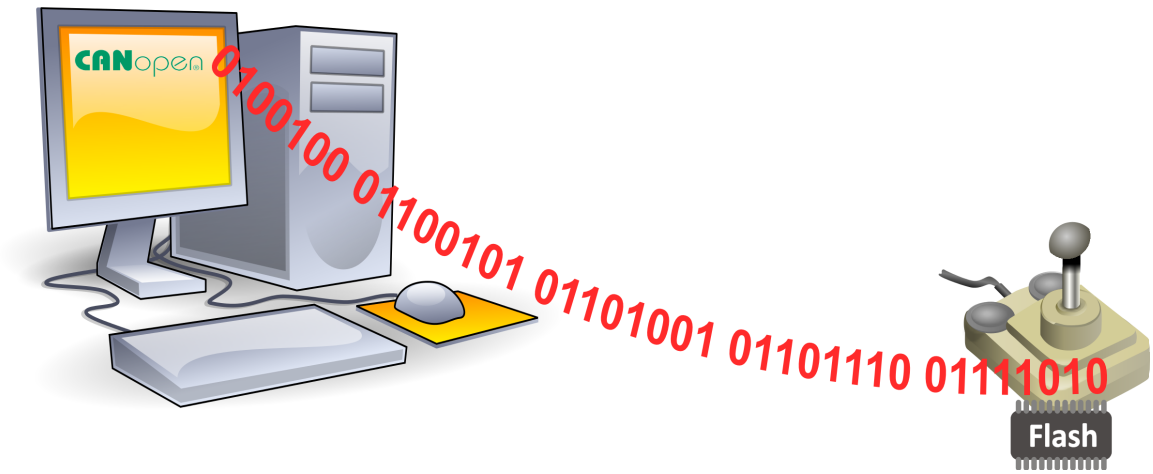
Paulus

CANopen Mini Bootloader

Benutzerhandbuch



Ablehnungshinweis
Alle Rechte vorbehalten

Die von *port* GmbH gelieferten Programme, Baugruppen und Dokumentationen werden mit großer Sorgfalt erstellt und in unterschiedlichen Einsatzfällen getestet und geprüft.

port GmbH kann trotzdem keine Gewähr oder Haftung dafür übernehmen, dass die Software, die Baugruppe und die Dokumentation fehlerfrei bzw. für spezielle Zwecke geeignet ist.

Insbesondere Beschreibungen und technische Daten sind keine zugesicherten Eigenschaften im rechtlichen Sinne.

Für Folgeschäden, die aufgrund der Benutzung der Programme und Baugruppen auftreten, wird deshalb jede juristische Verantwortung oder Haftung ausgeschlossen.

port hat das Recht, Änderungen an den beschriebenen Produkten oder an der Dokumentation ohne vorherige Ankündigung vorzunehmen, wenn sie aus Gründen der Zuverlässigkeit oder Qualitätssicherung vorgenommen werden oder dem technischen Fortschritt dienen.

Für Hinweise auf eventuelle Fehler sind wir dankbar und bitten um eine Benachrichtigung.

Wir werden uns bemühen, derartige Hinweise schnellstmöglich zu überprüfen.

Copyright

Regensburger Straße 7b
D-06132 Halle
Tel. +49 345 - 777 55 0
Fax. +49 345 - 777 55 20
E-Mail service@port.de
Internet http://www.port.de

Inhaltsverzeichnis

1. Allgemeines

Ein modernes Gerätedesign erfordert eine enorme Flexibilität der Hard- und Software. Diese Flexibilität wird durch die Integration von Download-Mechanismen und Programmierfunktionen in die Software und eine zukunftsorientierte Dimensionierung der Hardware erreicht.
Bootloader mit einem Kommunikationsinterface ermöglichen das Einspielen von Softwareerweiterungen über ein entsprechendes Netzwerk. Standardisierte Kommunikations-Objekte und Algorithmen gewährleisten eine hohe Transparenz und Bedienfreundlichkeit.

Der CANopen Bootloader bietet diese Flexibilität für Geräte in CANopen-Netzwerken. CANopen stellt mit dem SDO Transfer standardisierte Mechanismen für die Übertragung größerer Datenmengen bereit. Der Bootloader selbst arbeitet unabhängig von der Applikation als minimaler CANopen Slave Knoten nach CiA-301.

Ein Software-Update kann mit einem CANopen Master oder Konfigurationstool über den Anwenderbereich des Code-FLASH-Speichers durchgeführt werden.

Paulus ist ein auf Codegröße optimierter Bootloader, der weitgehend kompatibel zu CANopen ist. Um eine minimale Kodegröße zu erreichen, wird auf den Einsatz eines flexiblen CANopen Protokoll Stack verzichhtet. Die erfoderliche CANopen Funktionalität ist hart kodiert. Das Projekt ist jedoch so flexibel aufgebaut, das es einen hardwareunabhängigen Kodeteil für die Protokollabwicklung gibt, und jeweils einen zielprozessorspezifischen Teil mit Anfangsinitalisierung und speziell an den Prozessor angepassten FLASH routinen.

Es sind Implementierungen für die folgenden Prozessoren verfügbar:

- dsPIC33F von Microchip
- DSP Controller von Texas Instruments TMS320F2812/2808/28335
  (in Entwicklung)
- 32-bit ARM Controller STM32 von ST Microelectronics

Auf Grund der großen Nachfrage nach diesem Bootloader kann das Handbuch nicht immer den aktuellen Stand der Verfügbarkeit für bestimmte Prozessoren widerspiegeln. Fragen Sie deshalb bei unserem Vertrieb nach aktuellen Versionen. Gerne übernehmen wir auch in Ihrem Auftrag die Anpassung an noch nicht unterstützte Prozessoren.

Der Code des Bootloaders ist sehr allgemeingültig gehalten und modular geschrieben. Damit lässt er sich sehr leicht auf andere Architekturen übertragen. Hinzugefügt werden müssen dann:

- Die jeweilige Grundinitialisierung des Prozessors
- Das FLASH Handling zur Programmierung
- Die Vorschrift zum Linken von ladbaren Applikationen.

2. CANopen Eigenschaften

Einige der im Folgenden beschriebenen Eigenschaften sind immer vorhanden, andere können durch eine Konfigurationsdatei durch das Setzen entsprechnder `#define` in der Datei <*target*>*/bl_config.h* aktiviert werden.

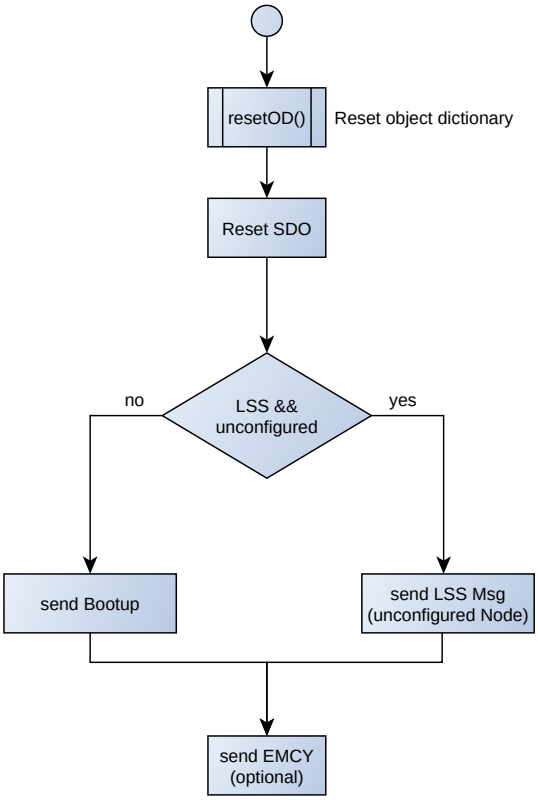
2.1. NMT

Eine NMT-Statemachine ist für den CANopen Bootloader wenig sinnvoll. Der CANopen Bootloader bleibt dauerhaft im Zustand Pre-Operational, welcher direkt nach dem Booten eingenommen wird. Zur Datenkommunikation wird nur SDO Transfer benutzt.

NMT Kommando	**Aktion**
Reset Applikation	Reset
Reset Communication	Bootup senden
Start	ignoriert
Preoperational	dauerhaft
Stopped	ignoriert

Das NMT Kommando *Reset Application* wertet der Bootloader jedoch für einen SW Reset aus. Z.B. wenn empfangen nach einem erfolgreichen Firmware-Download. Das NMT Kommando *Reset Communication* führt bei aktivem LSS zum Aktivieren der neuen Node Id. Außerdem wird die SDO Kommunikation zurückgesetzt.

