110	Die Anzahl der gewünschten Streifen ist größer 3

- **Rückgabewerte bei Erfolg**

War der Rückgabewert >0, beschreiben die einzelnen Bits wie die Arrays gefüllt wurden:

Gesetztes Bit	Konstante	Beschreibung
8	CONVERT_WIDTH	Das Array für die Reflektionsbreite wurde mit Daten gefüllt
9	CONVERT_MAXIMUM	Das Array für die maximalen Intensitäten wurde mit Daten gefüllt
10	CONVERT_THRESHOLD	Das Array für die Thresholds wurde mit Daten gefüllt
11	CONVERT_X	Das Array für die Positions-Koordinaten wurde mit Daten gefüllt
12	CONVERT_Z	Das Array für die Abstands-Koordinaten wurde mit Daten gefüllt
13	CONVERT_M0	Das Array für die M0s wurde mit Daten gefüllt
14	CONVERT_M1	Das Array für die M1s wurde mit Daten gefüllt

6.8 Funktionen zur Übertragung von partiellen Profilen

Das Messsystem bietet die Möglichkeit das zu übertragende Profil flexibel einzuschränken. Der Vorteil von diesem Verfahren ist eine geringere Größe der tatsächlich übertragenen Daten. Außerdem können damit nicht benötigte Bereiche eines Profils schon direkt im scanCONTROL verworfen werden.

- **GetPartialProfile ()**

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

Abfrage der Parameter für die Übertragung von partiellen Profilen.

Parameter

partialProfile Referenz auf partielle Profilstruktur

Rückgabewert

Standardrückgabewerte

Spezifische Rückgabewerte:

ERROR_PARTPROFILE_NO_PART_PROF	-350	Die Profilkonfiguration ist nicht auf PARTIAL_PROFILE eingestellt -> SetProfileConfig(PARTIAL_PROFILE); aufrufen

- **SetPartialProfile ()**

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

Setzen der Parameter für die Übertragung von partiellen Profilen. Alle Parameter der *SetPartialProfile()* Funktion müssen immer ein ganzzahliges Vielfaches der jeweiligen *UnitSize* der Funktion *GetPartialProfileUnitSize()* sein.

Parameter

partialProfile Referenz auf zu setzende partielle Profilvariable

Rückgabewert

Standardrückgabewerte

Spezifische Rückgabewerte:

ERROR_PARTPROFILE_NO_PART_PROF	-350	Die Profilkonfiguration ist nicht auf PARTIAL_PROFILE eingestellt -> *SetProfileConfig(PARTIAL_PROFILE);* aufrufen
ERROR_PARTPROFILE_TOO_MUCH_BYTES	-351	Die Anzahl der Bytes pro Punkt ist zu hoch -> nStartPointData oder nPointDataWidth ändern
ERROR_PARTPROFILE_TOO_MUCH_POINTS	-352	Die Anzahl der Punkte ist zu hoch -> nStartPoint oder nPointCount ändern
ERROR_PARTPROFILE_NO_POINT_COUNT	-353	nPointCount oder nPointDataWidth ist 0
ERROR_PARTPROFILE_NOT_MOD_UNITSIZE_POINT	-354	nStartPoint oder nPointCount sind kein Vielfaches von nUnitSizePoint
ERROR_PARTPROFILE_NOT_MOD_UNITSIZE_DATA	-355	nStartPointData oder PointDataWidth sind kein Vielfaches von nUnitSizePointData

- **GetPartialProfileUnitSize ()**

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

Abfrage der verfügbaren Schrittweiten zur Übertragung von partiellen Profilen.

Parameter

unitSizePoint	Ausgelesene UnitSizePoint-Größe
unitSizePointData	Ausgelesene UnitSizePointData-Größe

Rückgabewert

Standardrückgabewerte

6.9 Funktionen zur Extrahierung der Timestamp-Informationen

- **Timestamp2TimeAndCount ()**

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

Extrahieren der Belichtungsinformationen und des Profilzählers aus dem Timestamp.

Parameter

buffer	Referenz auf Timestamp-Bytes des Profilpuffers
shutterOpen	Ausgelesene Startzeit der Belichtung
shutterClosed	Ausgelesene Endzeit der Belichtung
profileCount	Ausgelesener Profilzähler
encTimes2OrDigIn	2x Wert Encoderzähler oder Status digitale Eingänge (binär)

Rückgabewert

Standardrückgabewerte

6.10 Kalibrierung der Einbaulage

- **SetCustomCalibration ()**

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

Kalibrieren der Sensoreinbaulage durch Rotieren und Verschieben des Profils.

