SYS TEC
ELECTRONIC

CANopen Bootloader

Software Manual

Auflage 10

Ausgabe Juni 2015

Dokument-Nr.: L-1112d_10

SYS TEC electronic GmbH   Am Windrad 2   D-08468 Heinsdorfergrund
Telefon: +49 3765 38600-0   Telefax: +49 3765 38600-4100
Web: http://www.systec-electronic.com   Mail: info@systec-electronic.com

Im Buch verwendete Bezeichnungen für Erzeugnisse, die zugleich ein eingetragenes Warenzeichen darstellen, wurden nicht besonders gekennzeichnet. Das Fehlen der © Markierung ist demzufolge nicht gleichbedeutend mit der Tatsache, dass die Bezeichnung als freier Warenname gilt. Ebenso wenig kann anhand der verwendeten Bezeichnung auf eventuell vorliegende Patente oder einen Gebrauchsmusterschutz geschlossen werden.

Die Informationen in diesem Handbuch wurden sorgfältig überprüft und können als zutreffend angenommen werden. Dennoch sei ausdrücklich darauf verwiesen, dass die Firma SYS TEC electronic GmbH weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgeschäden übernimmt, die auf den Gebrauch oder den Inhalt dieses Handbuches zurückzuführen sind. Die in diesem Handbuch enthaltenen Angaben können ohne vorherige Ankündigung geändert werden. Die Firma SYS TEC electronic GmbH geht damit keinerlei Verpflichtungen ein.

Ferner sei ausdrücklich darauf verwiesen, dass SYS TEC electronic GmbH weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgeschäden übernimmt, die auf falschen Gebrauch oder falschen Einsatz der Hard- bzw. Software zurückzuführen sind. Ebenso können ohne vorherige Ankündigung Layout oder Design der Hardware geändert werden. SYS TEC electronic GmbH geht damit keinerlei Verpflichtungen ein.

© Copyright 2015 SYS TEC electronic GmbH, D-08468 Heinsdorfergrund.

Alle Rechte vorbehalten. Kein Teil dieses Buches darf in irgendeiner Form ohne schriftliche Genehmigung der Firma SYS TEC electronic GmbH unter Einsatz entsprechender Systeme reproduziert, verarbeitet, vervielfältigt oder verbreitet werden.

Informieren Sie sich:

<table>
<tr><th>Kontakt</th><th>Direkt</th><th>Ihr lokaler Distributor</th></tr>
<tr><td>Adresse:</td><td>SYS TEC electronic GmbH
Am Windrad 2
D-08468 Heinsdorfergrund
GERMANY</td><td rowspan="5">Hier finden Sie eine Liste unserer Distributoren:

http://www.systec-electronic.com/distributors</td></tr>
<tr><td>Angebots-Hotline:</td><td>+49 3765 38600-0
info@systec-electronic.com</td></tr>
<tr><td>Technische Hotline:</td><td>+49 3765 38600-2140
support@systec-electronic.com</td></tr>
<tr><td>Fax:</td><td>+49 3765 38600-4100</td></tr>
<tr><td>Webseite:</td><td>http://www.systec-electronic.com</td></tr>
</table>

10. Auflage Juni 2015

Inhaltsverzeichnis

1 Einleitung 5

2 Referenzen 6

3 Konzept des CANopen Bootloaders 7
3.1 Prüfsumme Applikation 12
3.2 Start des Bootloaders 14
3.3 Softwarestruktur des Bootloaders 16
3.4 Vorüberlegungen für ein Bootloader Projekt 17
3.4.1 Anzahl der Applikationen 18
3.4.2 Ablage der Bootloader-spezifischen Parameter 18
3.4.3 Einsatz eines Betriebssystems 19
3.4.4 Gemeinsame Nutzung von Parametern 19
3.4.5 Gemeinsame Nutzung von Interrupts 19
3.4.6 Gemeinsame Nutzung von Funktionen 20
3.4.7 Rücksprung der Applikation in den Bootloader 21
3.4.8 Einsatz eines Watchdogs 21
3.4.9 Art der SDO-Übertragung 22
3.4.10 Verschlüsselung 23

4 Funktionsschnittstellen 24
4.1 Schnittstelle zum Bootloader 24
4.2 Schnittstelle zum Flash 27
4.3 Schnittstelle zum Timer 32
4.4 Schnittstelle für Bootloader-spezifische Parameter 33
4.5 Rücksprung in Bootloader 41
4.6 Schnittstelle zum CAN-Bus 43
4.7 Debug-Ausgaben 43
4.8 Konfiguration des Bootloaders 44
4.8.1 Konfigurationen in der Datei blcopcfg.h 44
4.8.2 Konfigurationen in der Datei tgtblcop.h 48
4.9 Hinweise zur Reduzierung des Code-Bedarfs des Bootloaders 49
4.9.1 Optimierungen des CAN-Treibers: 49
4.9.2 Optimierung des Timers 49
4.9.3 Optimierungen des OBD-Moduls 50
4.9.4 Optimierungen im SDO-Server Modul 50
4.9.5 Optimierung bei der Berechnung der Applikations-CRC 50

5 Flashtool auf dem Host 52
5.1 Das Tool BinaryBlockConv 52
5.2 Das Tool BinaryBlockDownload 57
5.2.1 Fehlercodes von BinaryBlockDownload 58
5.3 Das Tool DotNetFlashtool 60

6 Ressourcen 63
6.1 Code Daten Target 63
6.2 Interrupts Target 63

Abbildungsverzeichnis

Bild 1: CANopen Kommunikation über Objekte ... 8
Bild 2: Datenformat eines Blocks mit Programmdaten ... 10
Bild 3: Ablauf der Datenübertragung ... 10
Bild 4: Aufbau Flash ... 12
Bild 5: Programmablauf beim Start des Bootloaders ... 14
Bild 6: Softwarestruktur Bootloader ... 16
Bild 7: Beispiel für eine Flash-Aufteilung ... 18
Bild 8: Weiterleitung der Interrupt-Vektoren im Flash ... 20
Bild 9: Prinzipieller Ablauf der Bootloader main-Funktion ... 24
Bild 10: Bitpositionen für den Rückgabecode von TgtProcessAppInfoData ... 38
Bild 11: Lage von Vendor-ID und Produkt-ID in den Daten ... 39
Bild 12: Prinzipieller Ablauf des verschobenen Löschens der Applikation ... 39
Bild 13: Format des Parameters für die Version ... 53

Tabellenverzeichnis

Tabelle 1: Liste der CANopen Objekte für den Bootloader ... 7
Tabelle 2: Kommandos des Bootloaders in Index 0x1F51 ... 9
Tabelle 3: Status des Programmdownloads in Index 0x1F57 ... 9
Tabelle 4: Fehlercodes der Bootloader-Funktionen ... 26
Tabelle 5: Fehlercodes des Flash-Treibers ... 31
Tabelle 6: Defines für den Rückgabecode von TgtProcessAppInfoData ... 38
Tabelle 7: Unterstützte CPUs und CAN-Controller ... 43
Tabelle 8: Parameter des Tools BinaryBlockConv ... 53
Tabelle 9: Parameter des Tools BinaryBlockDownload ... 57
Tabelle 10: Fehlercodes von BinaryBlockDownload ... 59
Tabelle 11: Parameter des Tools DotNetFlashtool ... 61
Tabelle 12: CAN-Schnittstellen für das DotNetFlashtool ... 61
Tabelle 13: CAN-Bitraten für das DotNetFlashtool ... 62

Abkürzungen

API	Application Programming Interface
CAN	Controller Area Network
CiA	CAN in Automation
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
d.h.	das heißt
DMA	Direct Memory Access
EDS	Electronic Data Sheet
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIFO	First In First Out
ID	Identifier
IDE	Integrated Development Environment
LSB	Least Significant Bit
LSS	Layer Setting Service
MSB	Most Significant Bit
NVRAM	Non-Volatile Random-Access Memory
OD	Object Dictionary
PC	Personal Computer
PLL	phase-locked loop
RAM	Random-Access Memory
SDO	Service Data Object
usw.	und so weiter
z.B.	zum Beispiel

1 Einleitung

Der CANopen-Bootloader ist ein Softwarepaket, mit dessen Hilfe Programme im Binärformat mittels CANopen auf die Zielhardware übertragen und dort ausgeführt werden. Die Funktionalität orientiert sich an den im CANopen-Standard CiA-302 getroffenen Festlegungen.

Das Softwarepaket besteht aus zwei Teilen, den Flashtools und dem Bootloader. Die Flashtools wandeln die Anwendungsdaten (z.B. S3, INTEL-Hex) in ein binäres Format und übertragen sie an die Zielhardware. Der Bootloader empfängt die von den Flashtools gesendeten Daten, verifiziert sie und schreibt die Daten in den Flash, anschließend startet er die übertragene Anwendung. Die Kommunikation und Datenübertragung zwischen Bootloader und den Flashtools findet mittels CANopen SDO-Transfer statt.

Dieses Manual beschreibt die Funktionsweise des Bootloaders, das Format der Binärdaten sowie die Schnittstelle zur Anpassung des Bootloaders an eine Anwenderhardware:

- Schnittstelle zum Bootloader
- Schnittstelle zum Flash (Abs.)
- Schnittstelle Systemtimer (Abs.)
- Schnittstelle zu den Bootloader-spezifischen Parametern
- Schnittstelle zum Rücksprung in den Bootloader aus der Applikation (Abs.)
- Schnittstelle zum CAN-Bus, falls noch kein CAN-Treiber für den ausgewählten CAN-Controller existiert (Abs.)
- Schnittstelle für Debug-Ausgaben (Abs.)
- Schnittstelle Interrupt-Vektortabelle (Abs.)

Die Schnittstelle zur Anwenderhardware ist als Template implementiert.

2 Referenzen

/1/ *CANopen Additional application layer functions, CiA-302 Part 3, Version 4.0.2, 19: November 2008*

/2/ *CANopen Application layer and communication profile, CiA-301, Version V4.2.0.72, 05. June 2012*

/3/ *L-1020, CANopen User Manual, SYS TEC electronic GmbH, Auflage 14; Dezember 2014*

/4/ *L-1023, CAN-Treiber Software Manual, SYS TEC electronic GmbH, Auflage 3, Dezember 2004*

3 Konzept des CANopen Bootloaders

Die Realisierung basiert auf dem CANopen-Standard CiA—302 Part 3 (*siehe /1/*). Der Standard definiert Objekteinträge, mit deren Hilfe ein Programm-Download ausgeführt werden kann. Der Bootloader verwendet zur Datenübertragung Service Daten Objekte (SDO aus dem CiA-301 Standard – *siehe /2/*). Je nach Anwendung des Bootloaders können die geladenen Daten nichtflüchtig in einem Flash-Speicher programmiert werden. Die Verwendung des CANopen SDO-Transfers zur Übertragung der Daten hat den Vorteil, dass CANopen-Tools verwendet werden können, um die Applikation zu programmieren. Dazu wird ein EDS-File benötigt, das die Objekteinträge des CANopen-Knoten abbildet. Die Daten werden im Binärformat übertragen und müssen je nach Entwicklungsumgebung in dieses Format konvertiert werden (z.B. HEX → BIN – *siehe Kapitel 5.1*).

Der Bootloader stellt eine vollständige CANopen-Applikation dar, besitzt daher auch ein Objektverzeichnis. Jedoch ist dieses Objektverzeichnis nur für die Dauer der Ausführung des Bootloaders sichtbar. Der Bootloader besteht aus einem Target-spezifischen Teil und einem Teil, der die Kommunikationsdienste kapselt. Der Target-spezifische Teil enthält die Schnittstellen zum Flash, CAN-Controller, Timer und nichtflüchtigen Speicher. Hier sind durch den Anwender Anpassungen vorzunehmen.

Zur Übertragung der Daten wird der SDO-Transfer nach CiA-301 verwendet. Dieser Übertragungsdienst unterstützt Protokolle wie Segmented Transfer und Block Transfer. Der Segmented Transfer ist voreingestellt, da dieser Dienst weniger Programm-Code für die Ausführung benötigt. Jedoch ist die Übertragungsdauer auf Grund der segmentweisen Quittierung der Daten etwas größer als im Vergleich zum Block Transfer.

Objekt (Index/ Subidx)	**Datentyp**	**Attribut**	**Bedeutung**	**CiA Std.**
0x1000	Unsigned32	ro	Kennzeichnet den Gerätetyp	301
0x1001	Unsigned8	ro	CANopen Fehlerregister	301
0x1018/0	Unsigned8	ro	Anzahl der Subindizes in 0x1018	301
0x1018/1	Unsigned32	ro	CANopen Vendor-ID	301
0x1018/2	Unsigned32	ro	CANopen Produkt-ID	301
0x1018/3	Unsigned32	ro	Revisionsnummer	301
0x1018/4	Unsigned32	ro	Seriennummer	301
0x1F50/0	Unsigned8	ro	Anzahl der Subindizes in 0x1F50	302
0x1F50/1	Domain	wo	Programmdaten (Container bis zu 16384 Bytes	302
0x1F51/0	Unsigned8	ro	Anzahl der Subindizes in 0x1F51	302
0x1F51/1	Unsigned8	rw	Kommandokanal	302
0x1F56/0	Unsigned8	ro	Anzahl der Subindizes in 0x1F56	302
0x1F56/1	Unsigned32	ro	Identifiziert die Applikation (CRC der Applikation)	302
0x1F57/0	Unsigned8	ro	Anzahl der Subindizes in 0x1F57	302
0x1F57/1	Unsigned32	ro	Flash Fehlerstatus	302

Tabelle 1: Liste der CANopen Objekte für den Bootloader

(ro = read-only, rw = read-write, wo = write-only)

Das Objekt 0x1F50 dient zum Laden der Programmdaten. Das Objekt ist vom Typ DOMAIN und kann Daten bis zu einer vom Anwender definierten Blockgröße empfangen. Das Objekt 0x1F51/1 dient zum Ausführen von bestimmten Kommandos (Löschen des Flashs, Starten der Applikation, usw.). Das Objekt 0x1F56 enthält eine Identifikation der Applikation. Hier wurde die CRC der Applikation hinterlegt. Mit dem Objekt 0x1F57 kann der aktuelle Fehlerstatus ausgelesen werden. Die *Tabelle 1* listet alle CANopen Objekte und deren Bedeutung auf.

Bedeutung des Sub-Index
Der CANopen-Standard definiert bis zu 255 Sub-Indizes pro Index. Subindex 0 enthält hierbei die Anzahl der folgenden Subindizes. Für die Bootloader-Funktionalität wird jedem Subindex größer als 0 der Objekte 0x1F50 bis 0x1F57 ein Programm zugeordnet, d.h. es wäre möglich, in einem Gerät bis zu 255 verschiedene Programme zu übertragen. Jedem Programm ist daher ein Bereich innerhalb des Flash für die Ablage zuzuordnen. Sämtliche Anweisungen, wie Löschen, Schreiben, Programmieren für ein ausgewähltes Programm müssen mit der gleichen Referenz, d.h. gleichen Subindex auf diesen Bereich übertragen werden.

Bild 1 zeigt eine schematische Darstellung der Kommunikation über diese CANopen Objekte, wobei der SDO Dienst verwendet wird.

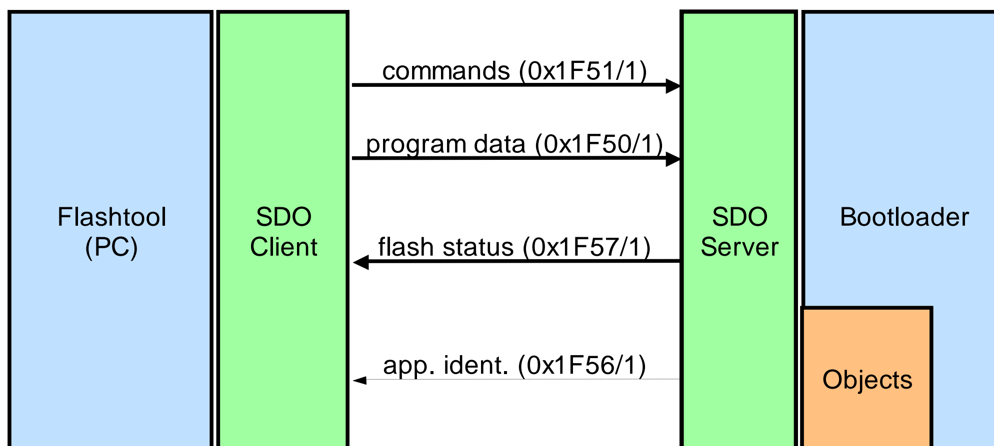


Bild 1: CANopen Kommunikation über Objekte

Bedeutung der Kommandos
Der Bootloader auf dem Target besitzt ein Kommando-Interface. Er führt generell nur Anweisungen auf Anforderung durch den Host aus. Der Host überträgt Kommandos (Schreiben eines Kommandos auf Objekt 0x1F51 oder Übertragen eines Datenblocks auf Objekt 0x1F50) und kontrolliert den Ausführungsstatus mit Hilfe des Objektes 0x1F57.

Die folgenden Kommandos sind über Index 0x1F51 ausführbar:

Wert	Kommando	Bedeutung
0x00	STOP	Das Target wird angewiesen, das laufende Programm zu stoppen. Dieses Kommando ist derzeit nicht implementiert.
0x01	START	Das Target wird angewiesen, das ausgewählte Programm zu starten.
0x02	RESET_STAT	Das Target wird angewiesen, den Status (Index 0x1F57) zurückzusetzen
0x03	CLEAR	Das Target wird angewiesen, den Bereich des Flashs zu löschen, der mit dem entsprechendem Subindex ausgewählt wurde.
0x80	START_BOOTLO ADER	Mit Hilfe dieses Kommandos ist es möglich, aus der Applikation in den Bootloader zurückzuspringen. Dieser Eintrag muss also von der Applikation unterstützt werden, um den Bootloader starten zu können (*siehe Kapitel 3.2*).
0x83	SET_SIGNATURE	Das Target wird angewiesen, das ausgewählte und programmierte Programm als „gültig“ zu kennzeichnen. Das ist neben einer gültigen CRC und Knotennummer die Voraussetzung, um das Programm selbständig nach einem Power-on-Reset zu starten.
0x84	CLR_SIGNATURE	Das Target wird angewiesen, das ausgewählte und programmierte Programm als „ungültig“ zu kennzeichnen. Nach einem Power-on-Reset würde daraufhin die Applikation nicht gestartet werden, der Bootloader bleibt aktiv.

Tabelle 2: Kommandos des Bootloaders in Index 0x1F51

Der in Index 0x1F57 rücklesbare Status kann die folgenden Zustände annehmen:

Statuswert	Bezeichnung	Bedeutung
0x00000000	OK	Das letzte übertragene Kommando wurde fehlerfrei ausgeführt.
0x00000001	BUSY	Die Ausführung eines Kommandos dauert an.
0x00000002	NOVALPROG	Es wurde versucht, ein ungültiges Applikationsprogramm zu starten.
0x00000004	FORMAT	Das Format der Binärdaten, welche auf Index 0x1F51 übertragen wurden, ist fehlerhaft.
0x00000006	CRC	Die CRC der Binärdaten ist fehlerhaft.
0x00000008	NOTCLEARED	Es wurde versucht, zu programmieren, obwohl ein gültiges Applikationsprogramm vorhanden ist.
0x0000000A	WRITE	Es trat ein Fehler beim Schreiben des Flash auf.
0x0000000C	ADDRESS	Es wurde versucht, eine ungültige Adresse im Flash zu beschreiben.
0x0000000E	SECURED	Es wurde versucht, einen geschützten Flashbereich zu beschreiben.
0x00000010	NVDATA	Beim Zugriff auf den nichtflüchtigen Speicher (z.B. Programmieren der Signatur) ist ein Fehler aufgetreten.
0x0000007E	UNSPECIFIED	Ein unspezifizierter Fehler ist aufgetreten
0x00000080	WRONG VID	Dem Bootloader wurde eine Firmware mit ungültiger Vendor-ID übergeben.
0x00000082	WRONG PID	Dem Bootloader wurde eine Firmware mit ungültiger Produkt-ID übergeben.
0x00000084	NO ACCESS	Das Programmieren der Firmware über den Bootloader ist (derzeit) nicht erlaubt.
0x00000086	MAN BUSY	Es wurde versucht, die Programmdaten auf das Objekt 0x1F50 zu schreiben, obwohl der Status auf Busy steht.
0x00000088	SEQUENCE	Der Bootloader hat eine falsche Abfolge der Blöcke in den Binärdaten erkannt.

Tabelle 3: Status des Programmdownloads in Index 0x1F57

Download der Daten
Die Übertragung der Binärdaten eines Programms erfolgt Blockweise. Nach jeder erfolgreichen Übertragung eines Blocks wird dieser in den Flash geschrieben. *Bild 2* zeigt den Aufbau eines Blocks.

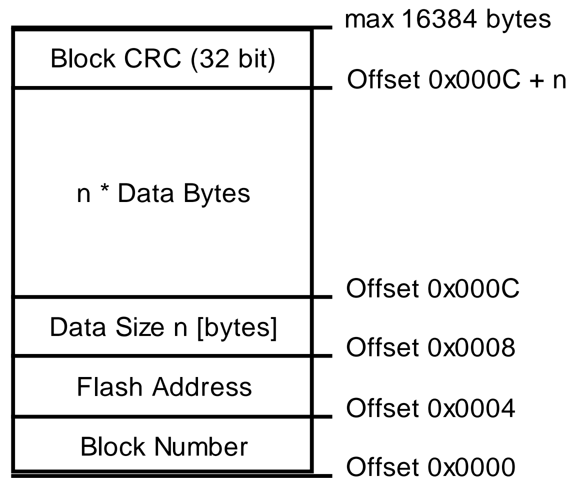


Bild 2: Datenformat eines Blocks mit Programmdaten

Eine CAN-Nachricht wird bereits durch den CAN-Controller mit einer CRC gesichert. Zusätzlich ist jedem Block eine CRC anhängig. Sie wird über Block-Nummer, Flash-Adresse, Anzahl der Datenbytes und den Datenbytes selbst berechnet. Ein fehlerhaft übertragener Block wird N-mal wiederholt (Parameter PC-Tool).