2.1.1. Implementation Reset Communication



2.2. NMT Error

Eine Heartbeat Erzeugung durch den Bootloader ist vorgesehen. Je nach Target wurden bisher unterschiedliche Varianten implementiert.

- Zählschleife in der main-loop, mit relativer Ungenauigkeit.
- Hardware Timer Nutzung, mit größerem Resourcenaufwand.

Auf die Heartbeat Erzeugung kann jedoch auch gänzlich verzichtet werden.

Wenn es die Kodegröße erlaubt, sendet der Bootloader eine Boot-up Message.

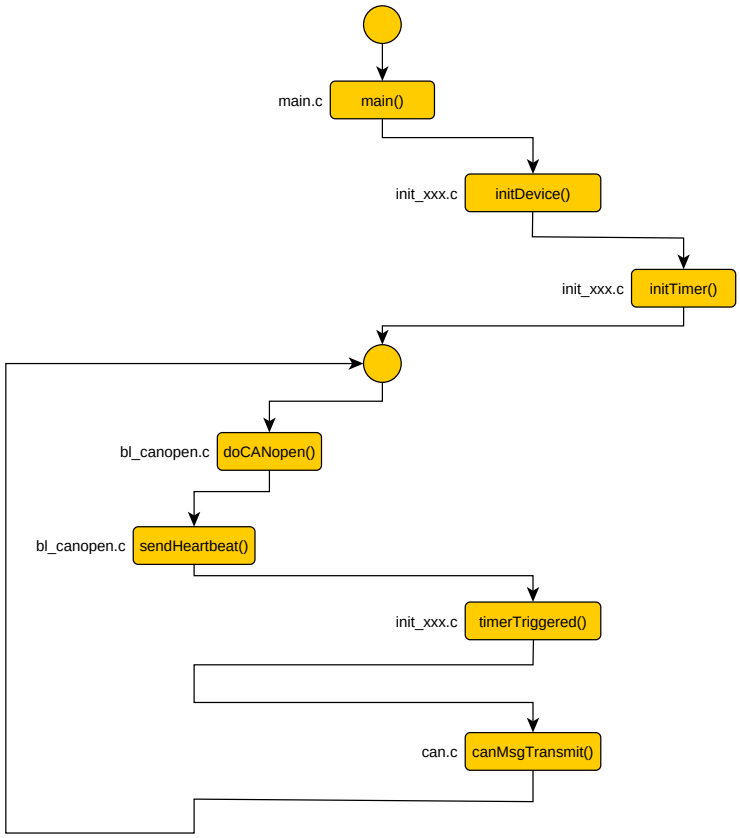
Dienst	**Aktion**
Bootup Message	wird unterstützt
Heartbeat	feste HB Zeit von 0
Heartbeat	optional - feste HB Zeit größer 0

Der Objektverzeichniseintrag 0x1017 für den Heartbeat Producer ist verfügbar. Für die optionale Funktionalität, ein zyklisches Heartbeat zu senden, wird the Funktion *timerTriggered()* benötigt.

2.2.1. Implementation

Soll zyklisch ein Heartbeat gesendet werden, muss die Funktionalität per Define aktiviert werden.

```
#define BL_USE_HB 1
```



2.3. Emergency

Wenn benötigt, können Emergency mit Einschränkungen genutzt werden. Aufgrund der Einschränkung reduziert sich die Emergency-Funktionalität auf das reine Senden einer CAN Nachricht.

- Die COB-ID der Emergency ist fest (predefined Connection Set).
- Die Inhibittime wird nicht unterstützt.
- 0x1001 wird nicht angepasst.
- 0x1003 wird nicht unterstützt.

Die EMCY Funktionalität muß in der Bootloader Konfiguration aktiviert werden.

```
#define BL_USE_EMCY 1
```

Zusätzlich zur Bootup Nachricht, kann beim Start des Bootloaders eine Emergency gesendet werden. Dies vereinfacht die Unterscheidung zwischen dem Start des Bootloaders und dem Start des Applikation.

```
#define BL_TXEMCY_AFTER_BOOTUP 1
```

Es wird die erste Emergency gesendet, welche im `emcyErrMsg[]` Array in *bl_canopen.c* definiert ist (`EMCY_0`).

Analog können auch eigene Emergencys generiert werden.

```
canMsgTransmit(EMCY_COBID, &emcyErrMsg[EMCY_0]);
```

2.4. CANopen Layer Setting Services, LSS

CANopen LSS wird als Slave unterstützt. Paulus kann durch einen LSS Master eine CANopen Node-ID vergeben werden. Bei Verwendung eines Shared Memory Bereiches kann diese Information an eine duch den Bootloader gestartete CANopen Anwendung weitergegeben werden.

Es wurden auch schon Lösungen für eine Node-Id Vergabe mit Hardware Daisy-Chain unter Nutzung vereinfachter LSS Dienste implementiert.

```
#define BL_USE_LSS 1
```

2.5. SDO

Der Paulus Bootloader ist SDO Server.

Der expedited und der segmentierte SDO Transfer werden zum Zugriff auf das Objektverzeichnis und Firmwaredownload unterstützt. Die Anzahl der Daten muss angegeben werden. Die Telegramme des segmentierten SDO Transfers sind auf 7 gültige Bytes festgelegt.
Grund für die Festlegungen, welche in den meisten Fällen angewendet werden, ist die feste Kodierung des SDO Kommandobytes. Eine detaillierte Entschlüsselung erfolgt nicht.

Der mögliche Abort-Error-Code ist auf wenige Error Codes beschränkt. Häufig wird der Error Code *Generic Error*, Error Code 0x0800 0000, genutzt.

2.6. PDO

Der PDO Dienst wird nicht unterstützt.

2.7. Objektverzeichnis

Die folgende Tabelle gibt einen überblick über die im Paulus implementierten Objekte. Optionale Einträge sind gekennzeichnet. Als Referenz verweisen wir auch auf die EDS Datei und deren Dokumentation im HTML Format[1]. Die EDS Datei liegt sowohl im traditionellen Format als *paulus.eds* als auch im XML Format nach CiA 311 als *paulus.xdd* vor. Beide wurden mit dem CANopen Design Tool[2] erzeugt.

Index	Subindex	Mode	Anmerkung
0x1000	0	co	Device Type
0x1001	0	co	(*) keine Fehlermeldungen unterstützt
0x1014	0	co	(*) feste Emcy COB-ID
0x1017	0	co	(*) 0 oder festgelegte Zeit
0x1018	0-2	co	(*) Identify objekt
0x1018	3-4	co	(*) mit LSS
0x1F50	0	co	(*) Number of Elements
0x1F50	1	wo	Domain Entry - neue Firmware
0x1F51	0-1	rw	(*) Program Control
0x1F56	0-1	ro	(*) Application software identification
0x1F57	0-1	ro	(*) Flash status identification

(*) auf diese Objekte kann zu Gunsten der Codegroße verzichtet werden.

Program Control — 0x1F51

Schreiben einer 0x01 in diesen Index, führt einen Start der geladenen Applikation aus. Schreiben einer 0x03 in diesen Index, bewirkt eine Löschung des Anwender-FLASH-Speichers.

Achtung: Das Starten der Applikation über diesen Eintrag ist möglich, aber ungünstig, da bereits ein Teil der Peripherie durch Paulus initialisiert wurde. Der Applikationsstart sollte über einen Power on Reset durchgeführt werden. In diesem Fall führt Paulus nur die CRC Prüfung der Anwendung durch und lässt z.B den CAN unberührt.

Application software identification — 0x1F56

Dieses Objekt wird je nach Implementierung nur sehr eingeschränkt unterstützt. In diesem Objekt kann man die CRC Summe der geflashten Applikation auslesen und damit identifizieren.

Flash status identification — 0x1F57

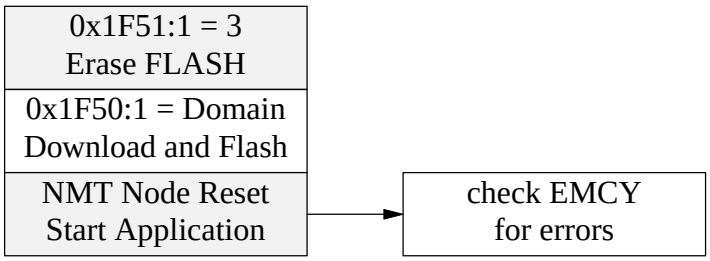
Dieses Objekt wird je nach Implementierung nur sehr eingeschränkt unterstützt.