Parameter

cX	Rotationszentrum X in mm
cZ	Rotationszentrum Z in mm

angle	Rotationswinkel in Grad
sX	Verschiebung des Rotationszentrums X in mm
sZ	Verschiebung des Rotationszentrums Z in mm

Rückgabewert

Standardrückgabewerte
Wie SetFeature()

- **ResetCustomCalibration ()**

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

Zurücksetzen der Sensor-Einbaulagenkalibrierung.

Rückgabewert

Standardrückgabewerte
Wie SetFeature()

6.11 Funktionen für das Post-Processing

Das Post-Processing stellt gewisse Module auf dem Sensor zur Verfügung, um Profile auszuwerten. Diese Module stehen nur für scanCONTROL SMART- oder gapCONTROL-Sensoren zur Verfügung.

- **ReadPostProcessingParameter ()**

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

Auslesen der Post-Processing-Parameter.

Parameter

CInterfaceLLT	LLT-Klasse
parameter	Pointer auf Post-Processing-Parameter-Array
size	Größe des Post-Processing-Parameter-Arrays (1024 UINT32)

Rückgabewert

Standardrückgabewerte

- **WritePostProcessingParameter ()**

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

Schreiben der Post-Processing-Parameter.

Parameter

CInterfaceLLT	LLT-Klasse
parameter	Pointer auf Post-Processing-Parameter-Array
size	Größe des Post-Processing-Parameter-Arrays (1024 UINT32)

Rückgabewert

Standardrückgabewerte

6.12 Sonstiges

- **SetPeakFilter ()**

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

Setzt die sog. Peakfilter, mit denen Charakteristika für gültige Profilpunkte eingegrenzt werden können.

Parameter

minWidth	Min. zulässige Reflexionsbreite
maxWidth	Max. zulässige Reflexionsbreite
minIntensity	Min. zulässige Intensität
maxIntensity	Max. zulässige Intensität

Rückgabewert

Standardrückgabewerte
Wie SetFeature()

- **SetFreeMeasuringField ()**

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

Setzt das Messfeld frei mittels Start- und Größenwerten. Alle Startwerte und Größen werden prozentual vom maximalen Wert 65535 angegeben. Bsp: *start_z* = 20000 → 20000/65535 * 100 % = 30,52 %; *start_z* liegt somit bei 30,52% der Matrixhöhe. Bei einem scanCONTROL 29xx mit 1024 Pixeln Matrixhöhe ist dies etwa Pixel 312. Es ist auf die Matrixrotation des jeweiligen Sensortyps zu achten (Überprüfung mit DeveloperDemo möglich).

Parameter

startX	Startwert X
sizeX	Größe in X
startZ	Startwert Z
sizeZ	Größe in Z

Rückgabewert

Standardrückgabewerte
Wie SetFeature()

- **SetDynamicMeasuringFieldTracking ()**

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

Setzt dynamisches Encoder-gesteuertes Messfeld.

Parameter

divX Verfahrsteps in X
divZ Verfahrsteps in Z
shiftX Verschiebung X
shiftZ Verschiebung Z

Rückgabewert

Standardrückgabewerte
Wie SetFeature()

6.13 Event handling (for Win-Kompatibilität)

- **CreateEvent ()**

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

Erzeugt Event handle.

Rückgabewert

Event handle oder Fehler

- **FreeEvent ()**

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

Löscht den Event handle

Parameter

eventHandle Event handle des zu löschenden Events

- **SetEvent ()**

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

Setzt Event.

Parameter

eventHandle Event handle des zu setzenden Events

- **ResetEvent ()**

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

Setzt Event zurück.

Parameter

eventHandle Event handle des rückzusetzenden Events

- **WaitForSingleObject ()**

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

Blockt bis der Event gesetzt wird oder der Timeout abgelaufen ist.

Parameter

eventHandle Event handle des rückzusetzenden Events
timeout Timeout in ms

Rückgabewert

0 bei Erfolg

ERROR_TRANSERRORVALUE_BUFFER_SIZE_TO_LOW	-451	Die Größe des übergebenen Puffers ist für den String zu klein

6.14 Konfiguration lesen/speichern

- **ExportLLTConfig ()**

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

Auslesen aller Parameter und speichern in eine Datei. Diese Konfigurations-Datei enthält alle relevanten Parameter und ist vor allem für Postprocessing-Anwendungen gedacht. Das Dateiformat entspricht dem Kommunikations-Protokoll für die serielle Verbindung mit dem scanCONTROL und kann daher ohne Änderungen mit einem Terminal Programm über die serielle Schnittstelle an das scanCONTROL gesendet werden. Alternativ kann auch ImportLLTConfig verwendet werden.