Wichtig:
Daten werden stets im Intel-Format (Little-Endian / LSB first) übertragen.

Die Blocknummer ist eine laufende Nummer beginnend mit 0, die nach jeder erfolgreichen Übertragung um 1 erhöht wird. Dabei kennzeichnet die Blocknummer 0 immer den Beginn der Übertragung. Hierdurch werden die Host-Seite und der Bootloader auf dem Target synchronisiert. Block 0 enthält Steuerinformationen für den Bootloader auf dem Target (falls erforderlich Blockgröße, Flash-Informationen, usw.), aber noch keine Programmdaten. Ab Blocknummer 1 werden dann Programmdaten übertragen. Der letzte Block erhält immer die Blocknummer „-1“ (0xFFFFFFFF). In diesem Block sind die Applikationsgröße und die Applikations-CRC eingetragen. Nach der Übertragung dieses Blockes beginnt der Bootloader die CRC über die Applikation zu bilden um diese zu vergleichen. Das Ergebnis wird dann auch im Objekt 0x1F56/1 eingetragen. Erst danach startet das Flashtool die Applikation, indem das Kommando „Start“ auf das Objekt 0x1F51/1 eingetragen wird.

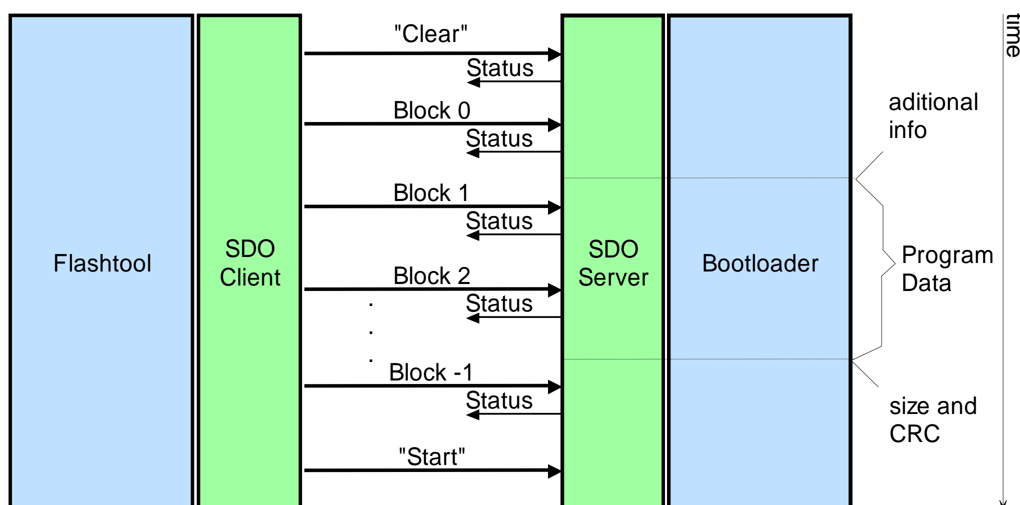


Bild 3: Ablauf der Datenübertragung

© SYS TEC electronic GmbH 2015

Prinzipieller Ablauf
Der Host (Flashtools) realisiert folgenden Ablauf, um eine neue Applikation auf das Target zu übertragen. Bei Fehlern wird der gesamte Vorgang abgebrochen.

1. Wandeln der Applikation in Binärformat (z.B. HEX → BIN).
2. Lesen von Objekt 0x1000 (Device Type), um festzustellen, ob der Bootloader aktiv ist. Entspricht der gelesene Wert dem des Bootloaders (0x10000000), wird mit Schritt 4 fortgefahren. Falls der SDO-Transfer mit Timeout fehlschlug, diesen Schritt wiederholen.
3. Ausführen des Kommandos für Rücksprung in den Bootloader (Schreiben des Wertes 0x80 auf Objekt 0x1F51/1). Danach Schritt 2 wiederholen.
4. Ausführen des Kommandos für Löschen des Flashs (Schreiben des Wertes 0x03 auf Objekt 0x1F51/1).
5. Lesen von Objekt 0x1F57, um festzustellen, ob das Löschen durchgeführt wurde. Falls der SDO-Transfer mit Timeout fehlschlug oder der Wert BUSY zurückgeliefert wurde, diesen Schritt wiederholen.
6. Ersten Datenblock aus Binärdatei laden.
7. Download des Datenblocks auf Objekt 0x1F50/1.
8. Lesen von Objekt 0x1F57, um festzustellen, ob das Schreiben des Datenblocks durchgeführt wurde. Falls der SDO-Transfer mit Timeout fehlschlug oder der Wert BUSY zurückgeliefert wurde, diesen Schritt wiederholen.
9. Wenn ein weiterer Datenblock in Binärdatei vorhanden ist, diesen Lesen und mit Schritt 7 fortfahren.
10. Ausführen des Kommandos Starten der Applikation (Schreiben des Wertes 0x01 auf Objekt 0x1F51/1).
11. Lesen von Objekt 0x1000 (Devicetype), um festzustellen, ob der Bootloader noch aktiv ist. Entspricht der gelesene Wert dem des Bootloaders (0x10000000), den gesamten Vorgang mit Fehler abbrechen. Falls der SDO-Transfer mit Timeout fehlschlug, diesen Schritt wiederholen.
12. Ausführen des Kommandos für Rücksprung in den Bootloader (Schreiben des Wertes 0x80 auf Objekt 0x1F51/1).
13. Lesen von Objekt 0x1000 (Devicetype), um festzustellen, ob der Bootloader aktiv ist. Entspricht der gelesene Wert nicht dem des Bootloaders (0x10000000), den gesamten Vorgang mit Fehler abbrechen. Falls der SDO-Transfer mit Timeout fehlschlug, diesen Schritt wiederholen.
14. Ausführen des Kommandos zum Setzen der Signatur (Schreiben des Wertes 0x83 auf Objekt 0x1F51/1).
15. Ausführen des Kommandos Starten der Applikation (Schreiben des Wertes 0x01 auf Objekt 0x1F51/1).

Hinweis:

Mit manchen Applikationen wird es nicht immer möglich sein, diese kurzzeitig zu starten, um zu prüfen, ob die Applikation tatsächlich startet. Deshalb wurden die Schritte 10 bis 13 aus dem aktuellen Flashtool (DotNetFlashtool – *siehe Kapitel 5.3*) entfernt. Der Anwender ist dafür verantwortlich, dass die Applikation lauffähig ist. Letztendlich ist die Übertragung mehrfach mit einer CRC abgesichert (über das CAN-Protokoll selbst, in den einzelnen Blöcken, und die CRC über die gesamte Applikation im Flash).

Das Tool BinaryBlockDownload unterstützt die Schritte 10 bis 13 weiterhin. Änderungen Im Bootloader sind dafür nicht vorzunehmen, macht jedoch unter Umständen Einschränkungen bei der Applikation:

- Die Applikation muss unbedingt eine CANopen-Applikation sein und den Rücksprung zurück in den Bootloader unterstützen (*siehe Kapitel 4.5*).

-	Die Applikation darf keine wichtigen Steuerungen sofort nach dem Start ausführen, da diese durch den Rücksprung in den Bootloader abgebrochen werden können.

Timeouts
Für die Übertragung der Daten vom Host (Flashtools) zum Target (Bootloader) sind zwei Timeouts relevant. Zum einen das SDO-Timeout für die Bestätigung des übertragenen SDO-Dienstes (Ein SDO-Transfer wird stets bestätigt). Das ist relativ klein gewählt, da hier nur die Verzögerung durch die Übertragungsstrecke (Bitrate, Busbelastung) berücksichtigt werden muss.

Das andere ist das Timeout für die Ausführung des angeforderten Kommandos. Generell gilt, dass ein Kommando vollständig übertragen und bestätigt sein muss, bevor die Ausführung startet. Die Ausführungsdauer auf dem Target und damit das zu wählende Timeout auf der Host-Seite richtet sich nach dem Kommando. Für das Löschen des Flash oder das Schreiben der Daten müssen gegebenenfalls andere Zeiten berücksichtigt werden, als für das Schreiben einer Signatur oder dem Starten der Applikation. Nach diesem Timeout muss das Target wieder reagieren (z.B. auf SDO-Anfragen) bzw. in Objekt 0x1F57 nicht mehr BUSY zurückliefern. Es obliegt dem Anwender, die korrekten Timeouts zu konfigurieren.

3.1 Prüfsumme Applikation

Für jeden Datenblock als auch für den gesamten Applikationsbereich wird eine Prüfsumme (CRC) gebildet. Es wird davon ausgegangen, dass sich der Bereich für die Applikation linear in den Adressraum der CPU abbilden lässt (*siehe Bild 4*).

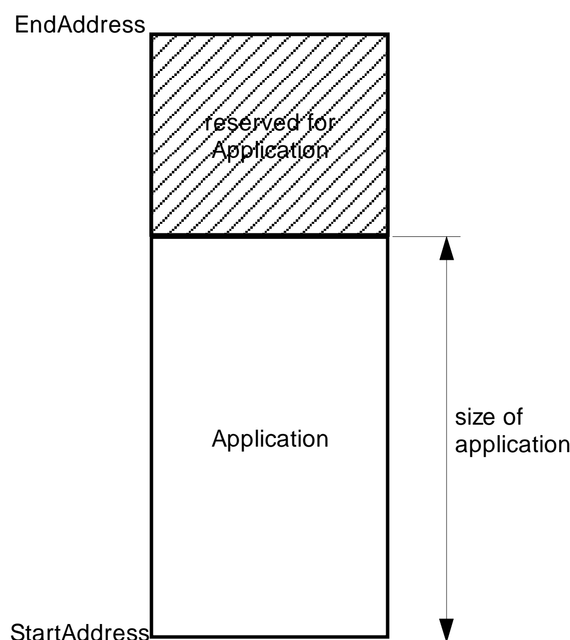


Bild 4: Aufbau Flash

Die Startadresse wird als Konstante im Source definiert. Die CRC als auch die Größe der Applikation werden vom Host an das Target im letzten Block -1 übertragen. Das Target hinterlegt diese beiden Werte in einem nichtflüchtigen Speicher. Dazu existieren Target-spezifische Funktionstemplates, die vom Anwender anzupassen sind. Als nichtflüchtiger Speicher kann hier der Flash selbst oder ein EEPROM verwendet werden.

Die CRC wird über die Applikation (Länge AppSize) des Applikationsbereichs berechnet. Die Funktion für die Berechnung der CRC ist in der Datei Crc32.c als CRC32 mit dem Polynom 0xEDB88320 implementiert. Der Startwert ist 0. StartAddress und EndAddress werden als Konstanten hinterlegt und müssen mit den Parametern für die Erzeugung der Binärdaten identisch sein.

Sollte eine CPU eine Peripherie für die Berechnung einer CRC verfügen, dann kann diese Peripherie genutzt werden. Für eine 16-Bitige CRC ist dies bereits erfolgreich umgesetzt worden. Für dessen Nutzung muss in der Datei blcopcfg.h das Define BLCOP_USE_CRC16 auf TRUE gesetzt werden. Der Bootloader ruft dann statt der Funktion CalcCrc32() aus Crc32.c die Funktion TgtCalcCrc16() auf. Diese Funktion muss der Anwender selbst implementieren (z.B. in der Datei Target.c).

Bei der Verwendung eines anderen CRC Polynoms muss beachtet werden, dass das Tool für die Konvertierung der Applikation als HEX-File in das binäre Format dieses CRC Polynom auch verwenden muss. Das Tool BinaryBlockConv.exe unterstützt dies mit dem Parameter --crc16 bereits mit dem 16 Bitigen Polynom 0x1021 (ab Version 2.2). Soll ein anderes CRC Polynom verwendet werden, dann muss dies erst in dieses Tool implementiert werden.

3.2 Start des Bootloaders

Bild 5 zeigt den realisierten Ablauf beim Start des Bootloaders.

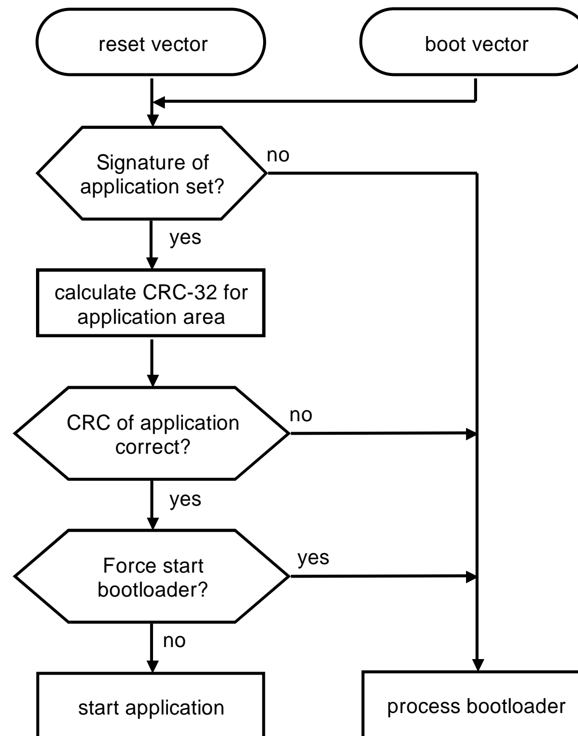


Bild 5: Programmablauf beim Start des Bootloaders

Es gibt zwei Einsprungspunkte für den Bootloader. Nach dem Hardware-Reset springt die CPU den Reset-Vektor an. Der zweite Einsprungspunkt ist ein Sprungvektor, den der Bootloader zur Verfügung stellt, damit die Applikation den Bootloader aktivieren kann. Dieser zweite Einsprungspunkt muss jedoch nicht in jedem Bootloader implementiert sein. Lesen Sie dazu das Kapitel 3.4.7, um nähere Informationen zu erhalten.

Das Starten der Applikation setzt voraus, dass die CRC über den Applikationsbereich identisch mit der auf dem Host berechneten und auf dem Target hinterlegten CRC ist, dass eine gültige Knotennummer existiert (die Knotennummer muss innerhalb eines für diese Applikation definierten Wertebereiches liegen) und dass eine gültige Signatur hinterlegt wurde. Ist eine Bedingung nicht erfüllt, so verweilt das Target im Bootloader.

Für die Signatur gibt es verschiedenen Realisierungsvarianten. Die Signatur kann im Flash oder einem anderen nichtflüchtigen Speicher (z.B. EEPROM, NVRAM) hinterlegt werden.

Denkbar wäre jedoch auch die Signatur durch ein Port Pin zu realisieren. In jedem Fall gilt: Ist die Signatur nicht gesetzt, dann wird der Bootloader gestartet. Dieser verweilt bis das Kommando zum Starten eine Applikation vom Flashtool empfangen wird.

Hinweis:
Für den Rücksprung aus der Applikation in den Bootloader gibt es unterschiedliche Realisierungsmöglichkeiten. Das hängt zum einen davon ab, wie ein Rücksprung durch den Mikrocontroller selbst als auch durch den Compiler und Linker unterstützt werden. Hierbei ist zu beachten, dass beim Verlassen der Applikation sämtliche Ressourcen des Mikrocontrollers, wie Interrupts, DMA-Channel, on-chip Peripherie freigegeben werden müssen (Interrupts sperren, DMA-Transfer beenden, Peripherie sperren) und der System-Stack zurückgesetzt wird. Eine einfache, hier dargestellte Methode ist das Auslösen eines RESET-Signals per Software. Dazu muss jedoch unmittelbar davor eine Warmstart-Signatur im RAM gesetzt werden. Beim Starten des Targets nach einem RESET wird der Bootloader gezwungen, zu starten, somit verweilt das Target im Bootloader und wartet auf weitere Kommandos vom Host. Um ein Starten des Bootloaders über das OD realisieren zu können, ist in der Applikation der Index 0x1F51 – „Program Control“ und das Kommando START_BOOTLOADER (*siehe Tabelle 2*) zu implementieren.

3.3 Softwarestruktur des Bootloaders

Bild 6 zeigt die Softwarestruktur des Bootloaders.

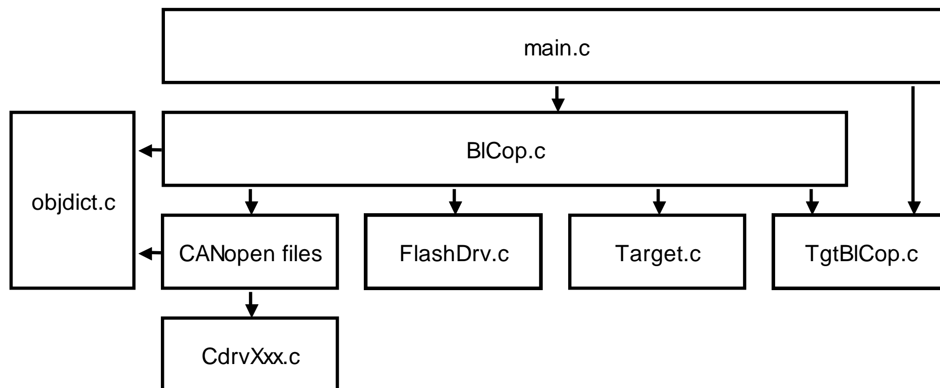


Bild 6: Softwarestruktur Bootloader

Der Bootloader besteht zum Teil aus Dateien aus dem CANopen Source Code SO-877. Das spiegelt sich in der Verzeichnisstruktur (*siehe Abschnitt weiter unten*) wider. Für nähere Informationen über die Funktionen aus dem CANopen Stack wird auf das Manual des CANopen Source Codes L-1020 verwiesen. Die Funktionen des CAN-Treibers sind im Manual L-1023 beschrieben.

blcop\	blcop.c blcop.h	API Schicht für den CANopen Bootloader
ccm\	...	Dateien für die CCM-Schicht der Flashtools auf dem Host
cdrv\	...	CAN-Treiber Dateien für unterschiedliche Targets
copstack\	amixxx.c	Enthält spezielle Funktionen für den Speicherzugriff
	cob.c	Funktionen für das Erzeugen von Kommunikationsobjekten.
	lsss.c	Unterstützung des LSS Service
	obd.c	Funktionen für den Zugriff auf das Objektverzeichnis
	sdoscomm.c	Funktionen für den SDO Server
	...	Weitere Dateien
include\		Include Dateien des CANopen Stacks
objdicts\	blcop_1app\... blcop_2app\...	Vordefinierte Objektverzeichnisse für den Bootloader
target\	pmc14\...	Projektdateien für die Hardware COMBImodul C14 (Infineon XC161)
	explorer16\...	Projektdateien für die Hardware Explorer16 (Microchip dsPIC33F)
	...	Weiter Targets
Tools\	binaryblockconv.exe dotnetflashtool.exe ...	PC Tools für die Konvertierung der Applikation in das binäre Blockformat und für den Download zum Target.

3.4 Vorüberlegungen für ein Bootloader Projekt

Vor dem Beginn der Implementierung eines CANopen Bootloaders muss der Anwender einige Fragen klären:

- Wie viele Applikationen soll der Bootloader verwalten können?
- Wie wird der Flash für die Ablage der Applikationen aufgeteilt?
- In welchem nichtflüchtigen Speicher werden die Bootloader-spezifischen Parameter (Größe der Applikation, Signatur und CRC) abgelegt (z.B. Flash, EEPROM, …)?
- Sind Knotennummer, Seriennummer, CAN Bitrate usw. konstante Werte, oder müssen diese anderweitig bestimmt werden (z.B. DIP-Schalter für Knotennummer, Einsatz des LSS Service, Ablage im nichtflüchtigen Speicher)?
- Soll der CANopen Bootloader im Zusammenhang mit einem Betriebssystem verwendet werden (z.B. Linux, eCos/Redboot, …)?
- Gibt es Parameter, Funktionen oder andere Ressourcen, die der CANopen Bootloader und die Applikation(en) gemeinsam nutzen müssen (z.B. gespeicherte Knotenadresse und CAN-Bitrate im nichtflüchtigen Speicher über den LSS-Service; Einsatz eines Callgates, um gemeinsame Funktionen aufrufen zu können; Interrupts; …)?
- Welcher Mechanismus wird verwendet, um den automatischen Start der Applikation durch den Bootloader zu verhindern? Diese Frage bezieht sich auch auf den Rücksprung der Applikation in den Bootloader.
- Wird ein Watchdog verwendet?
- Soll die Datenübertragung per SDO-Blocktransfer per SDO-Segmented Transfer durchgeführt werden?
- Müssen die Firmware-Daten der Applikation in der Binärdatei verschlüsselt vorliegen?

3.4.1 Anzahl der Applikationen

Der CANopen Bootloader ist in der Lage, mehrere Applikationen zu verwalten. Die maximale Anzahl an Applikationen wird zum einen durch die maximale Anzahl an möglichen Subindizes der Objekte 0x1FXX im Standard CiA-302 Part 3 mit 254 begrenzt. Zum anderen begrenzt die verwendete Hardware mit dem zur Verfügung stehenden Flash-Speicher die maximale Anzahl der zu verwaltenden Applikationen. Werden mehrere Applikationen verwaltet, dann muss der Anwender einen Mechanismus implementieren, der festlegt, welche der programmierten und gültigen Applikationen nach dem Start automatisch ausgeführt werden soll. Bild 7 zeigt ein typisches Beispiel für die Aufteilung des Flashs mit mehreren Applikationen.