[1] Sind als Anlagen verfügbar

[2] http://www.port.de/0640

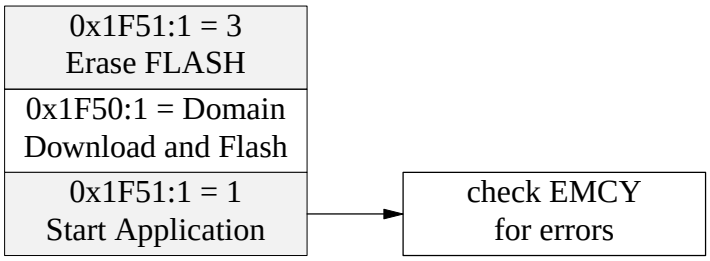
2.7.1. Programmierablauf

Das Aufspielen einer neuen Anwender Software sollte in folgenden Schritten erfolgen:



Während der Entwicklung kann die Applikation auch per SDO angesprungen werden. Dies erleichtert häufig das Debuggen.

```
#define BL_CALL_BY_SDO 1
```



2.8. CANopen Knotennummer und Bitrate

Wenn der LSS Dienst aufgrund des Resourcenverbrauchs, nicht implementiert wird, müssen einfachere Methoden, wie das Auslesen von Schaltern oder EEPROM Zellen, gewählt werden. In diesem Fall müssen die Funktionen *getNodeId*() und *getBitRate*() , welche in *<target>/<target>_init.c* zur Verfügung gestellt werden, die Informationen an die CANopen Schicht liefern.

Der LSS Dienst kann in der Bootloader Konfiguration[3] zugeschaltet werden.

```
#define BL_USE_LSS 1
```

2.9. Daten Austausch zwischen Paulus und der Anwendung

Die Schnittstelle zwischen der Paulus Protokoll Schicht und der Hardware-Anpassung hierfür befindet sich in *bl_interface.[ch]*.

Anwendungen sollen die in *bl_interface.h* beschriebenen Macros zum gezielten Rücksprung in den Bootloader verwenden. Nach dem Paulus Konzept fordert eine Anwendung

[3] Zielplattform spezifische Möglichkeiten und Hinweise sind immer der Datei *<target>/bl_config.h* in der Sourcekode-Auslieferung zu entnehmen.

ein Update an und springt dazu in den Bootloader mit folgender Anweisung:

```
BOOTLOADER_JUMP(APPL);
```

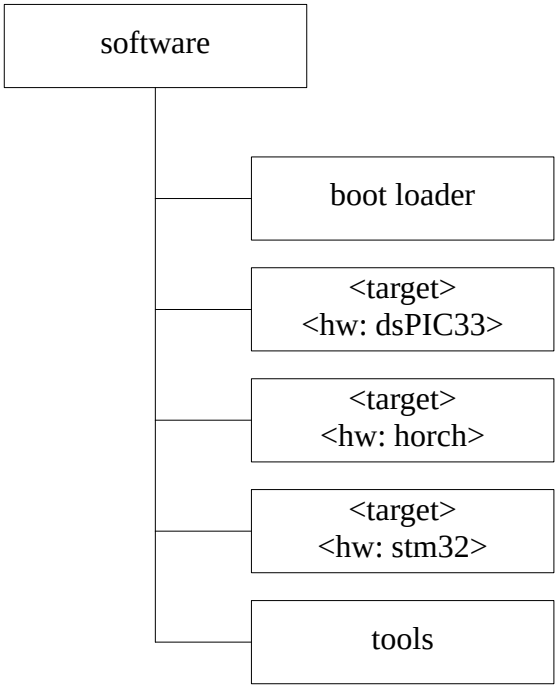
Mit der Anweisung

```
BOOTLOADER_JUMP(BL);
```

wird der Bootloader aufgerufen, führt eine erneute Prüfung des Anwenderimages aus und startet bei erfolgreicher Prüfung die Anwendung erneut. Im Falle eines CRC Fehlers verbleibt Paulus jedoch im Bootloader Mode und erwartet ein Update über 0x1F50.

3. Verzeichnisstruktur

Hardware-abhängiger und Hardware-unabhängiger Source werden in verschiedenen Verzeichnissen verwaltet. Dies vereinfacht die Verwaltung der Sourcen im CVS.



Das Verzeichnis *bootloader* enthält den hardwareunabhängigen Softwareteil. Der hardwareabhängige Softwareteil wird nach der jeweiligen Hardware benannt, z.B. *horch*. Die Konfiguration ist im *horch*-Verzeichnis abgelegt. Die Projekte und *main.c* werden unter *software* abgelegt.

Im Tools-Verzeichnis ist die Software enthalten, welche zum Erstellen von Images für den Bootloaders benötigt werden. Mehr dazu im jeweiligen Kapitel der Hardwareplattform.

4. Hardware Anforderungen

Durch die Trennung zwischen Protokoll-Schicht und HAL[4] ist prinzipiell der Einsatz auf allen Zielplattformen möglich. Nur die FLASH-Routinen sind an die verwendete Hardware anzupassen.

Der Resourcenverbrauch schwankt je nach CPU und verwendeten Kompiler und Kompilereinstellungen. Typische Werte sind 4-8KiB Flash und 2KiB RAM.
Als Client-Gegenstelle kann jede PC Interface Hardware verwendet werden
- z.B. ein USB-CAN Interface (CPC-USB oder USB-XS) oder ein Gateway gemäß CiA DSP309-3 (EtherCAN).

5. Software Anforderungen

Die Client-Gegenstelle ist mit einer CANopen-fähigen Software zu betreiben, die ein Domain-Download erlaubt. Ein Download-Programm ist kostenfrei verfügbar unter der Adresse http://www.port.de
Bestens geeignet sind Standard CANopen Konfigurationstools wie zum Beispiel der **CANopen Device Monitor**[5].

