Preliminary Application Note

Bootloader for 78K0/Kx2

8-Bit Single-Chip Microcontroller

78K0/Kx2

Document No. U18539EE1V0AN00

Date published 15/02/07

Printed in Germany

Legal Notes

- **The information contained in this document is being issued in advance of the production cycle for the product. The parameters for the product may change before final production or NEC Electronics Corporation, at its own discretion, may withdraw the product prior to its production.**

- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.

- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.

- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.

- NEC Electronics products are classified into the following three quality grades: "Standard", "Special", and "Specific". The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics products before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

Notes for CMOS Devices

1. **VOLTAGE APPLICATION WAVEFORM AT INPUT PIN**
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between VIL (MAX) and VIH (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between VIL (MAX) and VIH (MIN).

2. **HANDLING OF UNUSED INPUT PINS**
   Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to VDD or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

3. **PRECAUTION AGAINST ESD**
   A strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

4. **STATUS BEFORE INITIALIZATION**
   Power-on does not necessarily define the initial status of a MOS device. Immediately after the power source is turned ON, devices with reset functions have not yet been initialized. Hence, power-on does not guarantee output pin levels, I/O settings or contents of registers. A device is not initialized until the reset signal is received. A reset operation must be executed immediately after power-on for devices with reset functions.

5. **POWER ON/OFF SEQUENCE**
   In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current. The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

6. **INPUT OF SIGNAL DURING POWER OFF STATE**
Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

Regional Information

Some information contained in this document may vary from country tocountry. Before using any NEC product in your application, please contact theNEC office in your country to obtain a list of authorized representatives anddistributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and otherlegal issues may also vary from country to country.

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668, Japan
Tel: 044 4355111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554,
U.S.A.
Tel: 408 5886000
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211 65030
http://www.eu.necel.com/

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908 691133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01 30675800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091 5042787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 6387200

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02 667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven,
The Netherlands
Tel: 040 2654010

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27
ZhiChunLu Haidian District,
Beijing 100083, P.R.China
Tel: 010 82351155
http://www.cn.necel.com/

NEC Electronics Shanghai Ltd.
Room 2511-2512, Bank of China
Tower,
200 Yincheng Road Central,
Pudong New Area,
Shanghai 200120, P.R. China
Tel: 021 58885400
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
12/F., Cityplaza 4,
12 Taikoo Wan Road, Hong Kong
Tel: 2886 9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R.O.C.
Tel: 02 27192377

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253 8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku, Seoul,
135-080, Korea Tel: 02-558-3737
http://www.kr.necel.com/

Table of Contents

Chapter 1 Introduction

This application note describes how to implement a bootloader using the 78K0/Kx2 family microcontroller. Before going into detail of the bootloader code it is as well to explain what a bootloader is and the advantages of using one.

1.1 Definition and advantages of a bootloader

What is a bootloader? The “boot” part of the name comes from the fact that a bootloader is a piece of code that executes when the microcontroller powers up or “boots”. The “loader” part of the name comes from the fact that the main function of the bootloader is to “load” new application code if requested. “Load” in this context means getting the data into the microcontroller and writing it into flash memory.

The following picture shows the general bootloader flow.