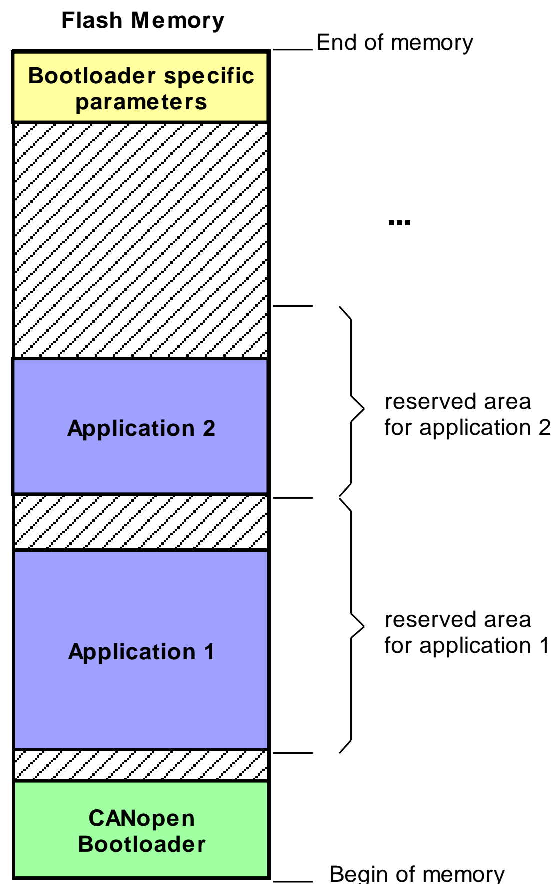


Bild 7: Beispiel für eine Flash-Aufteilung

Der CANopen Bootloader benötigt Flash-Treiber für die Programmierung einer Applikation in den Flash. Für diesen Flash-Treiber definiert der CANopen Bootloader eine feste Funktions-API (*siehe Kapitel 4.1*). Besteht bereits ein Flash-Treiber für die entsprechende CPU (bzw. für einen externen Flash) mit einer anderen Funktions-API, dann können die in Kapitel 4.1 beschriebenen Funktionen als Wrapper verwendet werden.
Weiterhin benötigt der CANopen Bootloader eine Zeitbasis, um zum Beispiel Timeouts für den SDO-Transfer prüfen zu können. Die dafür benötigten Funktionen muss der Anwender für die entsprechende CPU ausfüllen (*siehe Kapitel 4.3*).

3.4.2 Ablage der Bootloader-spezifischen Parameter

Es gibt Bootloader-spezifische Parameter, die der Bootloader für die korrekte Funktionalität in den nichtflüchtigen Speicher ablegen muss (z.B. Größe der Applikation, Signatur und CRC). Nach einem Neustart des Bootloaders werden diese Parameter wieder gelesen und überprüft. Abhängig von der Hardware des Gerätes kann der nichtflüchtige Speicher auf verschiedene Weise realisiert sein (z.B. ein reservierter Flash-Sektor, ein angeschlossener EEPROM usw.). Der Anwender muss den Zugriff auf diesen nichtflüchtigen Speicher über die in Kapitel 4.4 beschriebenen Funktionen implementieren.

3.4.3 Einsatz eines Betriebssystems

Soll der CANopen Bootloader mit einem Betriebssystem eingesetzt werden, dann ist darauf zu achten, dass die Bootloader-Funktionen nicht von mehreren Threads/Tasks aus gerufen werden. Diese Funktionen sind diesbezüglich nicht geschützt.

Auch bei Verwendung eines Betriebssystems muss die Prozessfunktion BlCopProcess() zyklisch gerufen werden. Sollten keine Interrupts verwendet werden, dann gilt dies auch für die Funktion TgtProcessEvents().

Bei Verwendung eines File-Systems für die Ablage der Applikationen (statt im Flash) müssen die Funktionen für die Flash-Treiber als Wrapper für die File-Funktionen dienen. Die Funktion FlsDrvWriteInitialize() kann dabei für das Erstellen einer neuen Datei dienen und die Funktion FlsDrvWriteData() schreibt Schritt für Schritt die Programmdaten in die Datei. Wenn der Bootloader die CRC über die Applikation berechnet, ruft er die Funktion FlsDrvReadData(). Diese Funktion muss dann aus der Datei Schritt für Schritt lesen. Das Starten der Applikation erfolgt in der main-Funktion, so dass auch hier die Möglichkeit besteht, die Datei zu laden und auszuführen.

3.4.4 Gemeinsame Nutzung von Parametern

Bei Verwendung des CANopen Bootloaders besteht die Software für das Gerät aus zwei oder mehreren Teilen (Bootloader und Applikation(en)). Da beide Teile auf den CAN-Bus zugreifen, sollte die CAN-Bitrate und die Knotenadresse in beiden Teilen gleich sein. Wird nun aber die Knotenadresse bzw. CAN-Bitrate per LSS bestimmt, dann muss sich der Anwender in der Konzeptphase überlegen, in welchem Teil der LSS Slave Service implementiert ist, und wie die über LSS eingestellten Parameter dem/den anderen Software-Teil(en) zur Verfügung gestellt werden sollen. Es gibt verschiedene Herangehensweisen:

- <u>Der Bootloader und die Applikation implementieren den LSS Slave Service</u>:
  Bei der Erstprogrammierung muss dem Bootloader eine gültige Knotenadresse vergeben werden. Diese Knotenadresse kann durch den Bootloader im nichtflüchtigen Speicher abgelegt werden, so dass die Applikation diese als Start-Wert für die Knotenadresse verwenden kann. Der LSS Master kann jederzeit die Knotenadresse der Applikation ändern. Die Applikation hat dann auch die Möglichkeit den Wert im nichtflüchtigen Speicher zu überschreiben.
- <u>Nur der Bootloader implementiert den LSS Slave Service</u>:
  Der Bootloader muss die Knotenadresse bzw. CAN-Bitrate im nichtflüchtigen Speicher für die Applikation hinterlegen. Eine Änderung dieser Parameter kann nur über den Bootloader erfolgen.
- <u>Nur die Applikation implementiert den LSS Slave Service:</u>
  Der Bootloader muss bei der Erstprogrammierung mit einer Default-Knotenadresse und –Bitrate starten. Wenn die Applikation zum ersten Mal gestartet wird, können Knotenadresse bzw. CAN-Bitrate per LSS eingestellt werden. Diese Parameter muss die Applikation im nichtflüchtigen Speicher ablegen. Beim nächsten Start des Bootloaders muss der Bootloader diese Parameter aus dem nichtflüchtigen Speicher lesen.

3.4.5 Gemeinsame Nutzung von Interrupts

Verwendet der Bootloader auch Interrupts (z.B. für den Timer und die CAN-Schnittstelle), dann bedarf es einen Mechanismus, wie beide Software-Teile diese Interrupts gemeinsam nutzen können. Oftmals existiert in der CPU nur eine Interrupt-Vector-Tabelle im Flash. In diesem Flash-Bereich ist aber üblicherweise der Bootloader programmiert. Um dieses Problem zu lösen, muss der Bootloader eine Interrupt-Vektor-Weiterleitung durchführen. Das kann man zum Beispiel über eine Spiegelung der Vektor-Tabelle in einem anderen Flash-Bereich (Bereich der Applikation – *siehe Bild 8*) realisieren. Oder man legt eine Vektor-Tabelle im RAM an, die die Applikation verändern kann.

Es gibt jedoch auch CPUs, deren Startadresse der Interrupt-Vektor-Tabelle über ein Register eingestellt werden kann. In diesem Fall muss der Bootloader vor dem Start der Applikation dieses Register verändern, so dass die Interrupts der Applikation angesprungen werden statt die des Bootloaders. Auf diese Weise ist eine zusätzliche Weiterleitung durch den Bootloader nicht notwendig.

Im Idealfall verwendet der CANopen Bootloader keine Interrupts. Die CAN-Nachrichten werden dann im Polling-Betrieb verarbeitet. Voraussetzung dafür ist, dass die Funktion TgtProcessEvents() zyklisch gerufen wird. Die CPU benötigt für diesen Fall auch einen Timer, dessen Counter-Register direkt gelesen und für die Zeitbasis umgerechnet werden kann (Zeitbasis in 100µs Auflösung).

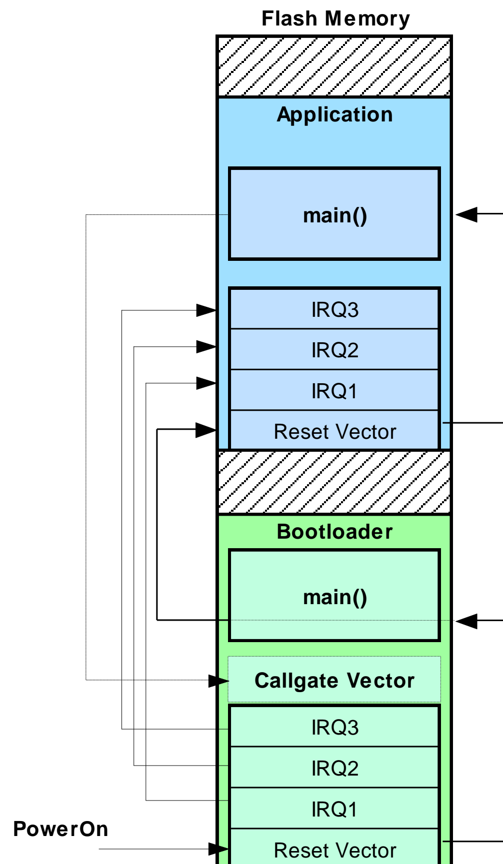


Bild 8: Weiterleitung der Interrupt-Vektoren im Flash

Wichtig:
Soll der CANopen Bootloader keinen Interrupt verwenden, dann muss das Define CDRV_USE_NO_ISR in der Datei copcfg.h mit TRUE definiert werden. Das führt dazu, dass in der Datei target.c keine Interrupt-Service-Routinen für CAN, Timer und UART angelegt werden.

3.4.6 Gemeinsame Nutzung von Funktionen

Oftmals ist es so, dass der verfügbare Flash-Speicher stark begrenzt ist. Damit besteht die Bedingung, dass Bootloader und Applikation vom Code-Bedarf möglichst klein gehalten werden müssen. Verwendet die Applikation Funktionen, die auch der Bootloader bereits implementiert, dann können diese Funktionen über ein sogenanntes „Callgate" der Applikation zur Verfügung gestellt werden (Datei blcop\blcopcg.c). Hier ist jedoch zu beachten, dass diese Funktionen möglichst nur Funktions-lokale Variablen verwenden, die der Compiler auf den Stack legt. Ist dies nicht immer möglich, dann müssen die globalen Variablen, die von diesen Funktionen verwendet werden, in einem für den Bootloader reservierten RAM-Bereich liegen. Die Applikation darf diesen RAM-Bereich nicht überschreiben. Die Realisierung eines solchen Callgates erfolgt durch eine feste Sprungadresse im Bootloader (*siehe Bild 8*). Über Funktionsparameter kann der Callgate-Sprungfunktion übermittelt werden, welche Aufgabe mit welchen zusätzlichen Parametern ausgeführt werden soll.

Als Beispiel wurde die Funktion zur Berechnung der CRC über den Applikationsbereich bereits erfolgreich über ein Callgate der Applikation zur Verfügung gestellt. Dieses Callgate kann auf Anfrage angefordert werden.

3.4.7 Rücksprung der Applikation in den Bootloader

Viele CPUs besitzen die Möglichkeit, einen Reset der CPU per Software auszulösen. Dies ist die optimale Variante, in den Bootloader zu wechseln. Doch ohne weitere Implementierungen würde der Bootloader die Applikation erneut automatisch starten, wenn die Konstante BLCOP_USE_AUTOSTART_APP auf TRUE gesetzt ist. Um dies zu verhindern, sollte für den Bootloader eine RAM-Variable reserviert werden, die die Applikation nie anderweitig verändern darf (üblicherweise die letzte RAM-Zelle des Systems). Vor dem Rücksprung der Applikation über die Funktion TgtStartBootloader() muss auf diese RAM-Zelle eine bestimmte Signatur schreiben. In der main-Funktion des Bootloaders muss nun diese RAM-Zelle gelesen und mit der Signatur verglichen werden. Stimmt diese Signatur überein, dann ruft die main-Funktion des Bootloaders die Funktion BlCopInitialize() mit dem Parameter bForceBootStart_p = TRUE. Damit startet der Bootloader die Applikation nicht neu.

Der Parameter bForceBootStart_p kann auch auf TRUE gesetzt werden, wenn eine spezielle Einstellung von DIP- oder HEX-Kodierschaltern erkannt wird. Dies kann hilfreich sein, wenn eine fehlerbehaftete Applikation gar nicht bis zum Rücksprung zum Bootloader kommt.

Besitzt die CPU nicht die Möglichkeit, einen Reset per Software auszulösen, dann muss die Applikation vor dem Rücksprung alle verwendeten Ressourcen (Timer, CAN-Controller, Interrupts usw.) wieder freigeben und den Bootloader über den Reset-Vektor direkt anspringen. Auch hier muss vor dem Rücksprung eine Variable im RAM für den Bootloader reserviert werden, um eine entsprechende Signatur hinterlegen zu können.

Alternativ kann auch ein Watchdog dazu verwendet werden, um einen Reset auszulösen. Der Watchdog ist dabei auf eine kleine Timeout-Zeit einzustellen. Ohne den Watchdog zu triggern wartet dann die Applikation darauf (z.B. über eine endlose while-Schleife), dass der Watchdog einen Reset auslöst.

Bei CPUs, die über einen Cache für Programmcode bzw. für die Daten im RAM besitzen, ist darauf zu achten, dass das Schreiben der Signatur in die reservierte RAM-Zelle abgeschlossen ist, bevor der Reset auslöst. Nutzen Sie bei Problemen eine Flush-Funktion des Caches oder schalten Sie den Cache vor dem Rücksprung einfach aus.

3.4.8 Einsatz eines Watchdogs

Der CANopen Bootloader ruft in regelmäßigen Abständen die Funktion TgtWatchdogProcess(), wenn das Define BLCOP_USE_WATCHDOG auf TRUE steht. Diese Funktion muss der Anwender in der Datei tgtblcop.c implementieren. Die Initialisierung des Watchdogs muss der Anwender in der Funktion TgtBlCopInitialize() vornehmen.

In welchen Abständen der Bootloader die Funktion TgtWatchdogProcess() aufruft, ist stark von der verwendeten CPU, deren Taktfrequenz und der Konfiguration des Bootloaders abhängig. Alle im Bootloader ablaufenden Prozesse sind so implementiert, dass sie in mehrere Teilschritte unterteilt sind. Zum Beispiel bei der Berechnung der CRC der Applikation wird die Routine zur Berechnung der CRC mehrfach mit kleineren Bereichen der Applikation aufgerufen. Die Größe dieser Bereiche muss der Anwender in der Datei blcopcfg.h mit dem Define BLCOP_MAX_CRC_STEP_SIZE einstellen. Das gleiche gilt auch für das Schreiben der Applikation in den Flash. Die Anzahl der Bytes, die mit einem Teilschritt in den Flash programmiert werden soll, muss der Anwender mit dem Define BLCOP_MAX_FLSWRITE_STEP_SIZE einstellen. Diese Größe muss aber mindestens einer schreibbaren Flash-Page entsprechen.

Der Anwender hat die Möglichkeit, die maximale Aufrufzeit experimentell zu bestimmen. Dafür sollte der Watchdog vorerst nicht scharf geschaltet werden. In der Funktion TgtWatchdogProcess() kann der Anwender die Zeitdifferenz zum vorigen Aufruf dieser Funktion berechnen. Übersteigt diese Zeitdifferenz der vorigen, dann wird dieser Wert global abgespeichert. Nun sollte der CANopen Bootloader als solcher mit seiner Funktionalität komplett durchlaufen werden. Das heißt Download einer Applikation über den CAN-Bus bis kurz vor den Start der Applikation. Nun kann die maximal bestimmte Zykluszeit für den Aufruf der Funktion aus dem globalen Speicher gelesen werden. Ein

eventuelles Fein-Tuning kann der Anwender über die Defines BLCOP_MAX_CRC_STEP_SIZE und BLCOP_MAX_FLSWRITE_STEP_SIZE vornehmen. Aber es muss darauf geachtet werden, dass, wenn diese Werte zu klein eingestellt werden, kann sich dies negativ auf die absolute Download-Zeit und auf die Startzeit der Applikation nach Power-On bzw. einem Reset auswirken.

Zu beachten ist auch, wie sich der Watchdog beim Übergang in die Applikation verhalten soll. Triggert die Applikation den Watchdog auch, muss der Watchdog vor dem Start der Applikation nicht deaktiviert werden (oftmals ist die Deaktivierung auch gar nicht möglich). Andernfalls muss der Bootloader den Watchdog deaktivieren. Das kann in der Funktion TgtJumpApplication() erfolgen, die für das Starten der Applikation zuständig ist.

3.4.9 Art der SDO-Übertragung

Für die Übertragung der Programmdaten wird der SDO-Service nach /2/ verwendet. Für diesen Service gibt es zwei unterschiedliche Arten, um größere Datenpakete zu übertragen.
Die Code-sparenste Variante ist der SDO-Segmented Transfer. Bei dieser Übertragungsart, wird jede CAN-Nachricht, die der Client (d.h. die Flashtools auf dem Host) an den Server (d.h. der Bootloader) sendet, über eine weitere CAN-Nachricht quittiert. Das kann dazu führen, dass die absolute Übertragungszeit einer Applikation über den CAN-Bus stark ansteigt.

Besteht die Forderung, dass die absolute Übertragungszeit möglichst klein gehalten werden soll, dann sollte der SDO-Blocktransfer verwendet werden. Dabei sendet der Client die Programmdaten in größeren Blöcken. Ein Block besteht aus mehreren CAN-Nachrichten. Erst nach dem Empfang eines kompletten Blockes sendet der Server eine Quittierung an den Client zurück. Die Aktivierung der SDO-Blocktransfers erfolgt in der Datei copcfg.h, indem das Define SDO_BLOCKTRANSFER auf TRUE gesetzt wird. Die Defines SDO_BLOCKSIZE_DOWNLOAD und SDO_BLOCKSIZE_UPLOAD bestimmen, aus wie vielen CAN-Nachrichten ein Block maximal besteht. Nähere Informationen zum SDO-Blocktransfer finden Sie im CANopen Manual L-1020.

Es ist zu beachten, dass das Senden eines Blockes (d.h. mehrere CAN-Nachrichten) auf dem CAN-Bus sehr schnell erfolgt. Der CANopen Bootloader muss in der Lage sein, diesen Burst an CAN-Nachrichten zu verarbeiten. Unter Umständen kann ein CAN-Controller mit einem internen Empfangs-FIFO abfangen. Besitzt der CAN-Controller keinen solchen FIFO, dann sollte die Verwendung des CAN-Interrupts statt des Pollings vorgezogen werden. Der Empfangspuffer im CAN-Treiber muss in jedem Fall groß genug sein, um den Burst an CAN-Nachrichten aufnehmen zu können.

Kann der CANopen Bootloader den Burst an CAN-Nachrichten nicht verarbeiten, dann erkennt man das am geänderten CAN-Treiber-Status. Der Bootloader ruft dabei intern die Funktion BlCopCbError() und sendet (falls aktiviert) eine Emergency Nachricht auf den CAN-Bus. Die Bytes 5 und 6 (die Zählung beginnt mit 0) der Emergency Nachricht beinhalten dann den eigentlichen CAN-Treiber-Status. Dieser Status ist Bit-orientiert, wobei mehrere Fehler gleichzeitig gesetzt sein können. Die möglichen Fehler im CAN-Treiber-Status können aus dem CAN-Treiber Manual L-1023 entnommen werden. Die für diesen Fall wichtigen Bits sind unten aufgeführt:

```
#define kCdrvOverrun            0x0020  // Overrun im CAN-Controller
#define kCdrvRxBuffLowOverrun   0x1000  // Overrun im Empfangspuffer
                                        // des CAN-Treibers
```

Läuft der Empfangspuffer des CAN-Treibers über, dann kann dies u.U. über die Vergrößerung des Empfangspuffers gelöst werden. Diese Einstellung erfolgt über die Datei obdcfg.h im Define CDRV_MAX_RX_BUFF_ENTRIES_LOW.

Tritt ein Overrun im CAN-Controller trotz Verwendung des CAN-Interrupts auf, dann ist evtl. der FIFO des CAN-Controllers zu klein, die Interrupt-Latenzzeit oder die Interrupt-Ausführungszeit zu groß. Letzteres kann unter Umständen über eine hohe Compiler-Optimierungsstufe behoben werden. Bei manchen CAN-Controllern/Treibern kann man die Größe des Empfangs-FIFOs einstellen. Da diese Einstellung sehr spezifisch für den verwendeten CAN-Controller ist, sollten Sie in diesem Fall Kontakt mit unserer Support-Abteilung aufnehmen.

3.4.10 Verschlüsselung

Müssen die Firmware-Daten verschlüsselt an den CANopen Bootloader übertragen werden, dann muss zunächst die Binärdatei durch BinaryBlockConv.exe verschlüsselt werden. Dies wird im Kapitel 5.1 beschrieben.

Der CANopen Bootloader im Target muss die verschlüsselten Daten mit dem gleichen Schlüssel wieder entschlüsseln, bevor sie in den Flash programmiert werden. Dazu ist zunächst die Datei .\target\source\decrypt.c mit in das Projekt des CANopen Bootloader aufzunehmen. In dieser Datei ist die Variable abDecryptKey_l mit einem neuen Schlüssel zu versehen, der letztendlich auch für die BinaryBlockConv.exe verwendet wird.

```
static CONST BYTE ROM   abDecryptKey_l[] =
{
    0xcf, 0x7a, 0x7f, 0xe1, 0x96, 0xf6, 0x4c, 0x6b,
    0x8f, 0x19, 0xe5, 0x83, 0x61, 0xd2, 0x39, 0x7e
};
```

Zusätzlich müssen zwei zusätzliche Konstanten in die Datei blcopcfg.h aufgenommen werden:

```
#define BLCOP_USE_DECRYPT                TRUE
#define BLCOP_DECRYPT_SUBTRAHEND         0x00
```

Der Wert, der für den Startwert des Verschlüsselungsalgorithmus mit der Konstante BLCOP_DECRYPT_SUBTRAHEND angegeben wird, muss dem Tool BinaryBlockConv.exe mit dem Parameter --add übergeben werden.

Hinweis:

Dieser Startwert, der hinter der Konstanten BLCOP_DECRYPT_SUBTRAHEND angegeben wird, sollte nicht dazu verwendet werden, mehrere Targets unterschiedlich zu verschlüsseln. Das sollte Aufgabe des 128-Bit Schlüssels sein (Variable abDecryptKey_l bzw. Aufrufparameter --key).