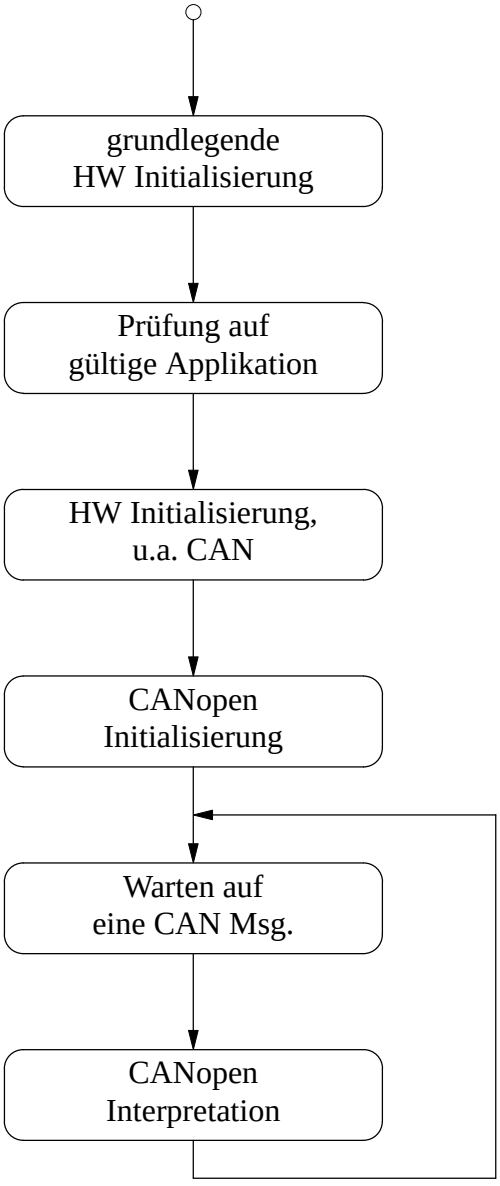
[4] HAL — Hardware Abstraction Layer

[5] http://www.port.de/0642

6. Implementierung

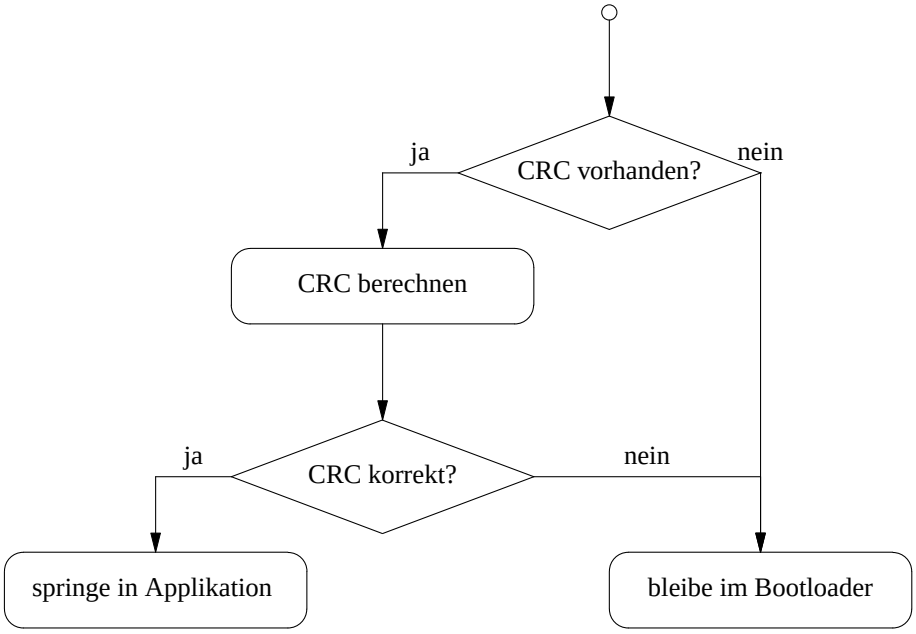
6.1. Abläufe

6.1.1. Hauptschleife



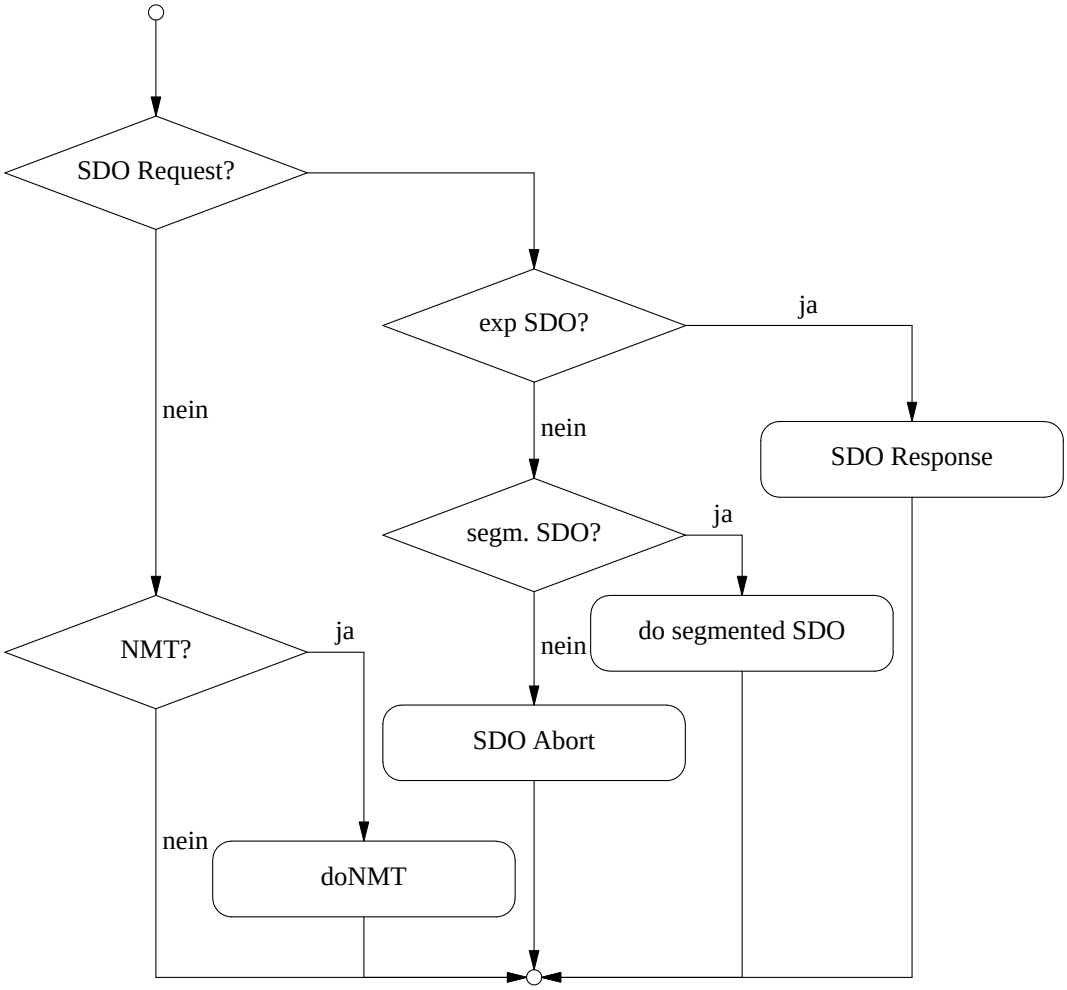
Ablauf in *main.c*

6.1.2. Test auf gültige Applikation



callApplication(), <hw>_appl.c

6.1.3. Interpretation von CANopen Requests



doCANopen(), bl_canopen.c

Weitere Dienste analog.

6.2. Verwendete Strukturen

6.2.1. CanMsgRx_T

```
typedef struct
{
  UNSIGNED16 StdId;             /**< identifier */
  UNSIGNED8 DLC;                /**< message length */
  union {
      UNSIGNED32 u32Data[2];    /**< data as 2x32bit values */
      UNSIGNED16 u16Data[4];    /**< data as 4x16bit values */
      UNSIGNED8 u8Data[8];      /**< data as 8x8bit values */
  } Msg;
} CanMsgRx_T;
```

low-level Receive message

CanMsgRx_T kommt von CAN Treiber. Der Inhalt ist leer/ungültig, wenn StdId == 0xFFFF ist.

6.2.2. CanMsgTx_T

```
typedef struct
{
  UNSIGNED8 DLC;                /**< message length */
  union {
      UNSIGNED32 u32Data[2];    /**< data as 2x32bit values */
      UNSIGNED16 u16Data[4];    /**< data as 4x16bit values */
      UNSIGNED8 u8Data[8];      /**< data as 8x8bit values */
  } Msg;
} CanMsgTx_T;
```

low-level Transmit message

Für DSPs müssen die UNSIGNED8 Einträge mit Vorsicht genutzt werden, da diese intern 16bit breit sind!

CanMsgTx_T wird zum Definieren der Sende-/Antwortnachrichten genutzt. Diese Definitionen können im Flash abgelegt werden. Daher enthalten diese keine CAN-ID. Diese ist in vielen Fällen von der Node-ID abhängig.

Beispiel

```
static const CanMsgTx_T bootupMsg = {
    1, {.u32Data[0] = 0ul, 0ul}
};
```

6.2.3. SdoRequest_T

```
typedef struct
{
    UNSIGNED32 request;   /**< first 4 Bytes of the SDO Request */
                          /**< contains command, Index and Subindex */
                          /**< of the Request */
    CanMsgTx_T response;  /**< complete SDO Response */

} SdoRequest_T;
```

Diese Struktur enhält praktisch das Objektverzeichnis. Es werden hier hauptsächlich die konstanten Einträge, welche per exp. SDO Transfer übertragen werden, definiert.

Aktuell stehen die Werte im Ram und können während der Initialisierung geändert werden. Somit ist es möglich, z.B. die Bootloaderversion im Objektverzeichnis zu hinterlegen. Genauso ist auch im späteren Verlauf möglich, Softwarezustände im Objektverzeichnis abzulegen.

Beispiel:

```
#define SDO_REQ_COBID (0x600 + nodeId)
#define SDO_RESP_COBID (0x580 + nodeId)

static SdoRequest_T sdoRequest[] = {
#define SDO_1000_0_IDX 0
    {0x00100040ul /* 0 - sdo 1000:0 */,
                    {8,{.u32Data[0]=0x00100043ul,0x0000FFFEul}} },
 ...
}
```

6.3. segmentierter SDO Transfer

SDO Upload wird nicht unterstützt!

Initiate SDO Download

	CMD	Index,SubIndex	Länge
Request	0x21	0x1F50 0x01	0x00004610
Response	0x60	0x1F50 0x01	0x00000000

SDO Download

	CMD	Daten
Request	0x00	7 Byte
Response	0x20	reserved
Request	0x00+0x10<togglebit>	7 Byte
Response	0x20+0x10<togglebit>	reserved
...		
Request	0x00+<togglebit>+<7-Länge>	0..7 Byte
Response	0x20+<togglebit>+<7-Länge>	reserved

7. Implementierungsdetails und Anforderungen an die Applikation

Mit dem Paulus CANopen Bootloader können derzeit nur Applikationen im Binärformat verarbeitet werden. Die Verarbeitung von Software im Intel-Hex-Format ist angedacht. Sie kann bei Bedarf nachgerüstet werden.