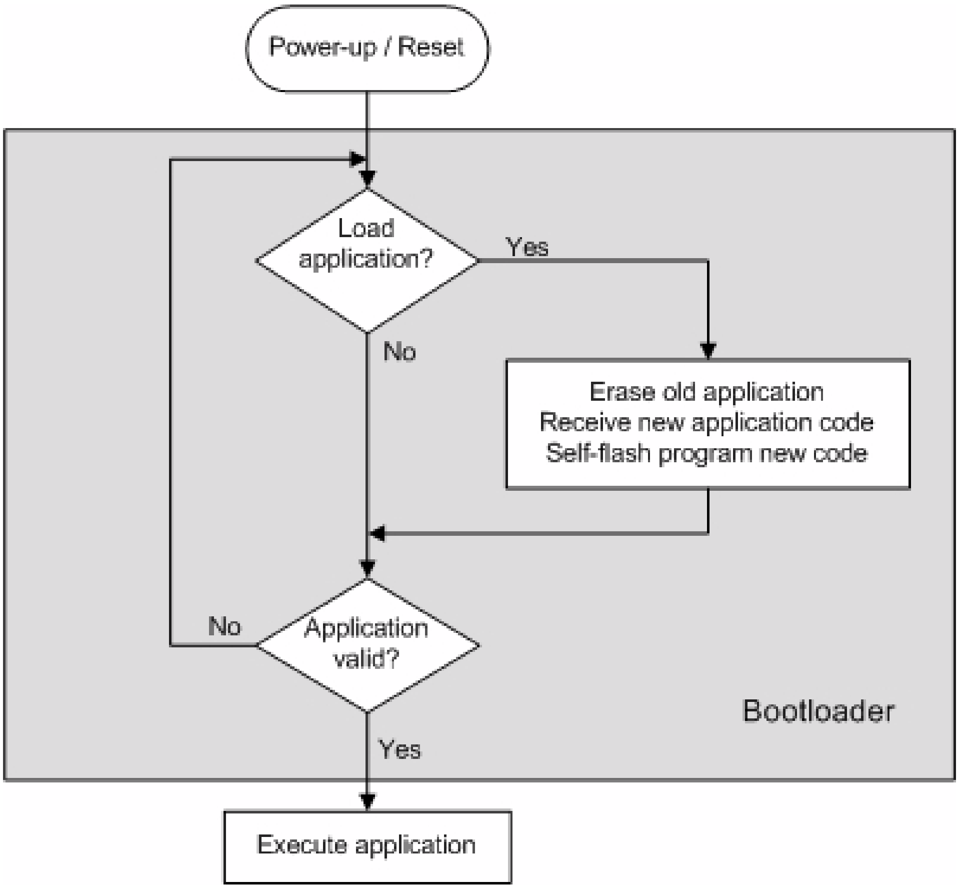


Figure 1-1 General bootloader flow

The main advantage of a bootloader is the ability to update or replace your application code without needing to use an external programmer. It opens up the possibility of updating your code remotely over a phone line, internet connection etc. A good example would be a micro that controls pay phones. If there were 5.000 microcontroller based pay phones in Germany and the phones needed a firmware update, the phone company would have two choices of how to perform the upgrade. One way would be for the phone company to drive around Germany and manually reprogramming all 5.000 phones one at a time using an external programmer. This would be a very costly time consuming effort. But if the micro has a bootloader, all 5.000 phones could be reprogrammed remotely from one central location.

1.2 Main tasks of a Bootloader

Signal to start the bootloading process

Let's say you have hundreds of vending machines all connected to the Internet and you want to update their firmware. Some signal is needed to tell the microcontroller to start the bootloading process. This could be an interrupt, a command byte sent over a serial channel etc. This will cause the program to reset and to run the bootloader code.

Signal to execute the bootloader

When the microcontroller starts up, it looks for an external signal to determine whether a new application is to be loaded, or whether it should execute the existing application. This can be as simple as checking a port pin on power up and making a decision based on whether the pin is high or low. It could also be based on a character received by the UART, or the reading taken by the ADC. It is up to the user to decide how to implement this.

Getting the new code into the microcontroller

The data can come in over a RS232, CAN, etc. serial port, or in parallel over a number of port lines. The user decides how the data gets into the microcontroller. The data coming in will typically be much larger than can be held in the microcontroller RAM, so there must be some provision to control the flow of the data. For an RS232 serial port, one solution would be to use a slow baud rate so the microcontroller has time to process the data and self-flash program it without being overrun. Another would be to use hardware handshaking using CTS and RTS lines to control the flow of data. Another would be to use software handshaking using the XON/XOFF protocol. The new code can be in any format decided by the user but will need to contain addressing information as well as checksums for error processing. Typically a standard format like Intel-Hex will be used.

Self-flash programming the new code

Each time the microcontroller receives a new batch of data it must self-flash program it into the correct memory locations. If the locations are not already blank, they must be erased before programming. Also, typically, they will be verified during or after programming.

Transferring control to a valid application program

Once the new code has been received and programmed successfully, the bootloader will write a checksum or other unique byte sequence to a fixed memory location. The bootloader checks for this valid application checksum or byte sequence. If present, it will transfer control to the application.

Chapter 2 Flash Memory Programming

Before going into the detail of the bootloader it is essential to understand the architecture of the memory and the usage of the self-programming. This description will focus on the µPD78F0547 microcontroller of the 78K0/Kx2 family.

2.1 Memory organization

The following figure shows the architecture of the flash memory(µPD78F0547). All 78K0/Kx2 devices have a common flash memory area and each device differs by the number of additional memory banks in its architecture.