4 Funktionsschnittstellen

4.1 Schnittstelle zum Bootloader

Die Bootloader-API selbst besteht aus 3 Funktionen, die von der main-Funktion ausgeführt werden müssen. Diese Funktionen werden in diesem Kapitel beschrieben. Bild 9 zeigt den prinzipiellen Ablauf der main-Funktion des Bootloaders. Das Starten der Applikation obliegt der main-Funktion, wobei die Funktion BlCopProcess() zurückgibt, welche Applikation gestartet werden soll. Der Anwender kann hier noch zusätzlichen Code implementieren, um eine Priorisierung vorzunehmen, falls mehrere Applikationen verwaltet werden.

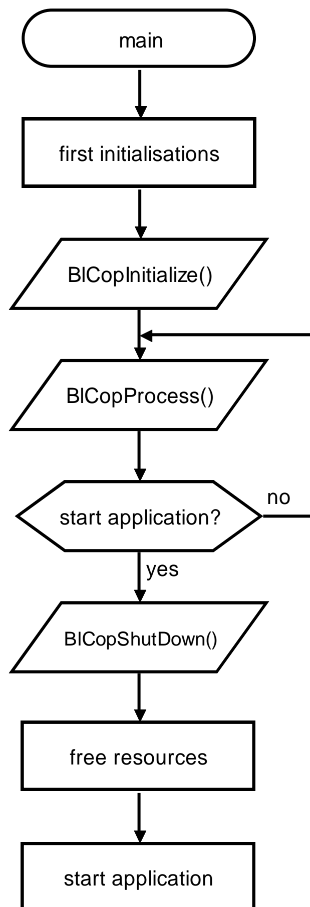


Bild 9: Prinzipieller Ablauf der Bootloader main-Funktion

© SYS TEC electronic GmbH 2015

Funktion BlCopInitialize

```
tBlCopState PUBLIC BlCopInitialize (
    BYTE  bNodeId_p,
    BYTE  bBaudIdx_p,
    BYTE  bForceBootStart_p,
    DWORD dwSerialNr_p);
```

Bedeutung:
Diese Funktion initialisiert den Bootloader. Sie muss vor dem Aufruf von BlCopProcess() einmal aufgerufen werden.

Parameter:

bNodeId_p: Knotenadresse, die der Bootloader verwenden soll. Der Bereich kann zwischen 1 und 127 liegen, kann zusätzlich über die Defines BLCOP_MIN_NODEID und BLCOP_MAX_NODEID eingeschränkt werden. Wird der LSS Service verwendet, dann ist auch der Wert 0xFF erlaubt. In diesem Fall wartet der Bootloader auf die Vergabe der Knotenadresse durch einen LSS Master, bevor ein Download einer Applikation möglich ist.

bBaudIdx_p: CAN-Bitrate, die der Bootloader verwenden soll. Der Wert liegt zwischen kBdi10kBaud und kBdi1MBaud (siehe CAN-Treiber Manual L-1023). Der Bereich kann zusätzlich über die Defines BLCOP_MIN_BAUIDX und BLCOP_MAX_BAUIDX eingeschränkt werden.

bForceBootStart_p: Steht dieser Paramater auf TRUE, dann bleibt der Bootloader aktiv, auch wenn eine gültige Applikation programmiert ist. Wenn FALSE, dann startet der Bootloader eine gültige Applikation. Dieser Parameter kann z.B. dazu verwendet werden, um nach dem Rücksprung aus der Applikation im Bootloader zu verweilen.

dwSerialNr_p: Die Seriennummer des Gerätes, die im Identity Object 0x1018 des Objektverzeichnisses eingetragen wird, muss mit diesem Parameter übergeben werden. Mit einer eindeutigen Seriennummer (bzw. LSS-Adresse) kann der Bootloader über den LSS Scann gefunden werden.

Rückgabe:
Fehlercode nach *Tabelle 4*

Sollten die Parameter Knotenadresse, CAN-Bitrate usw. für den Bootloader nicht fest definiert sein, dann muss das Lesen dieser Parameter aus dem nicht-flüchtigen Speicher vor dem Aufruf der Funktion BlCopInitialize() implementiert werden. Nutzen Sie gegebenenfalls dafür die Target-spezifischen Funktionen TgtGetNodeId(), TgtGetSerialNr() usw. für die Bestimmung dieser Paramenter.

Funktion BlCopProcess

```
tBlCopState PUBLIC BlCopProcess (void);
```

Bedeutung:
Nach der Initialisierung des Bootloaders über die Funktion BlCopInitialize() muss die Funktion BlCopProcess() zyklisch in einer Schleife gerufen werden. Sie verwaltet die State Machine des CANopen Bootloaders und ruft alle weiteren notwendigen Prozessfunktionen des CANopen Stacks. In diesem Kontext werden alle Timeouts für die Übertragung der Applikation(en) per SDO überwacht.

Parameter:
keine

Rückgabe:
Fehlercode nach *Tabelle 4*
Gibt die Funktion einen Wert kleiner als kBlCopForceStartApp zurück, dann muss diese Funktion weiter zyklisch in einer Schleife aufgerufen werden. Jeder Wert ab kBlCopForceStartApp zeigt der main-Funktion an, dass eine Applikation gestartet werden soll. Der Rückgabewert minus kBlCopForceStartApp kennzeichnet den Index der Applikation, die gestartet werden soll (0 für die erste Applikation, 1 für die zweite usw.).

Funktion BlCopShutDown

```
void PUBLIC BlCopShutDown (void);
```

Bedeutung:
Diese Funktion fährt den CANopen Bootloader herunter, indem es die Ressourcen für den verwendeten Speicher, Timer und für die CAN-Schnittstelle freigibt. Alle weiteren Ressourcen, die von der main-Funktion des Bootloaders zusätzlich reserviert worden sind, muss die main-Funktion selbst wieder freigeben.

Parameter:
keine

Rückgabe:
keine

Konstante	Wert	Bedeutung
kBlCopOk	0	Kein Fehler
kBlCopError	1	Ein allgemeiner Fehler ist aufgetreten.
kBlCopInitError	2	Ein Fehler bei der Initialisierung über die Funktion BlCopInitialize() ist aufgetreten.
kBlCopForceStartApp	128	Alle Werte ab 128 zeigen nach Aufruf der Funktion BlCopProcess() an, dass eine Applikation gestartet werden soll. Der Rückgabewert subtrahiert mit 128 ergibt den Index der Applikation, die gestartet werden soll.

Tabelle 4: Fehlercodes der Bootloader-Funktionen

4.2 Schnittstelle zum Flash

Die nachstehenden Funktionen sind in der Datei flashdrv.c zu implementieren. Bei einer Integration im Hause der SYS TEC electronic GmbH ist die einwandfreie Funktion mit einem Demoprogramm nachzuweisen.

Die Funktions-API des Flash-Treibers ist so angelegt, dass je nach verwendeter CPU die Parameter für Flash-Adresse und Größe angepasst werden können. Dafür definiert die Datei flashdrv.h zwei Typen FLSDRV_ADDR und FLSDRV_SIZE, die der Anwender entsprechend für die CPU anpassen muss. Handelt es sich um eine CPU, deren adressierbarer Bereich für den Flash 32 Bit umfasst, dann müssen beide Typen auf DWORD (bzw. „unsigned long int“) gesetzt sein. Um den Code-Bedarf für kleinere CPUs zu reduzieren, deren adressierbarer Flash-Bereich nur 16 Bit umfasst, können beide Defines auf WORD (bzw. „unsigned short int“) gesetzt werden.

Funktion FlsDrvInitialize

```
tFlsDrvResult PUBLIC FlsDrvInitialize (void);
```

Bedeutung:
Diese Funktion initialisiert den Flash-Treiber (Register und Modul-eigene Variablen).

Parameter:
keine

Rückgabe:
Fehlercode nach *Tabelle 5*.

Falls die Programmierung des Flashs nur aus dem RAM möglich ist, müssen die benötigten Funktionen in das RAM kopiert werden. Der Zugriff auf diese Funktionen ist dann mit Hilfe von Funktions-Pointern zu realisieren. Der Funktionsaufruf über die Pointer muss innerhalb des Flash-Treibers realisiert sein. Das Initialisieren der Funktions-Pointer obliegt dem Anwender in der Funktion FlsDrvInitialize().

Für die erleichterte Ablage der Funktionen im RAM, wurde für die Funktionen, die in den RAM kopiert werden müssen, der Präfix PLACE_IN_RAM angelegt. Dieses Define ist entsprechend für die CPU zu definieren.

Zum Beispiel für die Renesas RX Familie mit der IAR Entwicklungsumgebung:

```
#define PLACE_IN_RAM _Pragma("location=\"RAM_AREA\"")
```

Auf diese Weise kann das Kopieren der Flash-Funktionen automatisch über den Startup erfolgen, wenn dies im Linker-File entsprechend angegeben wird. Der Aufruf über Funktions-Pointer ist damit nicht nötig.

Beispiel für Renesas RX:

```
initialize by copy { rw, ro section D, ro section D_1, ro section D_2,
           section RAM_AREA };
```

Funktion FlsDrvEraseInitialize

```
PLACE_IN_RAM tFlsDrvResult PUBLIC FlsDrvEraseInitialize (
    tFastByte bAppIndex_p,
    FLSDRV_ADDR StartAddr_p,
    FLSDRV_ADDR EndAddr_p);
```

Bedeutung:
Diese Funktion initialisiert das Löschen eines Applikationsbereiches. Ein Applikationsbereich wird durch seine Startadresse und seine Endadresse gekennzeichnet und kann es einem bis mehreren Flash-Sektoren/Pages bestehen. Die Startadresse muss kleiner als die Endadresse sein.

Parameter:

bAppIndex_p:	Index der zu löschenden Applikation.
StartAddr_p:	Startadresse des Applikationsbereiches
EndAddr_p:	Endadresse des Applikationsbereiches

Rückgabe:
Fehlercode nach *Tabelle 5*.

Diese Funktion führt das Löschen des Flashs nicht aus, sondern zeigt den Beginn des Löschvorgangs an. Sie kann z.B. dazu verwendet werden, um den Adressbereich in eine Modul-spezifische Variable abzulegen und damit den zu löschenden Flash-Sektor in der Funktion FlsDrvEraseSector() zu prüfen.

Funktion FlsDrvEraseSector

```
PLACE_IN_RAM tFlsDrvResult PUBLIC FlsDrvEraseSector (
    FLSDRV_ADDR SecAddress_p,
    FLSDRV_SIZE* pSize_p);
```

Bedeutung:
Diese Funktion löscht einen Teil des Flashs innerhalb des Applikationsbereiches (Der komplette Applikationsbereich wurde durch die Funktion FlsDrvEraseInitialize() mit Hilfe der Parameter StartAddr_p, dwEndAddr_p spezifiziert.). Ein Applikationsbereich kann mehreren Flash-Sektoren/Pages umfassen. Der Wert SecAddr_p zeigt auf die erste zu löschende Speicherzelle innerhalb des Bereichs. Der Pointer pSize_p zeigt auf die Größe des zu löschenden Bereichs. Die Anzahl der tatsächlich gelöschten Bytes muss von dieser Funktion in *pSize_p hinterlegt werden.
Gegebenenfalls ist zusätzlich die Funktion TgtWatchdogProcess() zu rufen, falls der Löschvorgang zu lange dauert.

Parameter:

SecAddr_p:	Start-Adresse des zu löschenden Bereichs
pSize_p:	Zeiger auf die Größe des zu löschenden Bereichs

Rückgabe:
Fehlercode nach *Tabelle 5*.
Der Zeiger pSize_p zeigt nach dem Aufruf auf die Anzahl der tatsächlich gelöschten Bytes.

Die Funktion FlsDrvEraseSector() sollte immer ganze Vielfache von Flash-Sektoren löschen, aber nie mehr als beim Aufruf mit *pSize_p angegeben (weniger ist erlaubt). Nach jedem Aufruf dieser Funktion berechnet der Bootloader den als nächstes zu löschenden Adressbereich, indem er die Adresse mit der in pSize_p zurückgegebenen gelöschten Größe addiert. Auf diese Weise muss der Bootloader, die Größe eines Flash-Sektors nicht kennen.

Funktion FlsDrvWriteInitialize

```
PLACE_IN_RAM tFlsDrvResult PUBLIC FlsDrvWriteInitialize (
    tFastByte bAppIndex_p,
    FLSDRV_ADDR StartAddr_p,
    FLSDRV_ADDR EndAddr_p);
```

Bedeutung:
Äquivalent zur Funktion FlsDrvEraseInitialize() zeigt diese Funktion den Schreibvorgang eines Applikationsbereichs an. Sie kann dazu verwendet werden, den Adressbereich zu speichern, so dass die Teilbereiche mit den folgenden Aufrufen von FlsDrvWriteData() überprüft werden können.

Parameter:

bAppIndex_p:	Index der zu schreibenden Applikation.
StartAddr_p:	Startadresse des Applikationsbereiches
EndAddr_p:	Endadresse des Applikationsbereiches

Rückgabe:
Fehlercode nach *Tabelle 5*.

Funktion FlsDrvWriteData

```
PLACE_IN_RAM tFlsDrvResult PUBLIC FlsDrvWriteData (
    BYTE* pData_p,
    FLSDRV_ADDR Start_p,
    FLSDRV_SIZE* pSize_p);
```

Bedeutung:
Diese Funktion schreibt Daten in den Flash-Bereich der Applikation. Das Schreiben erfolgt in kleineren Teilbereichen, als mit FlsDrvWriteInitialize() angegeben. Die tatsächliche Anzahl an geschriebenen Bytes muss von dieser Funktion über den Pointer pSize_p an den Bootloader zurückgegeben werden.

Parameter:

pData_p:	Dieser Pointer gibt die Startadresse der Quelldaten an, die in den Flash geschrieben werden sollen.
Start_p:	Mit diesem Parameter wird die Startadresse des Flashs angegeben, an die die Quelldaten geschrieben werden sollen.
pSize_p:	Dieser Pointer zeigt auf die Anzahl der Bytes, die in den Flash geschrieben werden sollen.

Rückgabe:
Fehlercode nach *Tabelle 5*.
Der Zeiger pSize_p zeigt nach dem Aufruf auf die Anzahl der tatsächlich geschriebenen Bytes.

FlsDrvWriteData() sollte immer ein ganzzahliges Vielfaches einer schreibbaren Flash-Page zurückgeben. Auf diese Weise ruft der Bootloader diese Funktion auch immer Page-aligned auf, womit sich die Implementierung vereinfacht.

Funktion FlsDrvReadData

```
PLACE_IN_RAM tFlsDrvResult PUBLIC FlsDrvReadData (
    BYTE** ppbData_p,
    FLSDRV_ADDR* pStart_p,
    FLSDRV_SIZE* pSize_p);
```

Bedeutung:
Diese Funktion liest einen Datenblock aus. Die Startadresse pStart_p wird dabei um die gelesene Anzahl von Bytes inkrementiert. Der Zeiger *pSize_p zeigt beim Verlassen der Funktion auf die Anzahl der gelesenen Bytes. Der Puffer auf die gelesenen Bytes muss gegebenenfalls der Flash-Treiber zur Verfügung stellen und dessen Adresse auf den Pointer ppbData_p schreiben.

Parameter:

ppbData_p:	Zeiger auf Adresse mit den gelesenen Daten
pStart_p:	Zeiger auf Startadresse des zu lesenden Blocks im Code-Speicher
pSize_p:	Zeiger auf die Anzahl der zu lesenden Bytes

Rückgabe:
Fehlercode nach *Tabelle 5*.
Die Funktion muss über den Pointer ppbData_p den Puffer auf die gelesenen Daten zurückgeben. Weiterhin muss sie die Startadresse, der über pStart_p adressiert wird, entsprechend der zu lesenden Anzahl an Bytes weiterstellen. Unterscheidet sich die Anzahl der gelesenen Bytes von den zu lesenden Bytes, dann muss die Anzahl über den Pointer pSize_p entsprechend angepasst werden.

Die Funktion kapselt Target-spezifische Besonderheiten beim Lesen des Code-Speichers. Für die meisten Systeme enthält diese Funktion keinen Code, außer das Weiterreichen und das Aufaddieren der Startadresse (siehe unteres Beispiel für eine ARM Cortex-M3 CPU).

```
PLACE_IN_RAM tFlsDrvResult PUBLIC FlsDrvReadData (
    BYTE** ppbData_p, FLSDRV_ADDR* pStart_p, FLSDRV_SIZE* pSize_p)
{
    // Because of architecture of the Cortex we only need to overhand
    // the start address.
    // Some other CPU architectures needs to copy data from flash into
    // RAM and overhand the RAM address here.
    *ppbData_p = (BYTE*) *pStart_p;

    // update start pointer for next call
    *pStart_p += *pSize_p;

    return kFlsDrv_OK;
}
```

Fehlercode kFlsDrvErr_...	**Wert**	**Bedeutung**
kFlsDrv_OK	0x00	Kein Fehler.
…BUSY	0x01	Der Flash-Treiber ist momentan beschäftigt.
…FAILED	0x02	Allgemeiner unspezifierter Fehler
…TIMEOUT	0x03	Ein Timeout ist aufgetreten
…INVALID_ENABLED_BLOCK	0x0F	
…PROTECTED	0x10	Der Adressbereich ist schreibgeschützt.
…ADDRESS	0x11	Ein ungültiger Adressbereich wurde angegeben.
…PARTID	0x12	
…ALIGNMENT	0x13	Der Adressbereich wurde nicht der CPU entsprechend aligned angegeben.
…SHADOW_RANGE	0x14	
…ARRAY_RANGE	0x15	
…ENABLED_SMALL_BLOCK	0x16	
…ERASE_PEGOOD	0x17	
…PROGRAM_VERIFICATION	0x18	
…PROGRAM_PEGOOD	0x19	
…NOT_BLANK	0x1A	
…SRC_DEST_VERIFY	0x1B	
…SIZE	0x1C	
…CONNECT	0x1D	Es konnte keine Verbindung zu einem entfernten Flash-Treiber (z.B. über eine UART-Schnittstelle) aufgenommen werden.
…COMMUNICATION	0x1E	Es ist ein Kommunikationsfehler mit einem entfernten Flash-Treiber (z.B. über eine UART-Schnittstelle) aufgetreten.

Tabelle 5: Fehlercodes des Flash-Treibers

4.3 Schnittstelle zum Timer

Die nachstehenden Funktionen sind in der Datei Target.c zu implementieren. Wenn möglich, so sollte auf die Verwendung von Interrupts verzichtet werden. Die Systemzeit wird zur Überwachung von Timeouts während einer aktiven SDO-Verbindung benötigt. Da im CANopen diese Zeiten als Vielfache von 100µs verarbeitet werden, ist der Wert des Timers entsprechend zu skalieren, auch wenn die Genauigkeit nur einem Wert von 1ms entspricht.

Funktion TgtInitTimer

```
void PUBLIC TgtInitTimer (void);
```

Bedeutung:
Diese Funktion initialisiert einen System-Timer für das Bereitstellen einer Zeitbasis mit einer Auflösung von 100µs.

Parameter:
keine

Rückgabe:
keine

Funktion TgtStopTimer

```
void TgtStopTimer (void);
```

Bedeutung:
Diese Funktion beendet den System-Timer und gibt die benutzten Ressourcen (Timer, Interrupt) wieder frei, so dass diese von der Applikation wieder verwendet werden können.

Parameter:
keine

Rückgabe:
keine

Funktion TgtGetTickCount

```
tTime TgtGetTickCount (void);
```

Bedeutung:
Diese Funktion liefert den aktuellen Wert des System-Timers zurück, wobei der Rückgabewert auf Vielfaches von 100µs skaliert wird.

Parameter:
keine

Rückgabe:
Aktuelle Zeitbasis in 100µs Auflösung.

Sollte die Implementierung des System-Timers ohne Interrupt Service Routine nicht möglich sein, dann kann man den System-Timer so implementieren, dass der Timer-Interrupt alle 1ms gerufen wird, die Zeitbasis aber um den Wert 10 inkrementiert.

4.4 Schnittstelle für Bootloader-spezifische Parameter

Diese Schnittstelle ist in der Datei tgtblcop.c implementiert und wird benötigt, um z.B. die Knotennummer für das CANopen-Gerät bereitzustellen und gegebenenfalls nichtflüchtig zu speichern (bei Verwendung de LSS Service). Darüber hinaus stellt diese Schnittstelle Funktionen zur Verfügung, die der Bootloader vor dem automatischen Start der Applikation benötigt (tatsächliche Größe der Applikation, CRC der Applikation, Applikations-Signatur).

Funktion TgtBlCopInitialize

```
void PUBLIC TgtBlCopInitialize (void);
```

Bedeutung:
Die Funktion TgtBlCopInitialize() wird von der main-Funktion noch vor dem Aufruf der Funktion BlCopInitialize() gerufen. Hier muss der Anwender spezielle Initialisierungen der Hardware vornehmen, die für das Bootloader-Projekt wichtig sind (z.B. Watchdog, Treiber für den nichtflüchtigen Speicher, usw.).

Parameter:
keine

Rückgabe:
keine

Funktion TgtBlCopInitVars

```
BYTE PUBLIC TgtBlCopInitVars (void);
```

Bedeutung:
Nachdem der CANopen Bootloader die Initialisierung bis zum OBD Modul vorgenommen hat, ruft er diese Funktion auf. Ab hier ist der Zugriff auf das Objektverzeichnis des Bootloaders möglich. In dieser Funktion kann der Anwender spezielle Initialisierungen vornehmen, die Beziehungen mit dem Objektverzeichnis haben.

Parameter:
keine

Rückgabe:
Der Rückgabewert diese Funktion hat momentan keine Bedeutung. Daher kann jeder beliebige Wert zurückgegeben werden.

Funktion TgtProcessEvents

```
void PUBLIC TgtProcessEvents (void);
```

Bedeutung:
Der CANopen Bootloader ruft mit der Funktion BlCopProcess() ab einem bestimmten internen Status zyklisch die Funktion TgtProcessEvents() auf. Wenn kein CAN-Interrupt verwendet werden soll, dann muss diese Funktion die Funktion CdrvInterruptHandler() des CAN-Treibers aufrufen, um das Polling der zu empfangenden CAN-Nachrichten zu realisieren.