Der Bootloader führt nach der Datenübertragung eine Prüfung der empfangenen Daten durch. Dazu ist dem Applikations-Programm ein 128 Byte langer Applikations-Header voranzustellen,[6] der eine CRC Prüfsumme enthält. Dieser Header kann mit dem Tool **paulus_cksum** generiert werden und enthält folgende Informationen:

```
struct {
  UNSIGNED32  length;           /* application length       */
  UNSIGNED16  crc;              /* application crc          */
  UNSIGNED16  applicationType;  /* reserved                 */
  void (* entry_point)(void);   /* Application Entry point  */
} APPLICATION_HEADER_T;
```

Im Applikations-Header werden ungenutzte Bytes standardmäßig mit 0x00 belegt. Target spezifische Abweichungen sind möglich.

[6] teilweise aber auch plattformspezifisch, siehe folgende Unterkapitel

8. Portierung zum STM32

Die Portierung wurde mit der gcc-basierenden Entwicklungsumgebung CrossWorks für ARM Version 2 ausgeführt. In der Wurzel des Projektverzeichnisses liegt die Projekt-Datei *paulus_cw_stm32.hzp*. Das folgende Bild zeigt die Verzeichnisstruktur:

```
|-- Readme
|-- THUMB Debug
|-- THUMB Release
|-- bootloader
|   |-- bl_can.h
|   |-- bl_canopen.c
|   |-- bl_canopen.h
|   |-- bl_crc.c
|   |-- bl_hw.h
|   |-- bl_type.h
|   \-- bl_user.c
|-- eds
|   |-- paulus.can
|   |-- paulus.eds
|   |-- paulus.html
|   |-- paulus.xdd
|   \-- style.css
|-- flash_placement.xml
|-- hello
|   |-- flash_placement.xml
|   |-- init.c
|   |-- main.c
|   |-- main.c.bak
|   \-- stm32f10x_conf.h
|-- main.c
|-- paulus_cw_stm32.hzp
|-- paulus_cw_stm32.hzs
|-- stm32     $(TargetDir)
|   |-- STM32F10x_Startup.s
|   |-- STM32_Startup.s
|   |-- STM32_Target.js
|   |-- bl_config.h
|   |-- cw_settings
|   |-- environ.h
|   |-- fwlib3.3.0
|   |   |-- inc
|   |   \-- src
|   |-- linkeropts
|   |-- stm32_appl.c
|   |-- stm32_can.c
|   |-- stm32_flash.c
|   |-- stm32_flash.h
|   |-- stm32_init.c
|   |-- stm32f10x_conf.h
|   |-- stm32f10x_it.c
|   |-- stm32f10x_it.h
|   \-- thumb_crt0.s
\-- tools
    |-- paulus_cksum
    |-- paulus_cksum.c
    \-- create
```

Beim Einsatz anderer Entwicklungsumgebungen ist auf die Einhaltung der Directory Struktur zu achten und auf die korrekte Einrichtung der Inklude-Pfade. Für das jeweilige Target müssen CAN Treiber und Flashroutinen bereitgestellt werden.

$(TargetDir)/$(Target)_flash.[ch]	FLASH Routinen
$(TargetDir)/$(Target)_can.[ch]	CAN Routinen
$(TargetDir)/$(Target)_init.c	CPU Initialisierung
$(TargetDir)/environ.h	allgemeiner Header

Alle diese Module greifen auf Funktionen der von ST bereitgestellten Firmware Library zurück. Die Headerfiles der Firmware Lib werden über die allgemeine Header Datei *environ.h* inkludiert. Dazu ist auch die Konfigurationsdatei der Firmwarelib, *stm32f10x_conf.h* anzupassen.

Die Initialisierung des CAN Controllers erfolgt im Modul *<target>/<target>_can.c*. Zuvor werden im Modul *<target>/targetw>_init.c* die IO Pins in Funktion und Zuweisung für CAN-RX und CAN-TX initialisiert. Entsprechend dem Typ und der Vielfalt der Möglichkeiten müssen dafür eventuell Anpassungen vorgenommen werden.
Prüfen Sie den Code in den Funktionen *RCC_Configuration*(), *GPIO_Configuration*() und *initDevice*() in der Datei *stm32_init.c* und die gesetzten `#define`STM32F10* in *bl_config.h*.

Im Paulus wird die Auto-Buson Funktionalität des CAN Controllers genutzt. Diese prüft dauerhaft auf einen Rezessiven Bus und holt dann den CAN Controller automatisch aus dem Bus-Off. Sollte ein Hardware-Defekt vorliegen, kann dies den Rest der Anlage stören.

Der Anwender hat im Modul *stm32_init.c* zwei Funktionen bereit zu stellen *getBitRate*() und *getNodeId()*. Typischerweise sind dazu Jumper auszulesen, oder die Werte sind in einem bestimmten FLASH Bereich gespeichert. Intern wird zur Kodierung der CAN Bit Rate der CANopen Index verwendete, eine Zahl zwischen 0 und 8. Zur besseren Lesbarkeit gibt es Defines der folgenden Art in *bl_config.h*

```
#define BL_USED_BITRATE_INDEX    BITRATE_INDEX_1000K
```

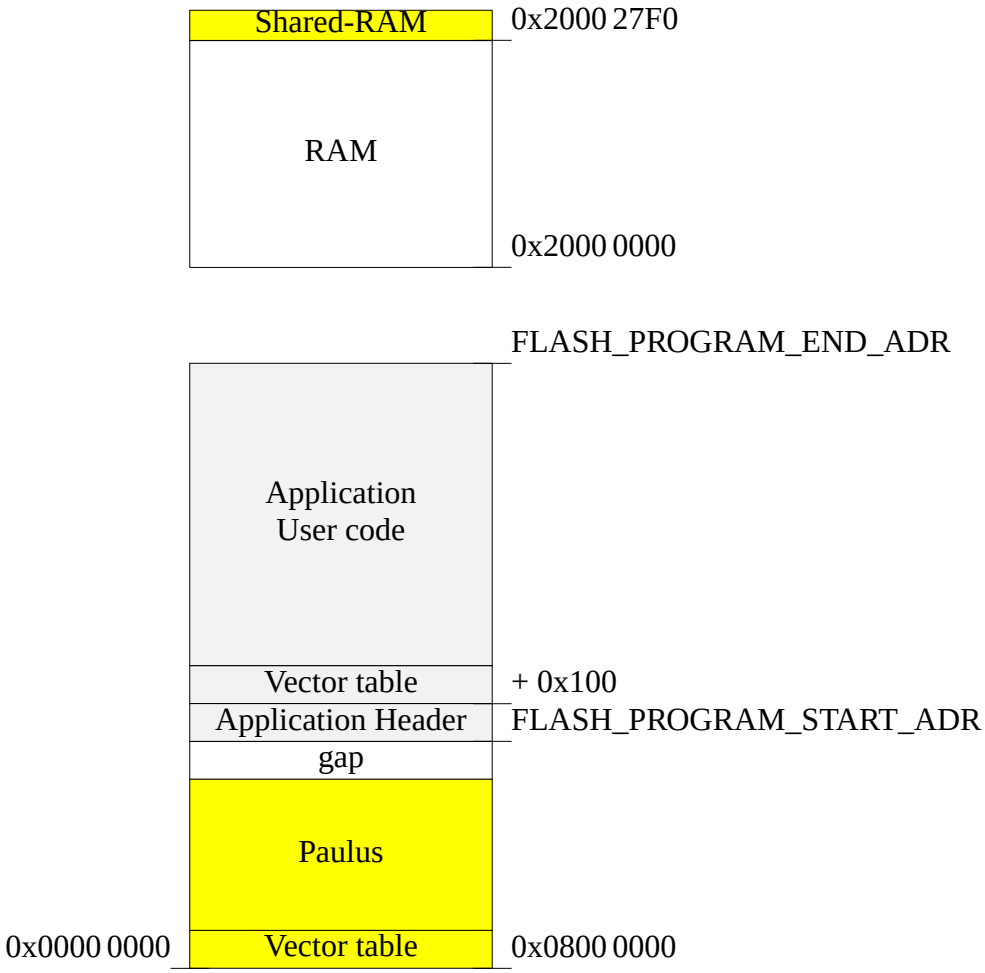
Während der Entwicklung kann eine Debug-Ausgabe über die serielle Schnittstelle aktiviert werden. Dazu wird im Header *bl_config.h* die Zeile

```
#define DEBUG 1
```

aktiviert.