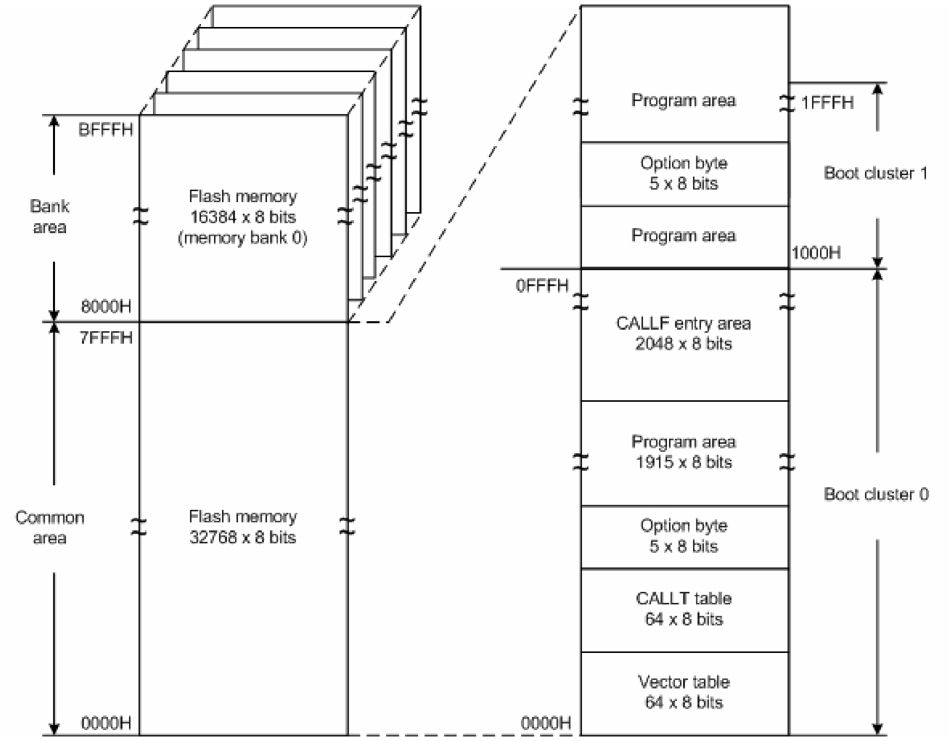


Figure 2-2 **This figure shows the memory architecture of the µPD78F0547**

The flash memory is divided into blocks of one kilobyte as shown in the following table. This is the smallest amount of memory that can be blank checked, erased or verified by the firmware.

Block	Address range
Block 00H	0000H - 03FFH
Block 01H	0400H - 07FFH
Block 02H	0800H - 0BFFH
....	....

2.1.1 Boot cluster

Within the common area are two boot clusters(bootcluster 0 and bootcluster 1), which have a size of 4 KB. The primary bootcluster(bootcluster 0) will be used for the bootloader whereas the second bootcluster is designed to temporarily save a new bootloader and perform a secure Boot Swap. The bootcluster 0 can be secured by flags, for example, to avoid the user accidentally erasing the bootcluster.

2.1.2 Difference in Representation of Memory Space

With 78K0/Kx2 products which support memory banking, addresses can be viewed in the following two different ways.

- Memory bank number + CPU address
- Flash memory real address (for flash programming)