Parameter:
keine

Rückgabe:
keine

Funktion TgtGetNodeId

```
BYTE PUBLIC TgtGetNodeId (BYTE MEM* pbNodeId_p);
```

Bedeutung:
Diese Funktion liefert die Knotenadresse (NodeID) zurück.

Parameter:

pbNodeId_p: Zeiger auf eine Variable für die Rückgabe der NodeID

Rückgabe:
Diese Funktion kann die NodeID wahlweise über den Pointer pbNodeId_p oder über den Rückgabewert der Funktion zurückgegeben werden. Der gültige Bereich der NodeID liegt zwischen 1 und 127. Ein Rückgabewert im ungültigen Bereich (z.B. eine 0), kann dazu verwendet werden, um einen Fehler bei der Bestimmung der NodeID anzuzeigen.

Diese kann z.B dazu verwendet werden, um die NodeID mit Hilfe von DIP- oder HEX-Codierschaltern zu bestimmen. Soll der LSS Service zur Bestimmung der NodeID vom Bootloader verwendet werden, dann gibt diese Funktion den im zuvor nichtflüchtigen Speicher abgelegten Speicher zurück. Wurde im nichtflüchtigen Speicher noch keine gültige NodeID hinterlegt, dann muss diese Funktion 0xFF als Node-ID zurückgeben.

Funktion TgtSetNodeId

```
BYTE PUBLIC TgtSetNodeId (BYTE bNodeId_p);
```

Bedeutung:
Diese Funktion hinterlegt eine mit LSS konfigurierte Knotennummer in einem nichtflüchtigen Speicher.

Parameter:

bNodeId_p : zu speichernde NodeID im Bereich 1 bis 127

Rückgabe:

0:	kein Fehler, Ablage erfolgreich
>0:	anwenderspezifischer Fehlercode

Funktion TgtGetFlashInfo

```
tBlCopFlashInfo GENERIC* PUBLIC TgtGetFlashInfo (void);
```

Bedeutung:
Die Funktion muss eine Tabelle zurückgeben, die Informationen über die zu programmierenden Applikationen beinhaltet.

Parameter:
keine

Rückgabe:
Die Funktion liefert die Adresse auf eine Tabelle mit Informationen über die zu programmierenden Applikationen zurück. Die Struktur tBlCopFlashInfo hat folgenden Aufbau:

```
typedef struct
{
  DWORD m_dwStartAddressOfArea; // Startadresse der Applikation im
                                // Flash
  DWORD m_dwSizeOfArea;         // maximal reservierter Bereich der
                                // Applikation im Flash
}
tBlCopFlashInfo;
```

Ein CANopen Bootloader kann mehrere Applikationen verwalten. Für den Fall muss ein Array des Typs tBlCopFlashInfo angelegt und mit den Informationen der Flash-Bereiche für jede Applikation

gefüllt werden. Gibt es nur eine Applikation, dann besteht dieses Array nur aus einen Eintrag dieser Struktur.

Beispiel für eine Applikation:

```
static CONST tBlCopFlashInfo ROM
   aBlCopFlashAreas_l[BLCOP_MAX_PROGRAMS] =
{
    {BLCOP_APPLICATION_START_AREA1, BLCOP_APPLICATION_SIZE_AREA1},
};

tBlCopFlashInfo GENERIC* PUBLIC TgtGetFlashInfo (void)
{
    return (tBlCopFlashInfo GENERIC*) &aBlCopFlashAreas_l[0];
}
```

Funktion TgtGetAppSize

```
DWORD PUBIC TgtGetAppSize(tFastByte bAppAreaNumber_p);
```

Bedeutung:
Die Funktion TgtGetAppSize() liest die tatsächliche Größe einer zuvor programmierten Applikation aus dem nichtflüchtigen Speicher des Targets.

Parameter:

bAppAreaNumber_p:	Index der Applikation, deren tatsächliche Applikationsgröße zurückgegeben werden soll.

Rückgabe:
Die Funktion liefert die tatsächliche Größe der zuvor programmierten Applikation aus dem nichtflüchtigen Speicher zurück.

Diese Funktion muss nicht prüfen, ob der Wert im nichtflüchtigen Speicher gültig ist. Denn diese Funktion wird nur dann vom Bootloader gerufen, wenn die Signatur der Applikation im nichtflüchtigen Speicher gültig ist (siehe Funktion TgtCheckAppSig()).

Funktion TgtSetAppSize

```
void PUBLIC TgtSetAppSize (DWORD dwSize_p,
    tFastByte bAppAreaNumber_p);
```

Bedeutung:
Die Funktion TgtSetAppSize() wird vom Bootloader gerufen, um die Größe der programmierten Applikation in einem nichtflüchtigen Bereich des Targets abzulegen. Der Wert wird benötigt, um vor dem Starten der Applikation die CRC für den Applikationsbereich zu berechnen.

Parameter:

dwSize_p:	Applikationsgröße für die Ablage im nichtflüchtigen Speicher
bAppAreaNumber_p:	Index der Applikation, deren Applikationsgröße in den nichtflüchtigen Speicher abgelegt werden soll.

Rückgabe:
keine

Funktion TgtGetAppCrc

```
void PUBLIC TgtGetAppCrc (DWORD GENERIC *pdwCrc_p,
    tFastByte bAppAreaNumber_p);
```

Bedeutung:
Die Funktion liest die CRC der Applikation aus dem nichtflüchtigen Speicher des Targets. Der Wert wurde auf der Host-Seite berechnet und beim Programmieren der Applikation mit Hilfe der Funktion TgtSetAppCrc() im nichtflüchtigen Speicher hinterlegt. Der Wert wird benötigt, um für das Starten der Applikation die ermittelte CRC mit dem Wert der Host-Seite zu vergleichen.

Parameter:

pdwCrc_p:	Zeiger auf die Variable für die Rückgabe der CRC
bAppAreaNumber_p:	Index der Applikation, deren CRC zurückgegeben werden soll.

Rückgabe:
Die Funktion liefert die auf der Host-Seite berechnete und im nichtflüchtigen Speicher hinterlegte CRC über den Pointer pdwCrc_p zurück.

Funktion TgtSetAppCrc

```
void PUBLIC TgtSetAppCrc (DWORD dwCrc_p,
    tFastByte bAppAreaNumber_p);
```

Bedeutung:
Die Funktion hinterlegt die CRC im nichtflüchtigen Speicher des Targets. Die CRC wurde auf der Host-Seite berechnet und während des Downloads an das Target übermittelt. Diese CRC wird vor dem Start der Applikation mit der im Target tatsächlich berechneten CRC verglichen.

Parameter:

dwCrc_p:	CRC für die Ablage im nichtflüchtigen Speicher
bAppAreaNumber_p:	Index der Applikation, deren CRC im nichtflüchtigen Speicher abgelegt werden soll.

Rückgabe:
keine

Funktion TgtCheckAppSig

```
BYTE PUBLIC TgtCheckAppSig (tFastByte bAppAreaNumber_p);
```

Bedeutung:
Der Programmierzustand einer Applikation wird im nichtflüchtigen Speicher abgelegt, damit der Bootloader prüfen kann, ob eine Applikation programmiert wurde oder nicht. Eine gültige Signatur ist Voraussetzung, um die Applikation zu starten. Die Signatur wird vom Bootloader mittels der Funktion TgtSetAppSig() auf Anforderung vom Host in den nichtflüchtigen Speicher geschrieben, nach dem die Applikation verifiziert wurde. Die Funktion TgtCheckAppSig() überprüft, ob eine gültige Signatur hinterlegt ist.

Parameter:

bAppAreaNumber_p:	Index der Applikation, deren Signatur aus dem nichtflüchtigen Speicher geprüft werden soll.

Rückgabe:

TRUE:	Signatur gültig
FALSE:	Signatur ungültig

Funktion TgtClrAppSig

```
BYTE PUBLIC TgtClrAppSig (tFastByte bAppAreaNumber_p);
```

Bedeutung:
Vor dem Programmieren einer neuen Applikation ruft der Bootloader diese Funktion, um eine zuvor programmierte Applikation ungültig zu machen. Diese Funktion muss die Signatur im nichtflüchtigen Speicher löschen.

Parameter:

bAppAreaNumber_p:	Index der Applikation, deren Signatur im nichtflüchtigen Speicher löschen soll.

Rückgabe:

TRUE:	Signatur erfolgreich gelöscht
FALSE:	Fehler beim Löschen der Signatur

Funktion TgtSetAppSig

```
BYTE PUBLIC TgtSetAppSig (tFastByte bAppAreaNumber_p);
```

Bedeutung:
Wenn der Download einer Applikation abgeschlossen ist, wird zunächst die CRC über die Applikation berechnet und mit der CRC verglichen, die auf dem Host berechnet wurde. Stimmt diese CRC, wird durch Aufruf dieser Funktion eine Signatur im nichtflüchtigen Speicher abgelegt, damit der Bootloader nach dem nächsten Power-On bwz. Reset weiß, ob eine gültige Applikation programmiert ist.

Parameter:

bAppAreaNumber_p:	Index der Applikation, deren Signatur im nichtflüchtigen Speicher gesetzt werden soll.

Rückgabe:

TRUE:	Signatur erfolgreich gesetzt
FALSE:	Fehler beim Setzen der Signatur

Die Signatur ist ein 32 Bit Wert, den der Anwender über die Konstante BLCOP_APPLICATION_SIGNATURE in der Datei tgtblcop.h definieren kann. Es wird empfohlen, einen Wert zu wählen, der einer Zeichenkette entspricht (z.B. den Wert 0x30505041 für „APP1“).

Funktion TgtGetSerialNr

```
void PUBLIC TgtGetSerialNr (DWORD MEM* pdwSerial_p);
```

Bedeutung:
Diese Funktion gibt die Seriennummer des CANopen Gerätes zurück, in dem der Bootloader programmiert ist. Die Seriennummer wird benötigt, um das Target innerhalb eines CANopen-Netzwerkes in Verbindung mit Vendor-ID, Produkt-ID und Revision-Code eindeutig zu identifizieren (entspricht dem Identity Object 0x1018 im OD des Bootloaders). Diese Funktion kapselt den Zugriff auf den nichtflüchtigen Speicher des Targets für das Lesen der Seriennummer.

Parameter:

pdwSerial_p:	Pointer auf eine DWORD Variable für die Rückgabe der Seriennummer.

Rückgabe:
keine

Die Ablage der Seriennummer im nichtflüchtigen Speicher wird nicht durch den Bootloader ausgeführt. Es liegt in der Hand des Anwenders die notwendigen Schritte zu implementieren. Im einfachsten Fall ist die Seriennummer des Gerätes eine Konstante, die über das Define BLCOP_IDENTITY_SERIALNR in der Datei blcopcfg.h definiert werden kann. In diesem Fall kann die Funktion TgtGetSerialNr() einfach diese Konstante zurückgeben, ohne den nichtflüchtigen Speicher zu lesen. Bedenken Sie jedoch, dass beim Einsatz des LSS Service (mit Knotenscan) jedes CANopen Gerät eindeutig identifiziert werden können muss. Das heißt, jedes Gerät muss eine eigene eindeutige

© SYS TEC electronic GmbH 2015

Seriennummer haben (Vendor-ID, Produkt-ID und Revision-Code können gleich sein). Andernfalls funktioniert das Scannen der Geräte über den LSS Service nicht.

Funktion TgtProcessAppInfoData

```
DWORD PUBLIC TgtProcessAppInfoData (tFastByte bAppAreaNumber_p,
    BYTE MEM* pbAppInfo_p,
    DWORD     dwSize_p);
```

Bedeutung:
Ist BLCOP_IDENTITY_CHECK in blcopcfg.h auf TRUE gesetzt, dann prüft der CANopen Bootloader beim Download der Applikation die Vendor-ID und Produkt-ID mit den Einträgen aus dem Identity Object 0x1018 im OD. Stimmen diese Werte nicht überein, dann wird die Programmierung der Applikation abgewiesen. Der Anwender kann mit Hilfe dieser Funktion die standardmäßige Prüfung dieser Werte überschreiben.

Parameter:

bAppAreaNumber_p: Index der Applikation, für die der Download begonnen hat.

pbAppInfo_p: Pointer auf die Daten, in denen die Vendor-ID und Produkt-ID hinterlegt sind (*siehe Bild 11*). Diese Daten müssen mit den AMI-Funktionen gelesen werden.

dwSize_p: Anzahl der Datenbytes, die mit dem Parameter pbAppInfo_p adressiert werden. Sind Vendor-ID und Produkt-ID in den Daten gültig hinterlegt, dann ist die Anzahl der Bytes größer oder gleich 12.

Rückgabe:
Der Rückgabecode ist ein Bit-orientierter 32-Bit Wert. Hier ist zu hinterlegen, ob die oben genannten Daten ausgewertet worden sind, ob ein Fehler dabei aufgetreten ist, und welcher Fehlercode. Die einzelnen Bits sind für die Rückgabe mit dem logischen ODER-Operator zu verknüpfen.

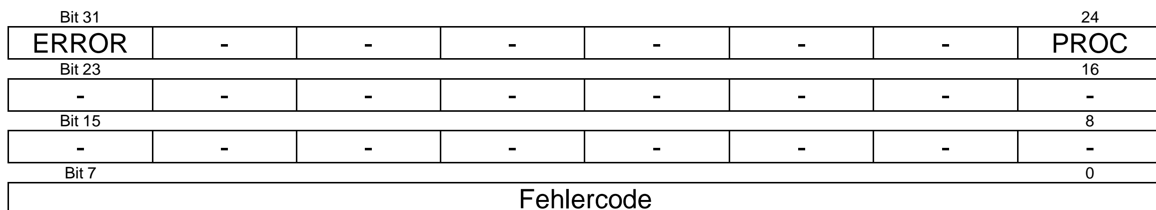


Bild 10: Bitpositionen für den Rückgabecode von TgtProcessAppInfoData

Define	Wert	Bedeutung
BLCOP_PROCFLAG_PROCECCED	0x01000000	Die Auswertung wurde durchgeführt.
BLCOP_PROCFLAG_ERROR	0x80000000	Es ist ein Fehler aufgetreten. Der Fehlercode gibt nähere Informationen an.
BLCOP_STATERRMAN_WRONG_VID	0x00000040	Fehlercode: Die Vendor-ID stimmt nicht mit 0x1018/1 überein.
BLCOP_STATERRMAN_WRONG_PID	0x00000041	Fehlercode: Die Produkt-ID stimmt nicht mit 0x1018/2 überein.

Tabelle 6: Defines für den Rückgabecode von TgtProcessAppInfoData

Kehrt diese Funktion zurück, ohne dass das Bit BLCOP_PROCFLAG_PROCECCED gesetzt ist, dann führt der Bootloader die standardmäßige Prüfung von Vendor-ID und Produkt-ID durch.

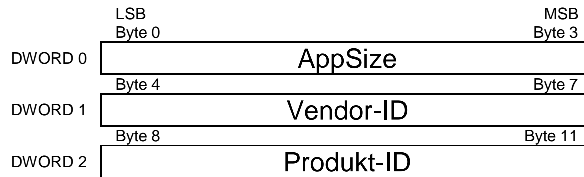


Bild 11: Lage von Vendor-ID und Produkt-ID in den Daten

Hinweis:
Zum Zeitpunkt der Prüfung von Vendor-ID und Produkt-ID ist eine zuvor programmierte Applikation bereits durch den Bootloader gelöscht worden. Es gibt jedoch eine Möglichkeit, das Löschen der vorigen Applikation zeitlich zu verschieben. Dazu ruft der Bootloader vor dem Löschvorgang die Funktion TgtCheckEraseApp() und nach dem Empfang des Blocks 0 die Funktion TgtCheckEarseReApp() auf (nur wenn BLCOP_IDENTITY_CHECK auf TRUE gesetzt ist). Bild 12 zeigt den prinzipiellen Ablauf für das zeitlich verschobene Löschen einer zuvor programmierten Applikation.

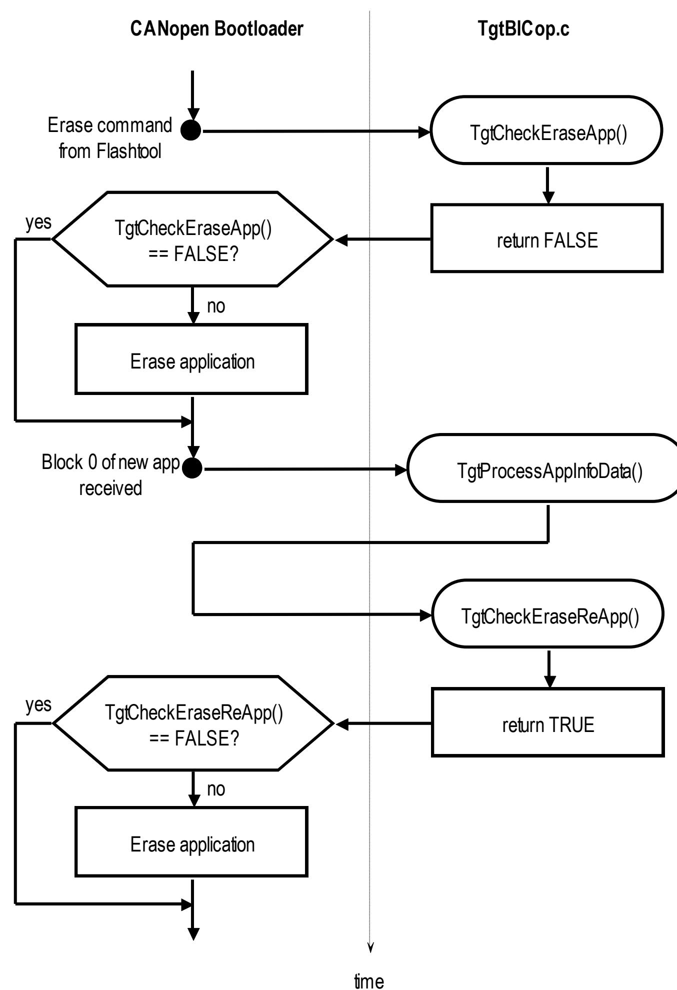


Bild 12: Prinzipieller Ablauf des verschobenen Löschens der Applikation

Funktion TgtCheckEraseApp

```
void BYTE PUBLIC TgtCheckEraseApp (tFastByte bAppAreaNumber_p);
```

Bedeutung:
Ist die Konstante BLCOP_IDENTITY_CHECK in blcopcfg.h auf TRUE gesetzt, dann ruft der CANopen Bootloader nach dem Empfang des Erase-Kommandos (für das Löschen einer zuvor programmierten Applikation) die Funktion TgtCheckEarseApp(). Gibt diese Funktion FALSE zurück, dann löscht der Bootloader die alte Applikation an dieser Stelle nicht (*siehe Bild 12*). Nach dem Empfang des ersten Blocks (Block 0) für die neue Applikation ruft der Bootloader die Funktion TgtCheckReEarseApp(), um festzustellen, ob die Applikation an dieser Stelle gelöscht werden soll.

Parameter:
bAppAreaNumber_p: Index der Applikation, die gelöscht werden soll.

Rückgabe:
FALSE: alte Applikation wird nicht gelöscht
TRUE: alte Applikation wird gelöscht

Funktion TgtCheckReEarseApp

```
void BYTE PUBLIC TgtCheckReEarseApp (tFastByte bAppAreaNumber_p);
```

Bedeutung:
Ist die Konstante BLCOP_IDENTITY_CHECK in blcopcfg.h auf TRUE gesetzt, dann ruft der CANopen Bootloader nach dem Empfang des ersten Blocks (Block 0) die Funktion TgtCheckReEarseApp(). Gibt diese Funktion FALSE zurück, dann löscht der Bootloader die alte Applikation an dieser Stelle nicht (*siehe Bild 12*).

Parameter:
bAppAreaNumber_p: Index der Applikation, die gelöscht werden soll.

Rückgabe:
FALSE: alte Applikation wird nicht gelöscht
TRUE: alte Applikation wird gelöscht

4.5 Rücksprung in Bootloader

Der Rücksprung der Applikation in den Bootloader wird mit dem zusätzlichen Objekt 0x1F51 im OD realisiert, den die Applikation implementieren muss. Die Definition in der Datei objdict.h für die Applikation sieht folgendermaßen aus:

```
OBD_BEGIN_INDEX_ROM(0x1F51, 0x02, AppCbProgramCtrl)
  OBD_SUBINDEX_ROM_VAR(0x1F51, 0x00, kObdTypUInt8, kObdAccR,
                     tObdUnsigned8, number_of_entries, 1)
  OBD_SUBINDEX_RAM_EXTVAR_NOINIT(0x1F51, 0x01, kObdTypUInt8,
                     kObdAccRW, tObdUnsigned8,  abFlsCopProgramCtrl_g[0])
OBD_END_INDEX(0x1F51)
```

Beim Zugriff der Flashtools auf dieses Objekt ruft der CANopen Stack die oben angegebene Callback-Funktion AppCbProgramCtrl() auf. Der Anwender muss diese Funktion ausfüllen, wie es im folgenden Abschnitt beschrieben ist.

Funktion AppCbProgramCtrl

```
tCopKernel PUBLIC AppCbProgramCtrl (
    tObdCbParam MEM* pParam_p);
```

Bedeutung:
Diese Callback-Funktion ist in der Applikation zu implementieren, um aus der Applikation einen Rücksprung in den Bootloader zu realisieren. Sie wird gerufen, wenn das Flashtool auf dem PC einen SDO-Zugriff auf das Objekt 0x1F51 durchführt. Schreibt das Flashtool den Wert 0x80 auf das Objekt, dann muss der Bootloader gestartet werden.

Parameter:

pParam_p:	Pointer auf die Struktur tObdCbParam mit näheren Informationen über den SDO-Zugriff (*siehe /3/*).