Im Allgemeinen wird die Initilaisierungsroutine nur die absolut notwendigen Peripherie Funktionen, Takt-System, CAN, Speichermanagement, für die Zwecke von Paulus initialisieren. In Ausnahmen kann es aber sein, dass Paulus Initialisierungen vornimmt, welche durch die Anwendung nachgenutzt wird. Als Beispiele dafür mögen ein UART zur Debug-Ausgabe, oder eine LED zur Statusanzeige sein.

8.1. Speichernutzung



Speichernutzung (Beispiel mit 10 KiB RAM)

Die native Flash Programmierung der firmware Lib schreibt 2-Byte weise. Daher benutzt Paulus beim Schreibprozess die FirmwareLib Funktion *FLASH_ProgramHalfWord*(); Diese Funktion übernimmt die Programmierfreigabe, das Schreiben und das Warten auf Beendigung. Der Speicherbereich für die Anwendung, d.h. der Bereich welcher gelöscht und mit der Anwendung wieder beschrieben werden kann, ist in *stm32/stm32_flash.h* mit den Konstanten

```
#define FLASH_PROGRAM_START_ADR
#define FLASH_PROGRAM_END_ADR
```

festgelegt und ist bei Wechsel des Prozessortyp anzupassen. Dadurch wird der Bereich festgelegt, welcher beim Schreiben auf 0x1f51:1=3 gelöscht wird. Der Bereich welcher geschrieben wird, ist durch die Länge des Download-Image bestimmt.

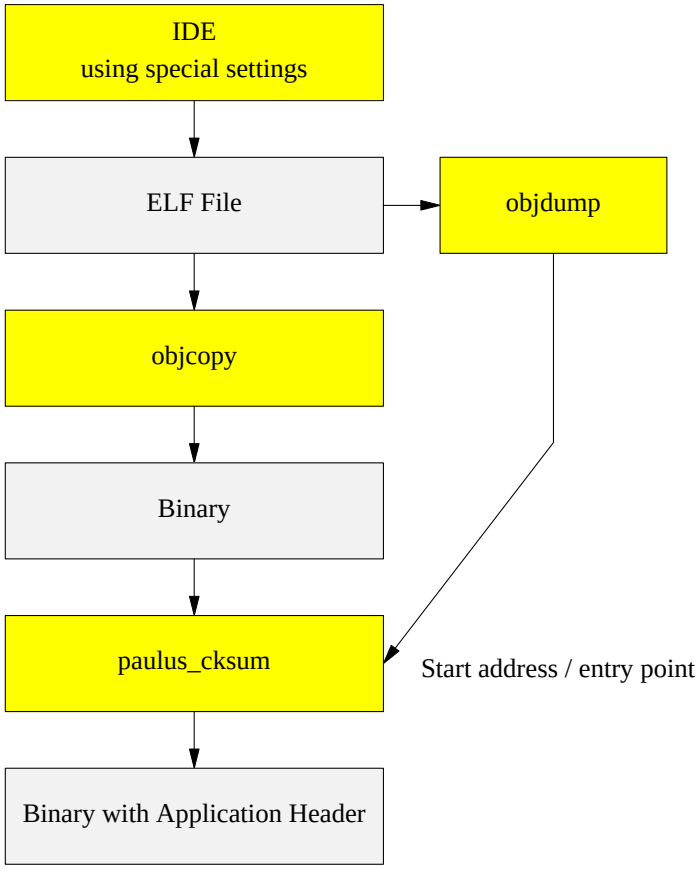
8.1.1. Code Größe

Die folgende Tabelle zeigt die typische Code Größe einer minimalen Paulus Variante[7].

section	size
.vectors	236
.init	384
.text	4.1 KiB
.rodata	564
.data	256
Gesamt	0x1660 (5.6 KiB)

8.2. Generierung des Anwender Image

Das Anwenderimage wird durch mehrere Schritte erzeugt.



Das Shell-Skript *tools/createstm32image* organisiert diesen Ablauf.

The following command sequence shows the result of objdump:

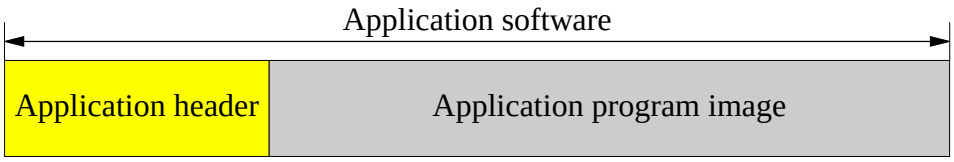
[7] ohne LSS

```
$ /usr/share/crossworks_for_arm_2.0/gcc/bin/objdump -f THUMB Release/hello.elf

THUMB Release/hello.elf:     file format elf32-littlearm
architecture: arm, flags 0x00000012:
EXEC_P, HAS_SYMS
start address 0x080022ed
```

In CrossWorks ist die *start address* die Adresse des *reset_handler* in *STM32_Startup.s*.

8.2.1. paulus_cksum

paulus_cksum berechnet die CRC-Prüfsumme über das Applikations-Programm im Binärformat, generiert den Applikations-Header und speichert den Applikations-Header zusammen mit dem Applikations-Programm in einer neuen Datei, die über den Bootloader im Gerät gespeichert werden kann.



Der Bootloader führt nach der Datenübertragung eine Prüfung der empfangenen Daten durch. Dazu ist dem Applikations-Programm ein Applikations-Header voranzustellen, der eine CRC Prüfsumme enthält. Dieser Header kann mit dem Tool **paulus_cksum** generiert werden. Die Struktur im Header ist die im Kapitel 7 beschriebene `APPLICATION_HEADER_T`.

Im Applikations-Header werden ungenutzte Bytes beim STM32 mit 0xFF belegt. Die Länge des Headers ist auf 256 Byte festgelegt.
Beispiel:

```
$ tools/paulus_cksum -v -l 256 -C -O download.bin -v -x $EXEC appl.bin
size: 0x00003524,  crc: 0x1cb5,  file: >appl.bin<
$ l appl.bin  download.bin
-rwxrwxrwx 1 oe users 13604  9. Sep 16:59 appl.bin*
-rw-rw-rw- 1 oe users 13732  9. Sep 17:00 download.bin
```

Neben der CRC Prüfung prüft der Bootloader auch die Längeninformation im Header. Ein Länge von 0 ist ungültig. Daher kann eine Anwendung auch diese Information mit dem ungültigen Wert 0 überschreiben, um die Anwendung als ungültig zu kennzeichnen. Dies ist im STM32 immer möglich, da gelöschter Inhalt 0xFF ist und ein überschreiben mit 0 erfolgen kann.

8.3. Erstellen der Anwendung

Die folgenden Ausführungen versuchen allgemein zu bleiben, beziehen sich in der Ausführung aber auf den Einsatz von CrossWorks für ARM Version 2.

8.3.1. Start Adresse

Wichtig ist, dass die Startadresse des Anwenderprogramms im FLASH mit der Information in der PAULUS Konfoguration des FLASH in *stm32/stm32_flash.h* übereinstimmt. Das Anwender Image wird nach dem Paulus-Code in den FLASH geladen. Das Image wird daher auf die Adresse

```
#define FLASH_DATA_START_ADR
```

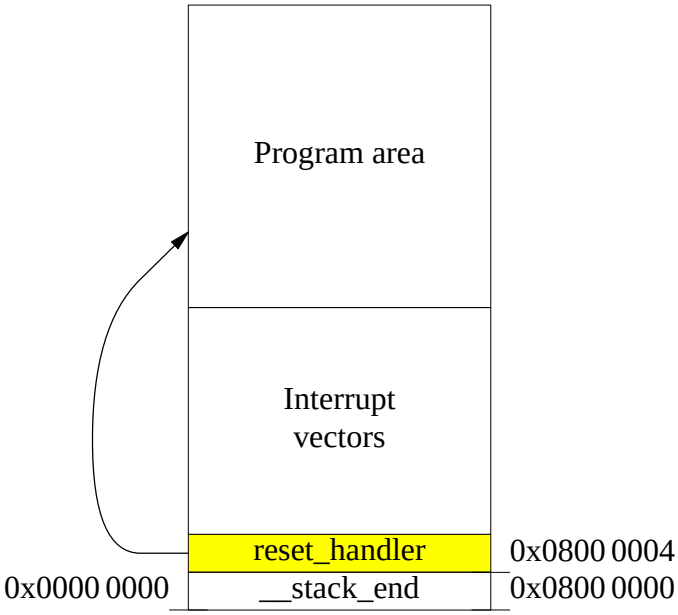
geflasht. Die eigentliche Anwendung beginnt 256 Bytes später, bedingt auch dadurch das die Interruptvektoren auf einer durch 0x100 teilbaren Adresse liegen müssen. Die dem Linker vorgegebene Start Adresse wird z.B bei CrossWorks mit dem

```
linker_section_placement_macros="FLASH_START=0x8002100"
```
[8]