Parameter

CInterfaceLLT LLT-Klasse
fileName Dateiname der Export-Datei

Rückgabewert

Standardrückgabewerte
Spezifischer Rückgabewert:

ERROR_READWRITECONFIG_CANT_CREATE_FILE	-500	Die angegebene Datei kann nicht erstellt werden

- **ExportLLTConfigString ()**

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

Auslesen aller Parameter und speichern in einen String. Dieser Konfigurations-String enthält alle relevanten Parameter und ist vor allem für Postprocessing-Anwendungen gedacht. Das Dateiformat entspricht dem Kommunikations-Protokoll für die serielle Verbindung mit dem scanCONTROL und kann daher ohne Änderungen mit einem Terminal Programm über die serielle Schnittstelle an scanCONTROL gesendet werden. Alternativ kann auch ImportLLTConfigString verwendet werden.

Parameter

CInterfaceLLT	LLT class
configData	Array für Export-String
size	Array Größe

Rückgabewert

Standardrückgabewerte
Rückgabewerte GetFeature()
Spezifischer Rückgabewert:

ERROR_READWRITECONFIG_QUEUE_TO_SMALL	-502	Datenarray zu klein

- **ImportLLTConfig ()**

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

Lesen und Setzen der von ExportLLTConfig exportierten Parameter. Kann auch .sc1 Dateien einlesen, solange diese mit einer scanCONTROL Configuration Tools Version >=5.2 gespeichert wurde. Das ignore calibration-Flag spezifiziert, ob die Einbaulagenkalibrierung von der Datei mit importiert werden soll.

Parameter

CInterfaceLLT	LLT-Klasse
fileName	Dateiname der Config-Datei
ignoreCalibration	falls wahr, wird Einbaulagenkalibrierung der Datei ignoriert

Rückgabewert

Standardrückgabewerte
Rückgabewerte SetFeature()
Spezifischer Rückgabewert:

ERROR_READWRITECONFIG_CANT_OPEN_FILE	-502	Die angegebene Datei kann nicht geöffnet werden.

ERROR_READWRITECONFIG_FILE_EMPTY	-503	Die angegebene Datei ist leer.
ERROR_READWRITE_UNKNOWN_FILE	-504	Datenformat der Datei falsch.
ERROR_READWRITECONFIG_CANT_CREATE_FILE	-500	Die angegebene Datei kann nicht erstellt werden

- **ImportLLTConfigString ()**

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

Liest Einstellungen die mittels ExportLLTConfigString exportiert wurden und setzt diese auf den Sensor. Das ignore calibration-Flag spezifiziert, ob die Einbaulagenkalibrierung von der Datei mit importiert werden soll.

Parameter

CInterfaceLLT	LLT-Klasse
configData	Array mit Config-String
size	Arraygröße
ignoreCalibration	falls wahr, wird Einbaulagenkalibrierung der Datei ignoriert

Rückgabewert

Standardrückgabewerte
Rückgabewerte SetFeature()
Spezifischer Rückgabewert:

ERROR_READWRITE_UNKNOWN_FILE	-504	Datenformat falsch.

- **SaveGlobalParameter ()**

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

Speichern der IP-Einstellungen und der Einbaulagenkalibrierung unabhängig vom User Mode.

Parameter

CInterfaceLLT	LLT-Klasse

Rückgabewert

Standardrückgabewerte

7 Anhang

7.1 Standardrückgabewerte

Alle Funktionen des Interfaces geben einen Integer-Wert als Rückgabewert zurück. Ist der Rückgabewert einer Funktion größer oder gleich GENERAL_FUNCTION_OK bzw. ’1’, so war die Funktion erfolgreich, ist der Rückgabewert GENERAL_FUNCTION_NOT_AVAILABLE bzw. ’0’ oder negativ, so ist ein Fehler aufgetreten.

Zur Unterscheidung der einzelnen Rückgabewerte stehen mehrere Konstanten zur Verfügung. In der folgenden Tabelle sind alle allgemeinen Rückgabewerte aufgeführt, die von Funktionen zurückgegeben werden können. Für die einzelnen Funktionsgruppen kann es zusätzlich noch spezielle Rückgabewerte/Fehlerwerte geben.