Rückgabe:
Gibt diese Funktion kCopSuccessful zurück, wird dem CANopen Stack ein erfolgreicher SDO-Zugriff signalisiert. Damit wird auch eine positive SDO-Response an das Flashtool zurückgesendet. Jeder andere Fehlercode (nach Definition der Fehlercodes der CANopen-API – siehe /3/) führt zu einem SDO-Abort. In diesem Fall ist der Parameter pParam_p->m_dwAbortCode mit dem entsprechenden SDO-Abort Code nach /2/ zu übergeben.

```
tCopKernel PUBLIC AppCbProgramCtrl (tObdCbParam MEM* pParam_p)
{
  // check the subindex
  if (pParam_p->m_bSubIndex > 0)
  {
    // check the event
    if (pParam_p->m_ObdEvent == kObdEvPostWrite)
    {
      // check control command
      if (*((BYTE GENERIC*) pParam_p->m_pArg) == 0x80)
      {
        TgtStartBootloader ();
      }
    }
  }

  return kCopSuccessful;
}
```

Hinweis:
Der Bootloader sollte im Event kObdEvPostWrite angesprungen werden. Zu diesem Zeitpunkt hat der SDO-Server das Senden der SDO-Response auf den laufenden SDO-Zugriff bereits angewiesen. Unter Umständen kann es aber sein, dass diese SDO-Response noch nicht auf den CAN-Bus gesendet wurde. Das würde dazu führen, dass das Flashtool auf dem PC einen SDO-Timeout beim Starten der Applikation meldet. Um dies zu verhindern, sollte innerhalb der Funktion TgtStartBootloader() ein paar Millisekunden gewartet wird (ca. 10ms – abhängig von der CAN-Bitrate), bevor der CAN-Interrupt gesperrt bzw. bevor ein Reset für den Rücksprung in die Applikation ausgelöst wird.

Funktion TgtStartBootloader

```
void TgtStartBootloader (void);
```

Bedeutung:
Diese Funktion ist in der Applikation zu implementieren, um aus der Applikation einen Rücksprung in den Bootloader zu realisieren. Vor einem Rücksprung sind durch die Applikation sämtliche Ressourcen freizugeben und der globale Interrupt zu sperren.

Parameter:
keine

Rückgabe:
keine

Der Rücksprung kann auf verschiedene Art und Weise realisiert werden. Lesen Sie dazu auch das Kapitel 3.4.7.

4.6 Schnittstelle zum CAN-Bus

Diese Schnittstelle basiert auf den SYS TEC CAN-Treibern für CANopen. Für nähere Informationen zu den CAN-Treiber Funktionen lesen Sie bitte das Handbuch zum CAN-Treiber L-1023.

In Tabelle 7 werden die CAN-Controller bzw. CPUs aufgelistet, auf denen der Bootloader bereits portiert wurde.

Anbieter und CPU	CAN-Controller
Infineon XC16x	Interner MultiCAN
Freescale PowerPC	Interner TouCAN
Microchip dsPIC33F	Interner ECAN
Microchip PIC32MX	Interner CAN des PIC32MX
NXP LPC21xx, LPC22xx und LPC23xx	interner CAN des LPC, virtuelle CAN-Schnittstelle über UART
ST Microelectronic STM32F1xx und STM32F2xx	Interner CAN des STM32
Silicon Laboratories C8051Fxxx	Interner Bosch CAN
Fujitsu MB90F352S	Interner CAN der 350 CPU
Renesas RX	interner CAN des RX, virtuelle CAN-Schnittstelle über UART

Tabelle 7: Unterstützte CPUs und CAN-Controller

4.7 Debug-Ausgaben

Innerhalb des Sources werden für die Ausgaben von Programmstatus-Informationen printf-Funktionen aufgerufen (durch Verwendung des TRACE-Makros), mit deren Hilfe Ausgaben über eine serielle Schnittstelle möglich sind. Hierfür wird eine entsprechende Funktion benötigt, die die Schnittstelle initialisiert. Diese Funktion ist unten dargestellt. Darüber hinaus müssen bei manchen CPUs/IDEs die Ausgaben der printf-Funktion auf die UART umgeleitet werden. Der Mechanismus, der dafür verwendet werden muss, hängt vom verwendeten System ab. Oftmals muss nur die Funktion putchar() überschrieben werden.

Funktion TgtInitSerial

```
void PUBLIC TgtInitSerial (void);
```

Bedeutung:
Diese Funktion initialisiert die serielle Schnittstelle für Debug-Ausgaben über die printf-Funktion.

Parameter:
keine

Rückgabe:
keine

4.8 Konfiguration des Bootloaders

Für die Konfiguration des Bootloaders gibt es zwei Dateien. In der Datei blcopcfg.h werden allgemeine Konfigurationen vorgenommen und spezielle, die auf den Code des Bootloaders wirken. In der Datei tgtblcop.h werden Target-spezifische Konfigurationen vorgenommen.

4.8.1 Konfigurationen in der Datei blcopcfg.h

Die Datei blcopcfg.h liegt im Projektverzeichnis des Bootloaders. Gibt es mehrere Bootloader Projekte, dann hat jedes Projekt eine eigene Datei blcopcfg.h, deren Inhalte unterschiedlich sein können.

BLCOP_MAX_PROGRAMS:

Diese Konstante gibt die Anzahl der Applikationen an, die mit dem Bootloader in den Flash programmiert werden können. Der Wertebereich liegt zwischen 1 und 254 und wird frühestens durch die Größe des Flashspeichers begrenzt.

BLCOP_MAX_PROGRAM_BUFFER:

Diese Konstante gibt die Anzahl der Bytes an, die mit einem Block an den Bootloader übertragen werden können. Die Blockinformationen (Blocknummer, Flash-Adresse, Anzahl der Datenbytes und Block-CRC) wird mit einbezogen. Es wird empfohlen, dass diese Größe des Puffers ein ganzzahliges Vielfaches der Konstante BLCOP_MAX_FLSWRITE_STEP_SIZE darstellt plus 16 Bytes für die zusätzlichen Blockinformationen. Der gleiche Wert ist dem Tool BinaryBlockConv.exe mit dem Parameter --block_size zu übergeben, um das HEX-File der Applikation in das binäre Format zu konvertieren.

BLCOP_MIN_NODEID:

Diese Konstante gibt die kleinste CANopen Knotenadresse an, die vom Bootloader unterstützt wird. Der kleinstmögliche Wert für diese Konstante ist 1 (begrenzt durch den CANopen Standard CiA-301). Weitere Informationen sind bei der Beschreibung des Defines BLCOP_MAX_NODEID angeben.

BLCOP_MAX_NODEID:

Diese Konstante gibt die größte CANopen Knotenadresse an, die vom Bootloader unterstützt wird. Der größtmögliche Wert für diese Konstante ist 127 (begrenzt durch den CANopen Standard CiA-301). Sind die Werte von BLCOP_MIN_NODEID und BLCOP_MAX_NODEID unterschiedlich, dann prüft die Funktion BlCopInitialize() die mit dem Parameter bNodeId_p übergebene Knotenadresse, ob diese in dem angegebenen Bereich liegt. Ist dies nicht der Fall, kehrt BlCopInitialize() mit dem Fehlercode kBlCopInitError zurück. Sind die Werte in diesen Konstanten gleich, dann wird keine Überprüfung der Knotenadresse durchgeführt.

BLCOP_MIN_BAUIDX:

Diese Konstante gibt den kleinsten Baudratenindex an, der vom Bootloader unterstützt wird. Es können Werte von kBdi10kBaud bis kBdi1MBaud angegeben werden. Die möglichen Werte können aus dem CAN-Treiber Manual L-1023 entnommen werden. Weitere Informationen sind bei der Beschreibung des Defines BLCOP_MAX_BAUIDX.

BLCOP_MAX_BAUIDX:

Diese Konstante gibt den größten Baudratenindex an, der vom Bootloader unterstützt. Sind die Werte von BLCOP_MIN_BAUIDX und BLCOP_MAX_BAUIDX nicht gleich, dann prüft die Funktion BlCopInitialize() die mit dem Parameter bBaudIdx_p übergebene CAN-Bitrate, ob sie im angegebenen Bereich liegt. Ist dies nicht der Fall, kehrt BlCopInitialize() mit dem Fehlercode kBlCopInitError zurück. Sind die Werte in diesen Konstanten gleich, dann wird keine Überprüfung der CAN-Bitrate durchgeführt.

Achtung: Ein kleinerer Wert für den Bitraten-Index entspricht einer größeren Bitrate auf dem CAN-Bus. Das muss bei der Festlegung von BLCOP_MIN_BAUIDX und BLCOP_MAX_BAUIDX beachtet werden. Soll also eine Überprüfung durchgeführt werden, dann muss BLCOP_MIN_BAUIDX eine höhere Bitrate zugeordnet sein als BLCOP_MAX_BAUIDX.

BLCOP_SEND_BOOTUP:
Diese Konstante gibt an, ob die CANopen Bootup-Nachricht nach CiA-301 gesendet werden soll, oder nicht. Ist dieses Define auf FALSE gesetzt, wird keine Bootup-Nachricht gesendet.

BLCOP_BDI_TABLE_PTR:
Diese Konstante gibt an, welche Baudratentabelle verwendet werden soll. Die Baudratentabelle enthält Registerwerte für den CAN-Controller, die für die jeweiligen Bitraten-Indizes eingestellt werden. Diese Registerwerte unterscheiden sich, wenn eine andere Quarzfrequenz-Frequenz verwendet wird, oder wenn bei CPU internen CAN-Controllern die PLL der CPU anders eingestellt wird. Eine Neuberechnung der Registerwerte ist unter Umständen notwendig.

BLCOP_BDI_TABLE_SIZE:
Diese Konstante gibt die Größe der Baudratentabelle in Bytes an. Verwenden Sie die sizeof-Direktive mit dem Namen der Baudratentabelle als Parameter, um den Wert für diese Konstante festzulegen.

BLCOP_BASE_REQUEST:
Diese Konstante gibt den Basis-CAN-Identifier für den SDO-Request (SDO-Client → SDO-Server) an, der in Verbindung mit der CANopen Knotenadresse den tatsächlichen CAN-Identifier bildet. Nach CiA-301 ist dieser Konstante der Wert 0x600 zu übergeben.

BLCOP_BASE_RESPONBSE:
Diese Konstante gibt den Basis-CAN-Identifier für die SDO-Response (SDO-Client ← SDO-Server) an, der in Verbindung mit der CANopen Knotenadresse den tatsächlichen CAN-Identifier bildet. Nach CiA-301 ist dieser Konstante der Wert 0x580 zu übergeben.

BLCOP_USE_CANCRTL:
Manche CPUs besitzen mehrere CAN-Schnittstellen. Diese Konstante gibt an, welche dieser CAN-Schnittstellen vom Bootloader verwendet werden soll. Der Wert 0 bestimmt immer die erste CAN-Schnittstelle.

BLCOP_USE_CANINTENABLE:
Diese Konstante gibt an, welche Funktion gerufen werden soll, um den CAN-Interrupt kurzzeitig zu sperren.

BLCOP_MAX_CANLOOPS:
Diese Konstante gibt an, wie viele CAN-Nachrichten maximal aus dem Empfangspuffer des CAN-Treibers ausgewertet werden sollen. Da das Löschen der Flash-Sektoren relativ lange dauert, sollte dieser Wert nicht auf 1 stehen.

BLCOP_IDENTITY_VENDORID:
Diese Konstante gibt den 32-Bit Wert für die Vendor-ID an, die vom Bootloader in das Objekt 0x1018/1 geschrieben wird. Die Vendor-ID wird von CiA vergeben. Steht keine Vendor-ID zur Verfügung, dann muss dieser Wert 0 erhalten.

BLCOP_IDENTITY_PRODUCTID:
Diese Konstante gibt den 32-Bit Wert für die Produkt-ID an, die vom Bootloader in das Objekt 0x1018/2 geschrieben wird. Die Produkt-ID ist ein Merkmal für das CANopen Gerät und kann vom Anwender selbst vergeben werden.

BLCOP_IDENTITY_REVISION:
Diese Konstante gibt die 32-Bit Revisionsnummer an, die vom Bootloader in das Objekt 0x1018/3 geschrieben wird. Der Wert ist in dem Format anzugeben, wie er im CANopen Standard CiA-301 definiert ist. Der Wert 0x00010002 würde zum Beispiel die Version V1.02 angeben.

BLCOP_IDENTITY_SERIALNR:
Diese Konstante gibt die 32-Bit Seriennummer an, die vom Bootloader in das Objekt 0x1018/4 geschrieben wird. Jedes Gerät muss laut CANopen Standard eine eindeutige LSS-Adresse besitzen. Zur LSS-Adresse gehören Vendor-ID, Produkt-ID, Revisionsnummer und Seriennummer. Stellt der Anwender eine Serie von Geräten her, in dem der Bootloader

programmiert wird, dann sind Vendor-ID, Produkt-ID und Revisionsnummer üblicherweise gleich. Nur über die Seriennummer kann nun jedem Gerät eine eindeutige LSS-Adresse zugeordnet werden. Die Funktion TgtGetSerialNr() kann dazu verwendet werden, die Seriennummer aus einem nichtflüchtigen Speicher zu lesen. Befindet sich das Gerät mit dem Bootloader in einem System, in dem kein LSS Service verwendet wird, dann kann die Seriennummer eine Konstante sein, die für jedes Gerät gleich ist. Dann nutzen Sie diese Konstante, um die Seriennummer der Funktion BlCopInitialize() mit dem Parameter dwSerialNr_p zu übergeben.

BLCOP_IDENTITY_CHECK:

Besteht das Gesamtsystem des Anwenders aus mehreren Geräten, die mit dem CANopen Bootloader verwendet werden, dann ist es oft notwendig, dass auch unterschiedliche Anwendungen in die einzelnen Geräte programmiert werden müssen. Die Konstante BLCOP_IDENTITY_CHECK bringt hierfür mehr Sicherheit gegen Programmierung einer falschen Applikation in ein Gerät. Ist es auf TRUE gesetzt, implementiert der Bootloader zusätzlichen Code für die Prüfung von Vendor-ID und Produkt-ID, die im Identity Objekt 0x1018 im OD hinterlegt sind. Die binäre Datei mit den Programmdaten muss dabei auch die Vendor-ID und Produkt-ID enthalten. Beide Parameter können mit den Aufrufparametern --vid und --pid des Tools BinaryBlockConv.exe übergeben werden.BLCOP_MAX_CRC_STEP_SIZE: Diese Konstante gibt an, wie viele Bytes maximal bei einem Durchlauf der Funktion BlCopProcess() für die Berechnung der CRC durchlaufen werden sollen. Ist dieser Wert zu klein, dauert es unter Umständen zu lange, bis die CRC über die komplette Applikation berechnet ist. Das Starten der Applikation verzögert sich dadurch. Ist dieser Wert zu groß, dann kann der Watchdog zuschlagen, falls dieser verwendet wird. Lesen Sie dazu auch das Kapitel 3.4.8.

BLCOP_MAX_FLSWRITE_STEP_SIZE:

Diese Konstante gibt an, wie viele Bytes maximal bei einem Durchlauf der Funktion BlCopProcess() für das Schreiben der Daten in den Flash durchlaufen werden sollen (mit einem Aufruf der Funktion FlsDrvWriteData()). Dieser Wert sollte ein ganzzahliges Vielfaches einer programmierbaren Flash-Page sein. Das vereinfacht die Implementierung der Flash-Treiber Funktionen. Ist der Wert für BLCOP_MAX_FLSWRITE_STEP_SIZE zu groß, dann kann der Watchdog zuschlagen, falls dieser verwendet wird. Lesen Sie dazu auch das Kapitel 3.4.8.

BLCOP_USE_AUTOSTART_APP:

Ist diese Konstante auf TRUE gesetzt, dann startet der CANopen Bootloader nach dem Power-On automatisch eine zuvor gültige Applikation. Verwaltet der Bootloader mehrere Applikationen, dann wird in der Konstante BLCOP_APP_START_INDEX der Index der Applikation konfiguriert, der nach dem Power-On automatisch gestartet werden soll.

BLCOP_APP_START_INDEX:

Wenn der CANopen Bootloader startet, prüft er zunächst, ob eine gültige Applikation programmiert ist (d.h. auch, ob die Applikations-CRC korrekt ist). Ist die Konstante auf TRUE gesetzt, dann wird diese gültige Applikation nach dem Start des Bootloaders gestartet. Wenn der Bootloader mehrere Applikationen verwaltet, benötigt er die Information, welche der Applikationen automatisch gestartet werden soll. Das wird mit der Konstante BLCOP_APP_START_INDEX festgelegt. Hier ist der Index der Applikation anzugeben, wie er auch in der Tabelle für die Flash-Informationen als Index verwendet wird. Index 0 bedeutet dabei die erste Applikation und entspricht dem Subindex 1 aus dem Objekt 0x1F50 (usw.).

BLCOP_USE_AMR_ACR_FILTER:

Wenn diese Konstante auf TRUE gesetzt wird, dann füllt der CANopen Bootloader die Parameter AMR und ACR für die Hardware-seitige Filterung der zu empfangenden CAN-Nachrichten aus. Solange der CAN-Treiber und der CAN-Controller dieses Feature unterstützen, wird vom Bootloader nur die CAN-Nachricht für den eigenen SDO-Server empfangen. Das ist oft sehr hilfreich, wenn zum Zeitpunkt des Programmdownloads andere CANopen Geräte angeschlossen sind, die eine erhöhte Datenübertragung über andere CAN-Identifier aufweisen.

BLCOP_USE_WATCHDOG:

Soll ein Watchdog verwendet werden, dann muss diese Konstante auf TRUE gesetzt werden. Der CANopen Bootloader ruft dann zyklisch die Funktion TgtWatchdogProcess() auf. Nähere Informationen zu diesem Thema finden Sie im Kapitel 3.4.8.

BLCOP_USE_DECRYPT:

Aktiviert die Entschlüsselung eines verschlüsselten Firmware-Downloads. In diesem Zusammenhang muss die Datei decrypt.c zum Projekt des CANopen Bootloaders hinzugefügt werden (siehe Kapitel 3.4.10).

BLCOP_DECRYPT_SUBTRAHEND:

Mit aktivierter Firmware-Entschlüsselung wird mit dieser Konstante der Startwert für den Verschlüsselungsalgorithmus festgelegt (siehe Kapitel 3.4.10).

In der Datei blcopcfg.h können weitere Konfigurationen angelegt sein, auf die in diesem Handbuch nicht weiter eingegangen werden soll. Spezielle Implementierungen wie z.B. das Callgate sind immer spezifisch für ein konkretes Projekt. Wir unterstützen Sie gern im Rahmen eines gemeinsamen Workshops bei der Umsetzung solcher Implementierungen.

4.8.2 Konfigurationen in der Datei tgtblcop.h

Die Datei tgtblcop.h liegt außerhalb des Projektverzeichnis des Bootloaders und kann gemeinsam von mehreren Bootloader-Projekten für eine Hardware verwendet werden. Müssen diese Einstellungen in den einzelnen Bootloader-Projekten unterschiedlich sein, dann muss der Anwender diese Konstanten mit einer bedingten Übersetzung definieren (Verwendung der #if-Direktive).

BLCOP_BOOTLOADER_START_AREA:
Diese Konstante gibt die Startadresse im Flash an, an der der Bootloader beginnt.

BLCOP_BOOTLOADER_END_AREA:
Diese Konstante gibt die letzte Adresse im Flash an, der für den Bootloader reserviert ist. Damit der Bootloader separat in den Flash programmierbar ist, muss dieser Wert an einem vollen Flash-Sektor enden. Bei der Entwicklung des Bootloaders wird empfohlen, die Debug-Ausgaben über die UART zu aktivieren, so dass Fehler oder andere Probleme zur Laufzeit erkannt werden können. Mit diesen Debug-Ausgaben nimmt der Bootloader mehr Code-Speicher ein, als ohne. Das sollte bei der Festlegung der Flash-Bereiche beachtet werden.

BLCOP_APPLICATION_START_AREA1:
Diese Konstante gibt die Startadresse im Flash an, an der die erste Applikation beginnt. Diese Adresse ist die Basis für das Aufrufen des Reset-Vektors der Applikation. Sie wird auch in der Tabelle hinterlegt, die mit der Funktion TgtGetFlashInfo() an den Bootloader übergeben wird. Soll der CANopen Bootloader mehrere Applikationen verwalten, dann müssen mehrere dieser Konstanten mit fortlaufender Nummerierung angelegt werden (BLCOP_APPLICATION_START_AREA2 usw.). Die Endadressen der Applikationsbereiche sind jeweils in der Konstante BLCOP_APPLICATION_END_AREA1 usw. zu definieren.

BLCOP_APPLICATION_END_AREA1:
Diese Konstante gibt die letzte Adresse im Flash an, die für die erste Applikation reserviert ist. Die Endadresse wird in der Tabelle hinterlegt, die mit der Funktion TgtGetFlashInfo() an den Bootloader übergeben wird. Soll der CANopen Bootloader mehrere Applikationen verwalten, dann müssen mehrere dieser Konstanten mit fortlaufender Nummerierung angelegt werden (BLCOP_APPLICATION_END_AREA2 usw.).

Die Datei tgtblcop.h kann weitere Definitionen enthalten, die spezifisch für das verwendete Target sind. Zum Beispiel die Ablage der Bootloader-spezifischen Parameter (Knotenadresse, CAN-Bitrate, Größe der Applikation(en), CRC der Applikationen(en) usw.) im nichtflüchtigen Speicher ist abhängig vom verwendeten Speichermedium und dessen Treiber (eigener Flash-Sektor, EEPROM, usw.). Deshalb wird in diesem Dokument nicht weiter auf diese Definitionen eingegangen. Es ist Aufgabe des Anwenders, diese Funktionalitäten zu implementieren.

4.9 Hinweise zur Reduzierung des Code-Bedarfs des Bootloaders

Oft besteht die Bedingung, dass der Bootloader in einem Target möglichst wenig Code-Speicher einnimmt. Dieses Kapitel beschreibt Maßnahmen, die zur Reduzierung des Code-Bedarfs führen.