Aus dem STM32 Handbuch:

> *After this startup delay has elapsed, the CPU fetches the top-of-stack value from address 0x0000 0000, then starts code execution from the boot memory starting from 0x0000 0004.*

Ein STM32 Programm startet wie im folgenden gezeigt (Adress Namen wie im CrossWorks):



Damit die Anwendung richtig startet, sollte in der **Release**-Version das *define* `STARTUP_FRO_RESET` beim Kompilieren von *STM32_Startup.s* gesetzt werden. Andernfalls zeigt der *reset_handler* zu einer Endlos-Schleife *reset_wait*, um einem Debugger die Möglcikeit zu geben das Programm nach einem ‘Reset definiert zu stoppen.

[8] unter der Annahme Paulus belegt 0x2000 Byte vergeben.

8.4. Anwendungsbeispiel

Der Paulus Bootloader und eine Beispielanwendung sind in der Auslieferung als zwei Projekte zu einer *Solution* zusammengefasst. Die Anwendung ist das Projekt **hello** welches über UART Instruktionen ausgibt und durch Bedienereingaben über UART verschiedene Rücksprungmöglichkeiten zum PAULUS zeigt. Dabei kommunizieren PAULUS und Anwendung über einen gemeinsamen Speicherbereich im RAM in dem jeder bestimmte Signaturen, bestehend aus 4 Byte, ablegt.

9. Portierung zum dsPIC33

Die Portierung wurde mit der Microchip Entwicklungsumgebung MPLAB ausgeführt. In der Wurzel des Projektverzeichnisses liegt die Projekt-Datei *paulus_dspic.mcp*. Das folgende Bild zeigt die Verzeichnisstruktur:

```
|-- bootloader
|   |-- bl_can.h
|   |-- bl_canopen.c
|   |-- bl_canopen.h
|   |-- bl_config.h_template
|   |-- bl_crc.c
|   |-- bl_hw.h
|   |-- bl_type.h
|   \-- bl_user.c
|-- dsPIC
|   |-- Readme
|   |-- bl_config.h
|   |-- bl_flash.h
|   |-- bl_interface.c
|   |-- bl_interface.h
|   |-- dspic_appl.c
|   |-- dspic_can.c
|   |-- dspic_flash.c
|   |-- dspic_flash.h
|   |-- dspic_init.c
|   |-- environ.h
|   |-- p33FJ256GP710.gld  >gcc 3.20
|   \-- p33FJ256GP710_old.gld <gcc 3.20
|-- eds
|   |-- paulus.can
|   |-- paulus.eds
|   \-- paulus.html
|-- examples    Example projects
|-- main.c
|-- paulus_dspic.bin
|-- paulus_dspic.cof
|-- paulus_dspic.hex
|-- paulus_dspic.map
|-- paulus_dspic.mcp  Paulus project
|-- paulus_dspic.mcs
|-- paulus_dspic.mcw
|-- tools
|   |-- paulus_cksum
|   |-- paulus_cksum.c
|   \-- dsPIC_binutils
|       |-- objcopy
|       \-- objdump
\-- version.h
```

Für die Portierung wurden folgende Komponenten genutzt:

Derivate	dsPic33FJ256GP710
Quarz	8MHz
IDE	MPlab 8.60
Compiler	pic30-gcc v3.24

Die Einstellungen für den CAN und den Flash sind vom verwendeten Derivate abhängig. Desweiteren bestehen Abhängigkeiten zum verwendeten Takt.

dsPIC/dspic_can.[ch]	CAN Routinen
dsPIC/dspic_init.c	CPU Initialisierung
dsPIC/dspic_flash.[ch]	FLASH Routinen
dsPIC/environ.h	allgemeiner Header
dsPIC/bl_interface.[ch]	Interface zu Applikation

Der Bootloader verzichtet zugunsten der Codegröße auf die Verwendung von Interrupts. Daher ist die IVT frei für die Verwendung durch die Applikation. Nur der Reset-Vektor muß auf die Einsprungadresse des Bootloaders zeigen, damit dieser in jedem Fall zu Beginn angesprungen wird und eine Checksummeprüfung durchführen kann. Verzichtet man hierauf, dann muß die Applikation den Bootloader anspringen und kann den Reset-Mechanismus, welcher auch die gesamte Peripherie in den Resetzustand versetzt, nicht nutzen.

Der Reset-Vektor wird beim Erase gelöscht und mit dem Wert des Applikations-Images beschrieben.

Für die Anpassung der Flash Routinen sind einige Anpassungen nötig. Diese finden sich in *dspic_flash.h* und sind Derivate-abhängig.

```
/* (reserved) Paulus code size */
#define FLASH_SIZE_PAULUS     16   /* in KiB */

/* first flash address (incl. Bootloader) */
#define FLASH_START_ADR            0x00000000ul

/* Define the FLASH Page Size depending on the used device */
#define FLASH_ERASE_PAGE_SIZE (512*2)
/* Number of words to gbe flashed at a time */
#define FLASH_PAGE_SIZE  (64*4)         /* in words */
```

Das Flashen beginnt bei Adresse 0. Die Applikation beginnt jedoch erst nach dem Bootloader. Zum Überspringen des Bootloaders werden die jeweiligen Adressen benötigt.

```
/** Applikationsstartaddr im Flash (incl header ) */

/* Reset vector - start of flashing */
#define FLASH_PROGRAM_START_ADR 0x0000ul

/* word address */
#define FLASH_PROGRAM_REAL_START_ADR 0x4000ul

/** max. Application size im Flash (incl header) */
#define FLASH_PROGRAM_MAX_SIZE     \
     (FLASH_PROGRAM_END_ADR - FLASH_PROGRAM_START_ADR + 1)

/* max siez without vectors - word size */
#define FLASH_PROGRAM_REAL_MAX_SIZE     \
     (FLASH_PROGRAM_END_ADR - FLASH_PROGRAM_REAL_START_ADR + 1)
```

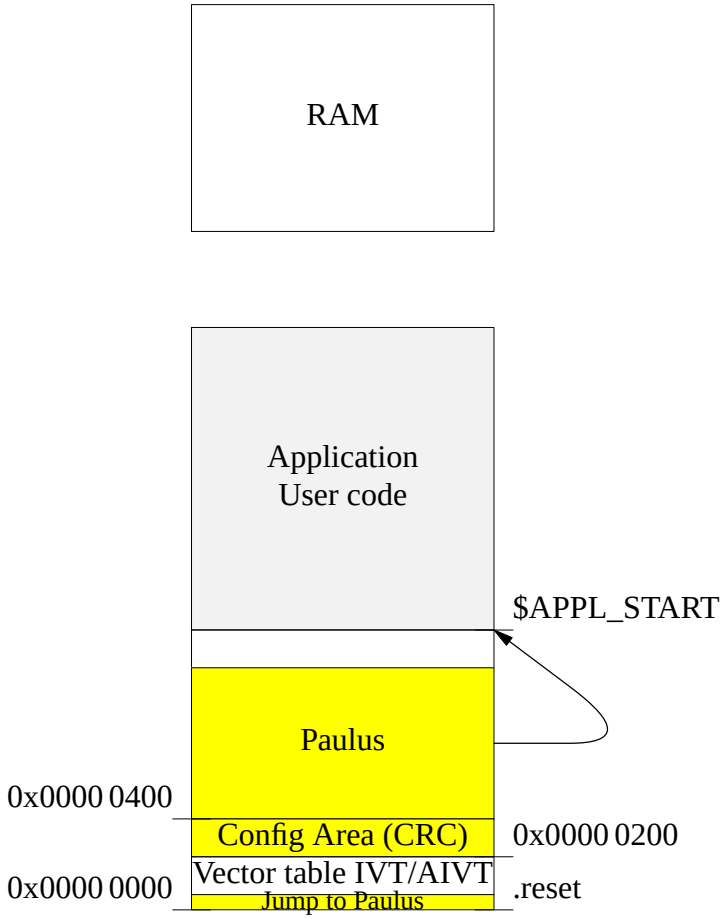
Die maximale Größe der Applikaiton und das Flash Ende ist vom gewählten Derivate abhängig.

```
/* FLASH config data - word address */
#if defined(DSPIC33FJ64)
#define FLASH_PROGRAM_END_ADR 0xABFF

#elif defined(DSPIC33FJ128)
#define FLASH_PROGRAM_END_ADR 0x157FF

#elif defined(DSPIC33FJ256)
#define FLASH_PROGRAM_END_ADR 0x2ABFF /* word address */

#else
# error "One DSP version has to be specified"
#endif
```

Wird die Größe des Paulus nicht geändert, sollte sich die Anpassung auf den Wert für `FLASH_PROGRAM_END_ADR` beschränken. Es empfiehlt sich, die restlichen Werte zu prüfen.