Konstante für den Rückgabewert	Wert	Beschreibung
GENERAL_FUNCTION_OK	1	Funktion erfolgreich ausgeführt
GENERAL_FUNCTION_NOT_AVAILABLE	0	Diese Funktion ist nicht verfügbar, evtl. neue DLL verwenden oder in den Ethernet-Mode wechseln
ERROR_GENERAL_NOT_CONNECTED	-1001	Es besteht keine Verbindung zum scanCONTROL -> *Connect()* aufrufen
ERROR_GENERAL_DEVICE_BUSY	-1002	Die Verbindung zum scanCONTROL ist gestört oder getrennt -> neu verbinden und Anschluss des scanCONTROLs überprüfen
ERROR_GENERAL_WHILE_LOAD_PROFILE_OR_GET_PROFILES	-1003	Funktion konnte nicht ausgeführt werden, da entweder das Laden von Profilen oder die Profilübertragung aktiv ist
ERROR_GENERAL_WHILE_GET_PROFILES	-1004	Funktion konnte nicht ausgeführt werden, da die Profilübertragung aktiv ist
ERROR_GENERAL_GET_SET_ADDRESS	-1005	Die Adresse konnte nicht gelesen oder geschrieben werden. Eventuell wird eine zu alte Firmware verwendet
ERROR_GENERAL_POINTER_MISSING	-1006	Ein benötigter Pointer ist NULL
ERROR_GENERAL_SECOND_CONNECTION_TO_LLT	-1008	Es ist eine zweite Instanz über Ethernet mit diesem scanCONTROL verbunden. Bitte schließen Sie die zweite Instanz

7.2 Übersicht der Beispiele im SDK

Als Leitfaden für die Integration des scanCONTROLs in eigene Projekte sind die Beispielprogramme im Projektordner gedacht. Sie stehen zur Anschauung komplett mit Quelltext zur Verfügung.

Name	Beschreibung
GetProfilesCallback	Übertragen von Profilen zur linLLT und Einlesen der Profile per Callback
GetProfilesPoll	Übertragen von Profilen zur linLLT und Einlesen der Profile via Polling Mode

VideoMode	Übertragen und Abspeichern des Matrixbildes
PartialProfile	Übertragen von partiellen Profilen
ContainerMode	Übertragen von Profil-Containern bzw. Graustufenbilder
PartialProfile_MultiLLT	Verwenden von mehreren scanCONTROLs in einer Anwendung mit partiellen Profilen
LLTPeakFilter	Setzen der Peakfilter, des frei definierbaren Messfeldes und des Encoder-nachgeführten Messfeldes
Calibration	Einbaulage kalibrieren

7.3 Einschränkungen

Scanner-Funktionalität die im Vergleich zur Win-SDK (LLT.dll) nicht in der linLLT abgebildet ist:

- SHOT-Transfer (MultiShot())
- Laden und Speichern von Profilen
- CMM-Triggerung

7.4 Unterstützende Dokumente

[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; Revision: c76c4eff608a ; Datum: 2016/06/29 09:55:05

[2] Operation Manual PartB 2700: Interface Specification for scanCONTROL 2700 Device Family; Firewire (IEEE 1394) Bus, Ethernet and Serial Port; Supplement B to the scanCONTROL 2700 Manual; MICRO-EPSILON Optronic GmbH; Revision: 1.50; Datum: 2015/10/05 10:10:20

[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; Revision: 1.98; Datum: 2013/12/09 07:21:00

[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; Revision: c76c4eff608a ; Datum: 2016/06/29 09:55:05

[5] scanCONTROL 2600 Quick Reference; Brief Introduction to scanCONTROL 2600 Device Family; MICRO-EPSILON Optronic GmbH; Revision: 9dac9498283b; Datum: 2016/06/29 09:57:00

[6] scanCONTROL 2700 Quick Reference; Brief Introduction to scanCONTROL 2700 Device Family; MICRO-EPSILON Optronic GmbH; Revision: 1.30; Datum: 2013/11/28 15:23:16

[7] scanCONTROL 2800 Quick Reference; Brief Introduction to scanCONTROL 2800 Device Family; MICRO-EPSILON Optronic GmbH; Revision: 1.33; Datum: 2011/05/24 10:12:50

[8] scanCONTROL 2900 Quick Reference; Brief Introduction to scanCONTROL 2900 Device Family; MICRO-EPSILON Optronic GmbH; Revision: 9dac9498283b; Datum: 2016/06/29 09:57:38

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