4.9.1 Optimierungen des CAN-Treibers:

Der CANopen Bootloader füllt automatisch die Parameter AMR und ACR für eine Hardware-seitige Filterung durch den CAN-Controller aus, wenn die Konstante BLCOP_USE_AMR_ACR_FILTER auf TRUE gesetzt ist. Vorausgesetzt, dass der CAN-Treiber diese Parameter in den CAN-Controller programmiert, empfängt damit der Bootloader nur die CAN-Nachricht für den SDO-Request (von Flashtool zum Target). Eine Software-seitige Filterung ist damit nicht notwendig. Um diese auszuschalten, muss in der Datei copcfg.h die Konstante CDRV_USE_IDVALID auf FALSE gesetzt werden.

Der CANopen Bootloader verwendet keine hochpriorisierten CAN-Nachrichten. Deshalb kann die Konstante CDRV_USE_HIGHBUFF auf FALSE gesetzt werden.

Wird der SDO-Kanal für die Übertragung des Programmcodes mit den in CiA-301 angegebenen CAN-IDs verwendet (*siehe Konstanten BLCOP_BASE_REQUEST und BLCOP_BASE_RESPONBSE in Kapitel 4.8.1*), dann benötigt der CAN-Treiber nicht dir Möglichkeit, CAN-Nachrichten mit 29 Bit CAN-IDs (CAN 2.0 B) zu verarbeiten. In diesem Fall kann die Konstante CDRV_CAN_SPEC auf CAN20A gesetzt werden.

In manchen CAN-Treibern ist die Verarbeitung der RTR-Nachrichten (Remote Request) abschaltbar. Diese sind für den CANopen Bootloader ebenfalls nicht relevant. Wenn sie abschaltbar sind, dann kann die Konstante CDRV_IMPLEMENT_RTR auf FALSE gesetzt werden.

Sendeseitig sendet der CANopen Bootloader nur die SDO-Response und (falls aktiviert) die Emergency-Nachricht. Damit ist kein hohes Aufkommen an CAN-Nachrichten zu erwarten. Daher ist es hilfreich für den Code- und RAM-Bedarf, wenn der CAN-Treiber keinen Sendepuffer verwendet. Sollte der CAN-Treiber dies unterstützen, dann kann die Konstante CDRV_USE_NO_TXBUFF auf TRUE gesetzt werden.

4.9.2 Optimierung des Timers

Manche CPUs erzeugen für 16-Bit Variablen weniger Code als für 32-Bit Variablen. Für den Timer wird standardmäßig eine 32-Bit Variable verwendet. Alle Timeout-Überprüfungen im Bootloader sind dann auch 32-Bitig. Die Umstellung auf 16 Bit erfolgt in der Datei copcfg.h, indem die Konstante COP_USE_SMALL_TIME auf TRUE gesetzt wird. Da die Auflösung des Timers 100µs beträgt, würde ein Überlauf der Timer-Variable nach 2^16*100µs = 6,5536 Sekunden auftreten. Timeouts in dieser Größenordnung für den SDO-Transfer sind nicht sinnvoll. Daher kann ohne Bedenken auf 16 Bit umgestellt werden.

Hinweis:
Die Timeout-Überwachungen im CANopen Bootloader sind so implementiert, dass ein Überlauf zu keinen Problemen führen kann. Es wird immer eine Differenz zwischen aktueller Zeit mit einem Startwert gebildet. Das heißt, im Normalfall wird von einem größeren Wert ein kleinerer Wert abgezogen und dann mit einem Timeout-Wert verglichen. Sollte die Timer-Variable gerade übergelaufen sein, wird von einem sehr kleinen Wert ein sehr großer Wert abgezogen. Da die Variable für den Timer immer ohne Vorzeichen ist, ist diese Differenz auch immer positiv und enthält die tatsächliche Zeitdifferenz.

4.9.3 Optimierungen des OBD-Moduls

Die Konstante OBD_SUPPORTED_OBJ_SIZE in copcfg.h bestimmt die maximale Größe eines Objekts im Objektverzeichnis. Das Objekt 0x1F50 ist ein Objekt vom Typ Domain für den Empfang der Programmdaten. Diese Domain wird mit einem Puffer im RAM verknüpft, deren Größe über die Konstante BLCOP_MAX_PROGRAM_BUFFER festgelegt wird (*siehe Kapitel 4.8.1*). Ein kleinerer Puffer als 256 Bytes wird für diesen Puffer nicht eingesetzt werden. Ebenso wird in keinem Bootloader-Projekt ein Puffer größer als 65536 Bytes eingestellt werden. Deshalb ist die beste Einstellung für diese Konstante der Wert OBD_OBJ_SIZE_MIDDLE (d.h. bis zu 65536 Bytes).
Das OBD-Modul enthält spezielle Funktionen, die vom Standard CANopen Stack nur bestimmte Module rufen, die aber im CANopen Bootloader nicht enthalten sind. Diese Funktionen können mit den folgenden Konstanten von der Übersetzung ausgeschlossen werden, wenn diese auf FALSE gesetzt werden:

OBD_IMPLEMENT_PDO_FCT
OBD_IMPLEMENT_ARRAY_FCT
OBD_IMPLEMENT_READ_WRITE
OBD_IMPLEMENT_DEFINE_VAR

4.9.4 Optimierungen im SDO-Server Modul

Wenn auf den SDO-Blocktransfer verzichtet werden kann, dann sollte dieser im CANopen Bootloader deaktiviert werden. Die Deaktivierung erfolgt, indem die Konstante SDO_BLOCKTRANSFER in der Datei copcfg.h auf FALSE gesetzt wird. Der SDO-Segmented Transfer darf für den Bootloader nicht deaktiviert werden. Das Objekt 0x1F50 für den Download der Programmdaten ist eine Domain, die in jedem Fall größer ist als 4 Byte. Damit ist mindestens der SDO-Segmented Transfer notwendig.
Der CANopen Bootloader arbeitet nur mit einem SDO-Server. Ein weiterer SDO-Server ist nicht notwendig. Daher kann die Konstante SDOS_MULTI_SERVER_SUPPORT auf FALSE gesetzt werden.

4.9.5 Optimierung bei der Berechnung der Applikations-CRC

Wie bereits im Kapitel 3.1 beschrieben, wird für die Applikations-CRC (und auch für die CRC der einzelnen Blöcke der Programmdaten – *siehe Bild 2*) eine 32-Bitige CRC mit dem Polynom 0xEDB88320 berechnet. Das Tool BinaryBlcokConv.exe verwendet das gleiche Polynom, wenn es das HEX-File der Applikation in das binäre Block-Format konvertiert. Der Bootloader ruft die Funktion CalcCrc32() aus der Datei crc32.c für die Berechnung, wobei eine Tabelle verwendet wird. Diese Tabelle ermöglicht die schnelle Berechnung der CRC, erzeugt aber zusätzlichen Speicherbedarf. Abhängig vom Define CRC32_RAM_TABLE kann diese Tabelle wahlweise im RAM oder im Flash abgelegt werden. Wenn diese Tabelle in den RAM verlegt werden soll, dann muss in den Präprozessor-Defines des Compilers das Define CRC32_RAM_TABLE angelegt werden (aus Kompatibilitätsgründen kann das nur in den Präprozessor-Defines erfolgen).
Verfügt die CPU über eine interne Peripherie zur Berechnung einer CRC, dann kann diese unter Umständen auch verwendet werden. Lesen Sie dazu das Kapitel 3.1, um nähere Informationen zu erhalten.

5 Flashtool auf dem Host

Die Flashtools auf dem Host unterteilen sich in zwei Teile: Ein Tool zum Konvertieren der Applikation in das binäre Block-Format (*siehe Kapitel 5.1*), der für den Download zum Target verwendet werden kann, und ein Tool für den Download selbst (*siehe Kapitel 5.2*).
Diese Tools sind für das Windows Betriebssystem geschrieben. Lösungen für andere Betriebssysteme (wie Linux), können auf Anfrage erhältlich sein.
Für den Download existiert eine zweites Tool, das auf das .NET Framework aufsetzt (*siehe Kapitel 5.3*).

5.1 Das Tool BinaryBlockConv

Dieses Tool erzeugt aus dem Toolchain-spezifischen Ausgabeformat (Motorola S-Record, Intel Hex-File) ein blockorientiertes Binärformat. Folgende Parameter sind zu übergeben:

Aufruf: BinaryBlockConv [options] input-file [output-file]

Optionen	Default	Bedeutung
-I <format_name>	ihex	Dieser Parameter definiert das Input-File-Format. Mögliche Werte sind: ihex für das Intel HEX-Format; S3, S5 und S9 für Motorola S-Record-Formate.
-O <format_name>	Binary	Dieser Parameter ist reserviert für künftige Anpassungen. Das Ausgabeformat entspricht dem Block-Binärformat wie in diesem Dokument beschrieben.
--start_address <val>	0x00000000	Dieser Wert definiert die Startadresse des Adressbereichs für die Applikation, der bei der Konvertierung zu berücksichtigen ist. Der Wert ist in C-Notation anzugeben.
--end_address <val>	0xFFFFFFFF	Dieser Wert definiert die Endadresse des Adressbereichs für die Applikation, der bei der Konvertierung zu berücksichtigen ist. Werte außerhalb des Bereich *start_address – end_address* werden nicht im Binärfile abgelegt. Der Wert ist in C-Notation anzugeben.
--block_size <val>	0x00000400	Dieser Wert definiert die Anzahl der Bytes pro Binärblock. Er korreliert mit der Größe des Puffers im Bootloader, dessen Größe mit der Konstante BLCOP_MAX_PROGRAM_BUFFER festgelegt wird. Der Wert <val> darf kleiner sein als die Größe des Puffers, aber nicht größer, und ist in C-Notation anzugeben.
--gap_fill <val>	0xFFFFFFFF	Dieser 32-Bit Parameter definiert den Wert, mit dem Lücken innerhalb eines Binärblocks aufgefüllt werden, die keine Programmdaten beinhalten. Da ein gelöschter Flash mit dem Wert 0xFF gefüllt ist, sollte dieser Wert 0xFFFFFFFF beinhalten, damit die Berechnung der Applikations-CRC auf beiden Seiten (Target und Host) den gleichen Wert ergibt.
--adjust_start <val>	0x00000000	Mit diesem Wert kann die Startadresse im Binärblock verändert werden. Der Wert ist in C-Notation anzugeben.

Optionen	Default	Bedeutung
--vid <val>	-	Gibt die Vendor-ID des CANopen Gerätes an, die der Bootloader mit dem Wert aus dem Identity Objekt 0x1018 vergleichen kann. Sind Venodor-ID und Produkt-ID angegeben, dann beinhaltet der Block 0 weitere Daten.
--pid <val>	-	Gibt die Produkt-ID des CANopen Gerätes an, die der Bootloader mit dem Wert aus dem Identity Objekt 0x1018 vergleichen kann. Sind Venodor-ID und Produkt-ID angegeben, dann beinhaltet der Block 0 weitere Daten.
--ver <val>	-	Gibt die Version der Applikation an. Sie wird zusätzlich in die binäre Datei eingetragen, wenn Vendor-ID und Prodkt-ID angegeben wurden. Der Wert für die Version muss in C-Notation als 32-Bit Wert angegeben werden (*siehe Bild 13*). Zusätzlich wird ein Zeitstempel vom POSIX-Typ time_t eingetragen, der das Erstellungsdatum der binären Datei darstellt.
--temp	-	Schreibt eine zusätzliche temporäre Datei mit den Programmdaten der Applikation in den aktuellen Pfad. Diese Datei hat den Namen der Ausgabedatei mit der Erweiterung „.tmp“ und das Format entspricht 1:1 der, wie die Programmdaten in den Flash des Targets geschrieben werden.
--crc16	-	Verwendet statt einer 32-Bit CRC das 16-Bit Polynom 0x1021. Diese 16-Bit CRC wird dennoch 32-Bitig abgelegt mit dem LSB an erster Stelle. Das Target muss ebenfalls dieses Polynom verwenden (*siehe Kapitel 3.1*)
--key <val>	-	Mit diesem Parameter kann ein 128 Bit Schlüssel angegeben werden, mit dem die Firmware-Daten verschlüsselt werden. Wird kein Schlüssel angegeben, dann erfolgt keine Verschlüsselung der Firmware-Daten. Der Schlüssel wird immer hexadezimal ohne führendes ‚0x‘ und ohne Trennung durch Leerzeichen angegeben (siehe Beispiel weiter unten).
--add <val>	0x00	Für den Verschlüsselungsalgorithmus, der mit --key aktiviert wird, wird ein Startwert benötigt. Er dient für eine zusätzliche Sicherheit gegen Entschlüsselung durch Dritte. Dieser Startwert muss der gleiche sein, wie er im CANopen Bootloader mit der Konstanten BLCOP_DECRYPT_SUBTRAHEND definiert ist.
--help	-	Ausgabe der unterstützten Parameter und File-Formate

Tabelle 8: Parameter des Tools BinaryBlockConv

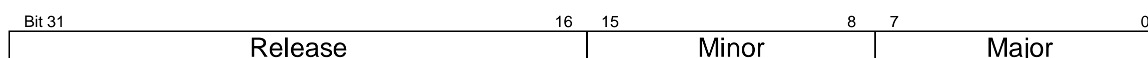


Bild 13: Format des Parameters für die Version

Beispiel:

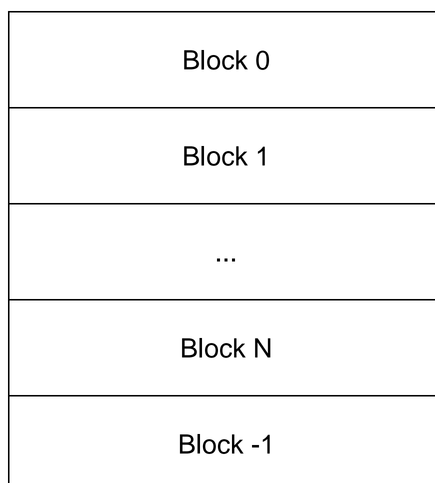
```
BinaryBlockConv –I ihex --block_size 0x110 --start_address 0xFFFC0000
        --end_address 0xFFFFBFFF --vid 0 --pid 1234 --ver 0x00640A01
        blcop_app.hex blcop_app.bin
```

Dieses Beispiel konvertiert die Datei blcop_app.hex mit Intel HEX Format in das binäre Blockformat. Es wird nur der Flash-Bereich 0xFFFC0000 bis einschließlich 0xFFFFBFFF für die Konvertierung berücksichtigt. Weiterhin wird die Vendor-ID 0, Produkt-ID 1234 und die Version V1.10r100 sowie der Erstellungszeitpunkt in die Binärdatei aufgenommen.
Nachfolgend wird das Format der binären Blockdatei dargestellt. Zunächst besteht die Datei aus mehreren Blöcken (0, 1, 2, …, N, -1). Nur die Blöcke 1 bis N enthalten den eigentlichen Programmcode. Block 0 enthält nähere Informationen über die Applikation, falls diese als Aufrufparameter des Tools BinaryBlockConv angegeben worden sind. Der letzte Block hat immer die Nummer -1 (bzw. 0xFFFFFFFF). Dort sind die Größe und die CRC der Applikation hinterlegt. Jeder Block besteht aus einem Block-Header mit Blocknummer, Flash-Adresse und CRC über den Block selbst.
Alle Daten innerhalb der Blöcke sind im Intel-Format abgelegt (d.h. LSB first).

Beispiel mit Verschlüsselung der Firmware-Daten:

```
BinaryBlockConv –I ihex --block_size 0x110 --start_address 0xFFFC0000
         --end_address 0xFFFFBFFF --vid 0 --pid 1234 --ver 0x00640A01
         blcop_app.hex blcop_app.bin --key cf7a7fe196f64c6b8f19e58361d2397e
         --add 0xA5
```

Aufbau der binären Blockdatei:



Aufbau Block 0:

Offset	Parameter	Inhalt
0x0000	Block Number	0
0x0004	Flash Address	0
0x0008	Data Size [bytes]	8
0x000C	Data bytes	reserviert
0x0010		reserviert
0x0014	Block CRC	CRC

Aufbau des Blocks 0 ohne Angabe von Vendor-ID und Produkt-ID.

Offset	Parameter	Inhalt
0x0000	Block Number	0
0x0004	Flash Address	0
0x0008	Data Size [bytes]	12
0x000C	Data bytes	reserviert
0x0010		Vendor-ID
0x0014		Produkt-ID
0x0018	Block CRC	CRC

Aufbau des Blocks 0 mit Angabe von Vendor-ID und Produkt-ID.

Offset	Parameter	Inhalt
0x0000	Block Number	0
0x0004	Flash Address	0
0x0008	Data Size [bytes]	24
0x000C	Data bytes	reserviert
0x0010		Vendor-ID
0x0014		Produkt-ID
0x0018		Version
0x001C		Erstellungsdatum vom Typ time_t
0x0020		
0x0024	Block CRC	CRC

Aufbau des Blocks 0 mit Angabe von Vendor-ID und Produkt-ID und Version der Applikation.

Aufbau Block 1 .. N:

<table>
<tr><th>Offset</th><th>Parameter</th><th>Inhalt</th></tr>
<tr><td>0x0000</td><td>Block Number</td><td>1 … N</td></tr>
<tr><td>0x0004</td><td>Flash Address</td><td>Zieladresse</td></tr>
<tr><td>0x0008</td><td>Data Size [bytes]</td><td>n</td></tr>
<tr><td>0x000C</td><td rowspan="2">Data bytes</td><td rowspan="2">Programmdaten</td></tr>
<tr><td>…</td></tr>
<tr><td>0x000C+n</td><td>Block CRC</td><td>CRC</td></tr>
</table>

Aufbau Block -1:

<table>
<tr><th>Offset</th><th>Parameter</th><th>Inhalt</th></tr>
<tr><td>0x0000</td><td>Block Number</td><td>-1</td></tr>
<tr><td>0x0004</td><td>Flash Address</td><td>0 … 0xFFFFFFFF</td></tr>
<tr><td>0x0008</td><td>Data Size [bytes]</td><td>8</td></tr>
<tr><td>0x000C</td><td rowspan="2">Data bytes</td><td>Größe der Applikation</td></tr>
<tr><td>0x0010</td><td>CRC der Applikation</td></tr>
<tr><td>0x0014</td><td>Block CRC</td><td>CRC</td></tr>
</table>

Die CRC wird über den gesamten Inhalt eines Blocks mit dem Polynom 0xEDB88320 berechnet. Der Startwert ist 0xFFFFFFFF. Mit demselben Algorithmus wird die CRC über die Applikation berechnet.

Besteht eine Applikation aus mehreren Teilapplikationen, so sind diese durch Angabe des Adressbereichs aus einem <input-file> oder mehreren input-files zu erzeugen. Jeder Teilapplikation ist dann eine Programmnummer zuzuordnen, die dem Subindex innerhalb des Objektes Programmdaten entspricht.

Hinweis:
Das Tool BinaryBlockConv wird nicht im Quellcode ausgeliefert. Auf Anfrage können aber weitere Optionen in das Tool integriert werden.

5.2 Das Tool BinaryBlockDownload

Dieses Tool überträgt die Binärblöcke aus dem mit dem BinaryBlockConv erzeugten Ausgabefile. Dabei überträgt dieses Tools genau eine Teilapplikation, die durch ihre Programmnummer gekennzeichnet ist. Als Schnittstelle zum CAN-Bus wird ein USB-CANmodul verwendet.

Aufruf: BinaryBlockDownload [options] input-file

Optionen	Default	Bedeutung
-P <val>	1	Dieser Parameter definiert die Programmnummer. Sie entspricht dem Subindex für die Übertragung der Kommandos und Programmdaten.
-B <index>	4	Dieser Parameter definiert über den Index die zu verwendende Bitrate. (4=125kBit/s, 3=250kBit/s)
-N <val>	1	Dieser Wert definiert die Knotenadresse des ausgewählten CANopen-Gerätes.
--delay <val>	1000	Dieser Wert definiert die Verzögerung für das Abfragen der Statusinformation nach dem Löschen oder Programmieren des Flashs.
--repeats <val>	0x000400	Dieser Wert in ms definiert die Anzahl der Wiederholungen für das Abfragen der Statusinformation nach dem Löschen oder Programmieren des Flashs.
--ser_num <val>	0	Serial Number für Konfiguration Knotennummer mit Hilfe von LSS
--rev <val>	0	Revision code für Konfiguration Knotennummer mit Hilfe von LSS
--pcode <val>	0	Product Code für Konfiguration Knotennummer mit Hilfe von LSS
--vendor <val>	0x3F	Vendor-ID für Konfiguration Knotennummer mit Hilfe von LSS
--timeout <val>	500	max. Timeout in ms für Warten auf Antwort vom Target
--help	-	Ausgabe der unterstützten Parameter und File-Formate

Tabelle 9: Parameter des Tools BinaryBlockDownload

Im BinaryBlockDownload-Tool ist eine Kommando-Shell integriert, die die Möglichkeit bietet, verschiedene Kommandos zur Steuerung des Downloads auszuführen.

Die folgenden Schritte führt das Tool BinaryBlockDownload aus:

1. Starten des Bootloaders beim Download

   Möglicherweise ist Gerät bereits eine Applikation aktiv. Beim Start des Downloads wird versucht, den Bootloader zu starten. Dieser Vorgang wird wiederholt, bis der Bootloader erkannt oder vom Benutzer abgebrochen wird.
2. Bootloader wurde gestartet

   Programm löschen und warten bis es gelöscht ist (Statusabfrage)
3. Block für Block aus BIN-File übertragen und warten, bis Block programmiert ist (Statusabfrage)
4. Nach dem letzten Block wird auf dem Target die CRC berechnet und im OD abgelegt. Die CRC wird ausgelesen und mit dem Wert auf dem PC verglichen.
5. Wenn bisher kein Fehler aufgetreten ist, dann kann die Applikation im Probelauf gestartet werden.
6. Etwas warten bis Applikation gestartet. Dann kontrollieren, ob die Applikation gestartet wurde (Device-Type lesen und vergleichen, Ident-Objekt lesen und vergleichen).