Der Wert für `FLASH_PROGRAM_REAL_START_ADR` muß mit den Linkersettings von Paulus (hinter Memory Bereich **program**) und den Applikation Linker Settings (Beginn von **program**) übereinstimmen!

9.1. Speicheraufteilung



Der Wert für APPL_START ist derzeit sehr reichlich auf 0x4000 gesetzt. Dies ermöglicht eine einfache Inbetriebnahme mit Debugging und reichlich printf-Ausgaben. Nachdem man seine eigenen Anpassungen durchgeführt hat, kann dieser Wert optimiert werden, um mehr Platz für die Applikation zur Verfügung zu haben.

9.2. Applikation

Im Linkerfile (z.B. *paulusExample_p33FJ256GP710.gld*) muß der Reset-Vektor wie immer auf Adresse 0 liegen. Er muß jedoch auf den Beginn von Paulus verweisen.

```
__CODE_BASE = 0x4000;
__BL_BASE = 0x400;
```

```
SECTIONS
{
  .reset :
  {
        /* Jump to the boot-loader entry */
        SHORT(ABSOLUTE(__BL_BASE));
        SHORT(0x04);
        SHORT((ABSOLUTE(__BL_BASE) >> 16) & 0x7F);
        SHORT(0);
  } >reset

}
```

Die Applikation selbst beginnt hinter Paulus.

```
MEMORY
{
    program (xr) : ORIGIN = 0x4000,         LENGTH = 0x26C00
}
```

Wenn alles korrekt ist, findet man im Linkerfile auf der ersten möglichen Adresse den Einsprungpunkt.

```
0x004000                  __resetPRI
```

Die Applikation sollte keine Fuse-Einstellungen enthalten. Diese vergrößern nur unnötig das Image. Paulus nutzt seine eigenen Fuse-Settings.

9.2.1. Debuggen

Während der Entwicklung der Applikation sollte man auf den Bootloader verzichten. Für das Debuggen mit Bootloader empfiehlt sich folgendes Vorgehen.

- Auf dem dsPic33 sollte sich bereits der Paulus befinden. Dieser wurde im Debugger-Modus geflasht.
- Im MPLAB ist das Projekt der Applikation geöffnet.
- Die Applikation wird erstellt. Das Download-Image wird erstellt. Im Debugger wird der Paulus per ’RUN’ aktiviert. Der Hinweis, dass sich der Speicher geändert hat, wird ignoriert Das Image wird per Paulus geflashed (nicht per Debugger).
- Nach einem Reset des Prozessors prüft der Paulus die Checksumme und startet die Applikation.
- Nun kann man wie gewohnt debuggen.

9.3. Binär-Image

Das Binär-Image ist derart aufgebaut, dass die Verarbeitung innerhalb der Paulus möglichst einfach ist. Es werden Routinen von MPLAB genutzt:

- _write_flash24() - flashing the image, needs 4 byte per instruction
- _memcpy_p2d24() - for CRC calculating of the application supplies 3 byte per instruction
- _memcpy_p2d16() - for data reading from flash, 2 byte per instruction

Instruction:

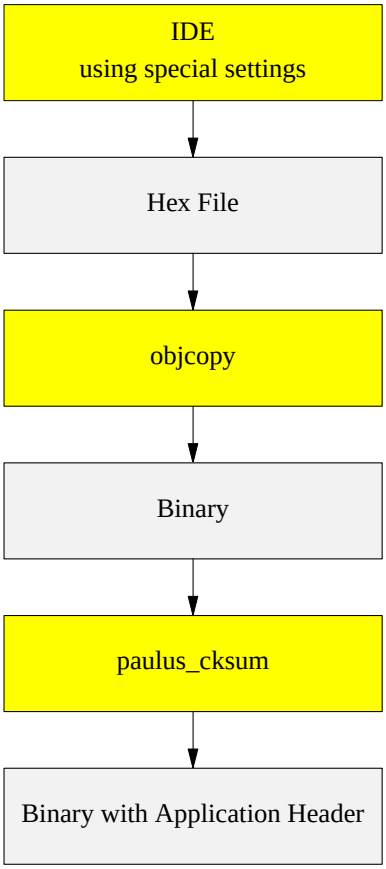
Doubleword			
0x00	Instruction		
		Constants	
Byte 3	Byte 2	Byte 1	Byte 0

Für die CRC Berechnung wird Byte 3 ignoriert.

Der Einfachheit halber wird der Adressbereich des Bootloaders im Image mit übertragen, jedoch nicht geflashed.

9.4. Generierung des Anwender Image

Das Anwenderimage wird durch mehrere Schritte erzeugt.



Das Shell-Skript *create* organisiert diesen Ablauf.

9.4.1. objcopy

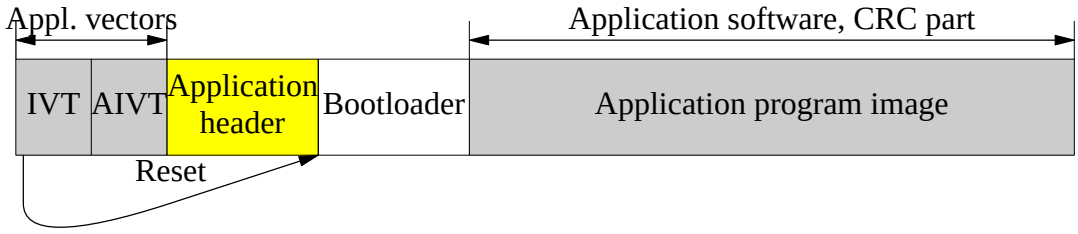
Das Binärimage wird aus dem erzeugten iHex-File erstellt. Es empfiehlt sich, keine Fuse-Bit Konfiguration in die Applikation hineinzucompilieren. Es wird die Fuse-Konfiguration des Paulus genutzt. Somit ist die Fuse-Konfiguration innerhalb der Applikation unnötig und vergrößert das Binär-Image erheblich.

```
objcopy -I ihex -O binary --gap-fill 0xFF appl.hex appl.bin
```

-I ihex	Inputformat ihex
-O binary	Outputformat binary
−−gap-fill 0xFF	fill gaps with 0xFF

9.4.2. paulus_cksum

paulus_cksum berechnet die CRC-Prüfsumme über das Applikations-Programm im Binärformat, generiert den Applikations-Header und speichert den Applikations-Header zusammen mit dem Applikations-Programm in einer neuen Datei, die über den Bootloader im Gerät gespeichert werden kann.



Der Bootloader führt nach der Datenübertragung eine Prüfung der empfangenen Daten durch. Dazu ist dem Applikations-Programm ein 256 Byte langer Applikations-Header voranzustellen, der eine CRC Prüfsumme enthält. Dieser Header kann mit dem Tool **paulus_cksum** generiert werden. Die Struktur im Header ist die im Kapitel 7 beschriebene `APPLICATION_HEADER_T`.

Bsp: dsPic33FJ256GP710

```
paulus_cksum -P -C -a 0x400 -b 0x8000 -c 0x55800 -x 0x4000 \
-O appl.crc appl.bin
```

-P dsPic33
-C CANopen CRC
-a CRC Block address (byte address)
-b Application start address (byte address)
-F Flash end address (byte address)
-O output file (incl appl.bin) == complete domain
-x Entry point (without conversion)

Die Flash Ende Adresse wird benötigt, wenn die Applikation die Configuration Bits enthält. Diese müssen aus dem Image entfernt werden. Die verwendeten Adressen sind Byte-Adressen. Die beim dsPic33 üblichen Word-Adressen müssen somit verdoppelt werden.