7. Dann in den Bootloader wechseln und wenn bisher kein Fehler festgestellt wurde (Device-Type ok, CRC ok, Ident-Objekt ok, Probelauf ok) die Signatur setzen und warten bis diese programmiert ist (Statusabfrage).
8. Wenn kein Fehler, dann kann die Applikation gestartet werden.

5.2.1 Fehlercodes von BinaryBlockDownload

Die Fehlercodes der Funktionen im Flashtool und der Flashtool.exe selbst (Systemvariable ERRORLEVEL) sind kompatibel zum POSIX Standard. Wenn im Fehlercode das Bit 30 gesetzt ist, dann handelt es sich um einen anwenderspezifischen Fehlercode. Andernfalls handelt es sich um einen Fehlercode aus dem Betriebssystem.

Die Fehlercodes der Flashtool Software beginnen ab dem Wert 0x60000000. Alle möglichen Fehlercodes sind in aufgelistet.

Konnte jedoch zum Beispiel das Input-File nicht geöffnet werden, dann liest das Flashtool den Fehlercode von der Variable „errno“ und gibt ihn mit einer entsprechenden Fehlermeldung auf der Konsole zurück. Eine Liste der möglichen Fehlercodes entnehmen Sie bitte aus den MSDN.

Fehlercode	Name: Bedeutung
0x00000000	Kein Fehler
0x60000001	ERR_BL_NO_VALID_PROGRAM: Der Bootloader meldete eine ungültige Applikation. Das tritt auf, wenn der Bootloader eine andere Applikations-CRC berechnet als das Flashtool.
0x60000002	ERR_BL_DATA_FORMAT_UNKNOWN: Der Bootloader meldete ein unbekanntes Datenformat im Block nach *Bild 2*.
0x60000003	ERR_BL_CRC_ERROR: Der Bootloader meldete einen CRC Fehler im Datenblock.
0x60000004	ERR_BL_FLASH_NOT_CLEARED: Der Bootloader meldete, dass der zu programmierende Flashbereich nicht gelöscht ist. Der Blank-Check schlug fehl oder das Kommando zum Löschen der Applikation wurde nicht gesendet.
0x60000005	ERR_BL_FLASH_ERROR: Der Bootloader hat einen allgemeinen (nicht näher spezifizierten) Fehler beim Schreiben der Daten in den Flash festgestellt.
0x60000006	ERR_BL_ADDRESS_ERROR: Der Bootloader hat festgestellt, dass über den Flash-Bereich der Applikation hinaus geschrieben werden soll.
0x60000007	ERR_BL_FLASH_SECURED: Der Bootloader hat festgestellt, dass in den geschützten Flash-Bereich geschrieben werden soll, der für den Bootloader reserviert ist.
0x60000008	ERR_BL_ERROR: Der Bootloader hat einen Fehler festgestellt, der dem Flashtool unbekannt ist.
0x60000010	ERR_NO_BOOTLOADER_RUNNING: Das Flashtool kann den Bootloader nicht ansprechen. Möglicherweise ist der Bootloader nicht aktiv.
0x60000011	ERR_USER_ABORT: Der Anwender hat die Übertragung abgebrochen.
0x60000012	ERR_DOWNLOAD_RUNNING: Die Funktion FtCopStartDownload() wurde aufgerufen, obwohl noch ein Download zum Bootloader aktiv ist.
0x60000013	ERR_NODE_NOT_FOUND: Es wurde ein SDO-Timeout festgestellt. Der Bootloader antwortet nicht auf ein Request.
0x60000014	ERR_START_FAILED: Das Start-Kommando konnte nicht zum Bootloader übertragen werden.

Fehlercode	Name: Bedeutung
0x60000015	ERR_PROGRAM_TOO_LARGE: Das Flashtool hat festgestellt, dass über den Flash-Bereich der Applikation hinaus geschrieben werden soll. Dabei werden die Adressen im Motorola HEX-File überprüft.

Tabelle 10: Fehlercodes von BinaryBlockDownload

5.3 Das Tool DotNetFlashtool

Das Tool DotNetFlashtool ist eine neuere Variante für den Download der Applikation(en) in das Target. Es nutzt das .NET Framwork und setzt auf die CANopen .NET API auf. Das Tool ist als Quellcode enthalten, so dass es möglich ist, vom Anwender eine grafische Oberfläche für dieses Tool zu implementieren.

Aufruf:

```
DotNetFlashtool.exe -F <filename> [options]
```

Optionen	Default	Bedeutung
-F <filename>		Dateipfad auf die Applikation als binäre Blockdatei, die mit BinaryBlockConv.exe konvertiert wurde und zum Target übertragen werden soll. Der Pfad kann relativ zu Arbeitspfad oder absolut angegeben werden.
--node_id <val>	0x01	Knotenadresse des Gerätes, auf den die Applikation geladen werden soll. Der Wertebereich liegt zwischen 1 und 126. Das DotNetFlashtool selbst verwendet die Knotenadresse 127.
--gateway_id <val>	0x00	Knotenadresse des SDO Gateways, an dem das Gerät für den Download angeschlossen ist. Der Wertebereich liegt zwischen 0 und 126. Der Wert 0 (Default-Wert) bedeutet, dass kein SDO Gateway verwendet werden soll.
--network_id <val>	0x00	Bezeichnet die Netzwerk-ID, in der sich das Gerät für den Download bei Verwendung eines SDO Gateways befindet.
--vxd_type <val>	4	Gibt an, welche PC CAN-Schnittstelle für die Übertragung der CAN-Nachrichten verwendet werden soll. Mit DotNetFlashTool --vxd_type ? wird Liste von unterstützten Schnittstellen ausgegeben. Die derzeit unterstützten Schnittstellen werden in *Tabelle 12* aufgelistet.
--device_no <val>	255	Gibt die Gerätenummer der zu verwendenden CAN-Schnittstelle an. Dieser Paramater hängt vom Parameter --vxd_type ab. Für die virtuelle CAN-Schnittstelle (VCAN --vxd_type 16) bedeutet dieser Parameter die Nummer der COM-Schnittstelle (z.B. 7 für COM7).
--br_idx <val>	0	Gibt die zu verwendende CAN-Bitrate als Index-Parameter an. Mit DotNetFlashTool --br_idx ? wird Liste von unterstützten Bitraten ausgegeben. In *Tabelle 13* werden die unterstützten CAN-Bitraten aufgelistet.
--baurate <val>	0	Gibt die zu verwendende CAN-Bitrate in kBit/s für die CAN-Schnittstelle an. Wenn die SYS TEC VCAN-Schnittstelle verwendet werden soll (VCAN --vxd_type 16), muss hier die Bitrate für die RS232-Schnittstelle in Bit/s angegeben werden. Momentan wird nur die RS232-Bitrate 115200 Bit/s unterstützt.
--chk_tgt <val>	0	Wird dieser Wert größer als 0 angegeben, dann prüft das DotNetFlashtool vor dem Download, ob die Vendor-ID und Produkt-ID des Targets mit den Werten aus der binären Block-Datei übereinstimmt. Stimmen die Werte nicht überein, dann wird der Download der Applikation mit einem Fehlercode verweigert.
--prog_no <val>	1	Gibt die Programmnummer an, an die die Applikation geladen werden soll. Sie entspricht dem Subindex des Objektes 0x1F50 im OD des Targets. Der Wert 1 adressiert immer die erste Applikation. Der Wert 0 ist nicht erlaubt.

Optionen	Default	Bedeutung
--sdo_timeout <val>	500	Gibt den SDO Timeout Wert in Millisekunden für den SDO Client im DotNetFlashtool an. Antwortet das Gerät innerhalb dieser Zeit nicht, dann bricht das Tool mit einem Fehlercode ab. Durch Änderung dieses Wertes können unter Umständen Probleme bim Download beseitigt werden.
--ip_addr <val>	0.0.0.0	Bei Verwendung des SYS TEC CAN-Ethernet Gateways (--vxd_type 14) wird hier die IP-Adresse für das Gateway angegeben.
--ip_port <val>	8234	Zusätzlich zur IP-Adresse des SYS TEC CAN-Ethernet Gateways wird mit diesem Parameter der UDP-Port angegeben.
--chk_run		Wird dieser Parameter angegeben, dann startet das DotNetFlashtool die Applikation nach dem Download aber vor dem Setzen der Gültigkeits-Signatur in einem Probelauf. Nach erfolgreichem Start versetzt das DotNetFlashtool das Gerät wieder in den Bootloader Modus und setzt nachträglich die Gültigkeits-Signatur der Applikation.

Tabelle 11: Parameter des Tools DotNetFlashtool

Achtung:
Bei Verwendung eines SDO Gateways ist darauf zu achten, dass das Netzwerk (Router-Einträge und Netzwerk-IDs im OD des SDO Gateway) bereits konfiguriert sein muss, wenn das DotNetFlashtool gestartet wird. Dieses Tool nimmt keine automatische Konfiguration des Netzwerks vor.

Die Parameter --br_idx und --baudrate sind kongurent. Werden beide Parameter in einem Aufruf verwednet, dann wird die CAN-Bitrate eingestellt, deren Position im Aufruf als letztes angegeben wurde. Die Bitrate für die virtuelle CAN-Schnittstelle kann nur mit dem Parameter --baudrate angegeben werden.

Wert für --vxd_type	Verwendete PC Schnittstelle
1	PhyCAN Treiber mit pcNetCAN-Karte für den ISA-Bus
2	PCAN 1.x Treiber mit pcNetCAN-Karte für den ISA-Bus
3	PCAN 1.x Treiber mit PCAN-Dongle von der Firma PEAK-System Technik GmbH
4	USBCAN 4.x Treiber mit dem SYS TEC USB-CANmodul
5	PCAN-PCI Treiber mit der PCI-Karte von der Firma PEAK-System Technik GmbH
6	Reserviert
7	PCAN 2.x Treiber mit pcNetCAN-Karte für den ISA-Bus
8	PCAN 2.x Treiber mit PCAN-Dongle von der Firma PEAK-System Technik GmbH
9	PCAN 2.x Treiber mit der PCI-Karte von der Firma PEAK-System Technik GmbH
10	PCAN-Dongle Treiber mit PCAN-Dongle von der Firma PEAK-System Technik GmbH
11	PCAN-USB Treiber mit dem PCAN-USB Adapters der Firma PEAK-System Technik GmbH
12	Reserviert
13	Reserviert
14	Treiber für das SYS TEC CAN-Ethernet Gateway
15	PCAN 2.x Treiber mit PCAN-USB Adapter von der Firma PEAK-System Technik GmbH
16	Virtuelle CAN Schnittstelle über die serielle COM-Schnittstelle (RS232, UART, VCAN).

Tabelle 12: CAN-Schnittstellen für das DotNetFlashtool

Wert für --br_idx	Verwendete CAN-Bitrate
0	1000 kBit/s (bzw. 1 MBit/s)
1	800 kBit/s
2	500 kBit/s
3	250 kBit/s
4	125 kBit/s
5	100 kBit/s
6	50 kBit/s
7	20 kBit/s
8	10 kBit/s

Tabelle 13: CAN-Bitraten für das DotNetFlashtool

Beispiele:

- Übertragen der Datei blcop_app.bin als erste Applikation an das Gerät mit der Node-ID 0x01 unter Verwendung des USB-CANmoduls, welches als erstes am PC gefunden wird, und CAN-Bitrate 1000 kBit/s:

```
DotNetFlashtool –F blcop_app.bin
```

- Übertragen der Datei blcop_app.bin als erste Applikation an das Gerät mit der Node-ID 0x62 unter Verwendung des USB-CANmoduls mit der Gerätenummer 5 und CAN-Bitrate 500 kBit/s:

```
DotNetFlashtool –F blcop_app.bin --node_id 0x62 --baudrate 500
  --device_no 5
```

- Übertragen der Datei blcop_app2.bin als zweite Applikation an das Gerät mit der Node-ID 15 unter Verwendung des USB-CANmoduls, welches als erstes am PC gefunden wird, und CAN-Bitrate 125 kBit/s:

```
DotNetFlashtool –F blcop_app2.bin --node_id 15 --br_idx 4
  --device_no 255 --prog_no 2
```

- Übertragen der Datei blcop_app.bin als erste Applikation an das Gerät mit der Node-ID 15 unter Verwendung der VCAN-Schnittstelle COM7 und RS232-Bitrate 115200 Bit/s:

```
DotNetFlashtool –F blcop_app.bin --node_id 0xF --vxd_type 16
  --baudrate 115200 --device_no 7
```

- Übertragen der Datei blcop_app.bin als erste Applikation an das Gerät mit der Node-ID 20 unter Verwendung des CAN-Ethernet Gateways mit der IP-Adresse 192.168.1.10 und UDP-Port 1234:

```
DotNetFlashtool –F blcop_app.bin --node_id 20 --vxd_type 14 --ip_addr
192.168.1.10
  --ip_port 1234
```

Der Quellcode des Tools wird mit dem CANopen Bootloader Add-on ausgeliefert. Die Solutions für Microsoft Visual Studio sind unter folgendem Pfad zu finden:

```
c:\systec\cop\target\x86\windows\canopendotnet\examples\vc9\dotnetflashtool\
```

oder

```
c:\systec\cop\target\x86\windows\canopendotnet\examples\vc10\dotnetflashtool\
```

6 Ressourcen

6.1 Code Daten Target

Die tatsächlichen Werte sind abhängig vom Compiler, dem unterstützten Speichermodell, dem Optimierungslevel und von der CPU. Für die Ausführung des Bootloaders sollten auf einem 16Bit-System 32kByte Flash und ca. 6kByte RAM zur Verfügung stehen.
Falls die vorhandenen Ressourcen nicht ausreichen, den Bootloader zu integrieren, so können die Softwaremodule im Rahmen eines Adaptation-Workshops angepasst werden.

6.2 Interrupts Target

Der Bootloader basiert auf den Standard-CAN-Treiber für CANopen. Das Empfangen von Nachrichten, das Senden von Nachrichten sowie Wechsel des CAN-Controller-Status werden mit Hilfe von Interrupts signalisiert. Kann, bedingt durch weitere Aufgaben des Mikrocontrollers während des Update-Vorgangs, die Verwendung von Interrupts nicht unterstützt werden, so sind die entsprechenden CAN-Treiber-Routinen durch den Anwender anzupassen. Eine Anpassung kann auch im Rahmen des Adaptation Workshops in Zusammenarbeit mit SYS TEC erfolgen.

Weiterhin wird ein System-Timer benötigt, der in der Regel auch Interrupt-gesteuert realisiert ist.

Da sich der Bootloader und die mit Hilfe des Bootloaders übertragene Applikation die Interrupts für den CAN-Controller teilen, sind entsprechende Anpassungen in Form von Interrupt-Weiterleitung oder Interrupt-Vektortabellen im RAM vorzunehmen. Abhängig von der verwendeten CPU muss unter Umständen die Interrupt-Vektortabelle gespiegelt werden. Eine andere Variante wäre diese Tabelle im RAM aufzubauen und jeder Treiber trägt dort den Vektor der Interrupt-Service-Routine ein. So kann der Bootloaders bei Verwendung von Interrupts für Timer und CAN-Controller die Vektoren modifizieren. Beim Start der Applikation werden dann die relevanten Einträge durch die Vektoren der Applikation ersetzt. In Vorbereitung auf eine Integration des Bootloaders ist die jeweilige Vorgehensweise abzustimmen und vom Projektpartner ein Beispielprogramm zu implementieren, das die Umsetzung demonstriert.

Die beste Variante ist, wenn der Bootloader komplett ohne Interrupts verwendet wird. Dafür muss die CPU eine Timer Peripherie besitzen, deren Timer Value Register so einstellbar ist, dass die Zeitbasis von 100µs direkt abgelesen (unter Umständen mit einer Umrechnung) werden kann. Für den CAN-Controller besteht die Möglichkeit, dass dieser im Polling abgefragt wird. Dafür ist die Funktion TgtProcessEvents() zu implementieren. Sie muss dabei die Funktion CdrvInterruptHandler() aufrufen. Der Aufruf von TgtProcessEvents() erfolgt durch den Bootloader selbst.

Indexverzeichnis

A

AppCbProgramCtrl 41

B

Betriebssystem 17, 19
BinaryBlockConv 52
BinaryBlockDownload 57
Bitrate 17, 19, 25, 44
BLCOP_APP_START_INDEX 46
BLCOP_APPLICATION_END_AREA1 48
BLCOP_APPLICATION_START_AREA1 48
BLCOP_BASE_REQUEST 45
BLCOP_BASE_RESPONBSE 45
BLCOP_BDI_TABLE_PTR 45
BLCOP_BDI_TABLE_SIZE 45
BLCOP_BOOTLOADER_END_AREA 48
BLCOP_BOOTLOADER_START_AREA 48
BLCOP_DECRYPT_SUBTRAHEND 47
BLCOP_IDENTITY_CHECK 38, 40, 46
BLCOP_IDENTITY_PRODUCTID 45
BLCOP_IDENTITY_REVISION 45
BLCOP_IDENTITY_SERIALNR 37, 45
BLCOP_IDENTITY_VENDORID 45
BLCOP_MAX_BAUIDX 25, 44
BLCOP_MAX_CANLOOPS 45
BLCOP_MAX_CRC_STEP_SIZE 21, 46
BLCOP_MAX_FLSWRITE_STEP_SIZE 21, 46
BLCOP_MAX_NODEID 25, 44
BLCOP_MAX_PROGRAM_BUFFER 44
BLCOP_MAX_PROGRAMS 44
BLCOP_MIN_BAUIDX 25, 44
BLCOP_MIN_NODEID 25, 44
BLCOP_PROCFLAG_ERROR 38
BLCOP_PROCFLAG_PROCECCED 38
BLCOP_SEND_BOOTUP 45
BLCOP_STATERRMAN_WRONG_PID 38
BLCOP_STATERRMAN_WRONG_VID 38
BLCOP_USE_AMR_ACR_FILTER 46
BLCOP_USE_CANCRTL 45
BLCOP_USE_CANINTENABLE 45
BLCOP_USE_DECRYPT 47
BLCOP_USE_WATCHDOG 47
BlCopInitialize 25
BlCopProcess 26
BlCopShutDown 26
Blocknummer 10
Bootloader 14
Bootloader-API 24
Bootup-Nachricht 45

C

Callgate 17, 20
CAN-Ethernet Gateway 62
CAN-Treiber 43
CRC 10, 33, 36, 46, 50
CRC32_RAM_TABLE 50

D

Debug-Ausgaben 43
DotNetFlashtool 60

E

Emergency 22
ERR_BL_ADDRESS_ERROR 58
ERR_BL_CRC_ERROR 58
ERR_BL_DATA_FORMAT_UNKNOWN 58
ERR_BL_ERROR 58
ERR_BL_FLASH_ERROR 58
ERR_BL_FLASH_NOT_CLEARED 58
ERR_BL_FLASH_SECURED 58
ERR_BL_NO_VALID_PROGRAM 58
ERR_DOWNLOAD_RUNNING 58
ERR_NO_BOOTLOADER_RUNNING 58
ERR_NODE_NOT_FOUND 58
ERR_PROGRAM_TOO_LARGE 59
ERR_START_FAILED 58
ERR_USER_ABORT 58

F

Fehlercodes 26, 31
Fehlerstatus 7
Filterung 46
Flash-API 27
Flashtools 52
Flash-Treiber 27
FLSDRV_ADDR 27
FLSDRV_SIZE 27
FlsDrvEraseInitialize 28
FlsDrvEraseSector 28
FlsDrvInitialize 27
FlsDrvReadData 30
FlsDrvWriteData 29
FlsDrvWriteInitialize 29

G

Gerätetyp 7

H

Host 52

I

Intel-Format 10
Interrupt 19, 32, 33

K

kBlCopError 26
kBlCopForceStartApp 26
kBlCopInitError 26
kBlCopOk 26
Knotenadresse 19, 34, 44
Knotennummer 17, 33
Kommandokanal 7
Konfiguration 44

L

Little-Endian 10
LSS 17, 19, 25, 33, 34

O

Objekt 8
Optimierungen 49
Overrun 22

P

PLACE_IN_RAM 27
Produkt-ID 7, 37, 38, 45
Programmdaten 7
Prozessfunktion 26

R

Reset-Vektor 14
Revision...45
Revisionsnummer 7
Rücksprung 21, 25, 41

S

SDO...22
SDO_BLOCKSIZE_DOWNLOAD 22
SDO_BLOCKSIZE_UPLOAD.....................22
SDO_BLOCKTRANSFER 22
SDO-Blocktransfer 17, 50
Seriennummer 7, 17, 25, 37, 45
Signatur 21, 33, 36

T

tBlCopFlashInfo 34
TgtBlCopInitialize 33
TgtBlCopInitVars 33
TgtCheckAppSig 36
TgtCheckEraseApp 40
TgtCheckReEarseApp...............................40
TgtClrAppSig 37
TgtGetAppCrc 36
TgtGetAppSize 35
TgtGetFlashInfo...34
TgtGetNodeId 34
TgtGetSerialNr 37
TgtGetTickCount 32
TgtInitSerial 43
TgtInitTimer 32
TgtProcessAppInfoData 38
TgtProcessEvents 33
TgtSetAppCrc 36
TgtSetAppSig 37
TgtSetAppSize 35
TgtSetNodeId 34
TgtStartBootloader 42
TgtStopTimer 32
Timer 19, 32, 49

U

USB-CANmodul 62

V

VCAN-Schnittstelle 62
Vendor-ID 7, 37, 38, 45
Verschlüsselung 23, 53, 54
Vorüberlegungen 17

W

Watchdog 17, 21, 33, 47

Dokument:	Software Manual CANopen Bootloader
Dokumentnummer:	L-1112d_10, Auflage 10

Wie würden Sie dieses Handbuch verbessern?

Haben Sie in diesem Handbuch Fehler entdeckt? Seite

Eingesandt von:

Kundennummer:

Name:

Firma:

Adresse:

Einsenden an:

SYS TEC electronic GmbH
Am Windrad 2
D-08468 Heinsdorfergrund
GERMANY
Fax : +49 (0) 3765 / 38600-4100

Veröffentlicht von

SYS TEC ELECTRONIC

Printed in Germany
Best.-Nr. L-1112d_10

1 © SYS TEC electronic GmbH 2015