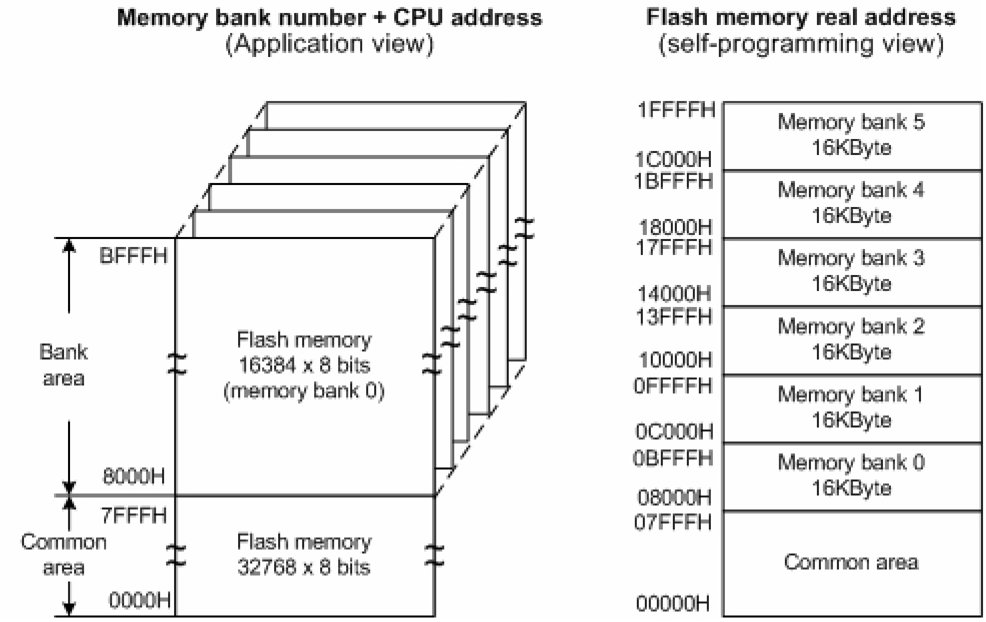


Figure 2-3 Memory addressing

The update file format must use real flash addresses which is required by the bootloader for the self-programming, whereas for the CRC calculation, the bootloader uses the "Memory bank number + CPU address" instead.

2.2 Self programming

As previously mentioned the microcontroller family 78K0/Kx2 supports self programming and allow the bootloader to write new application or bootloader into the flash memory. The write access to the flash memory will be processed by the firmware. The following figure shows the access flow to the flash memory.

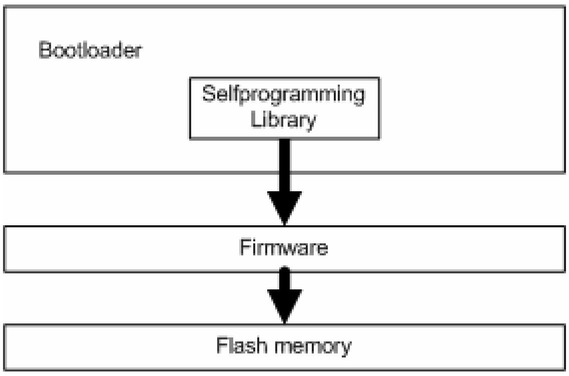


Figure 2-4 Access flow from bootloader to flash memory access

Below will be described the usage of the self-programming library.

2.2.1 Hardware environment

To execute self programming, a circuit that controls the voltage on the FLMD0 pin of the 78K0/Kx2 is necessary. The voltage on the FLMD0 pin must be low while an ordinary user program is being executed (in normal operation mode) and high while self programming is being executed (in flash rewriting mode).

If the firmware and software for rewriting runs while the FLMD0 pin is low, the circuit for rewriting flash memory does not operate. Therefore, the flash memory is not actually rewritten. To rewrite the flash memory, the FLMD0 pin must be pulled high by manipulating a general purpose port.

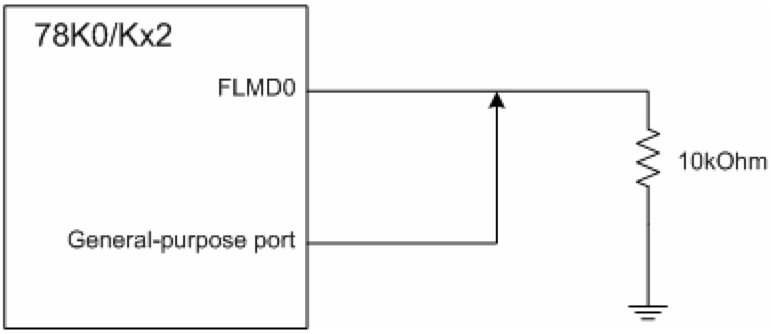


Figure 2-5 FLMD0 pin must be set on high for self programming

2.2.2 Software environment

To control the self-flash programming process there are three operating modes for the microcontroller which are described below.

Mode	Description
Normal Mode	- Execution of user application - After RESET operation starts in this mode
Mode A1	- Setting up self-programming - During this mode the firmware can be executed via CALL 8100H
Mode A2	- The firmware functions will be executed - This mode is not visible to the user

In the normal mode the application will be executed and the firmware is not visible for the user. The firmware will be activated by the self-programming library when an instruction CALL 8100H is performed.

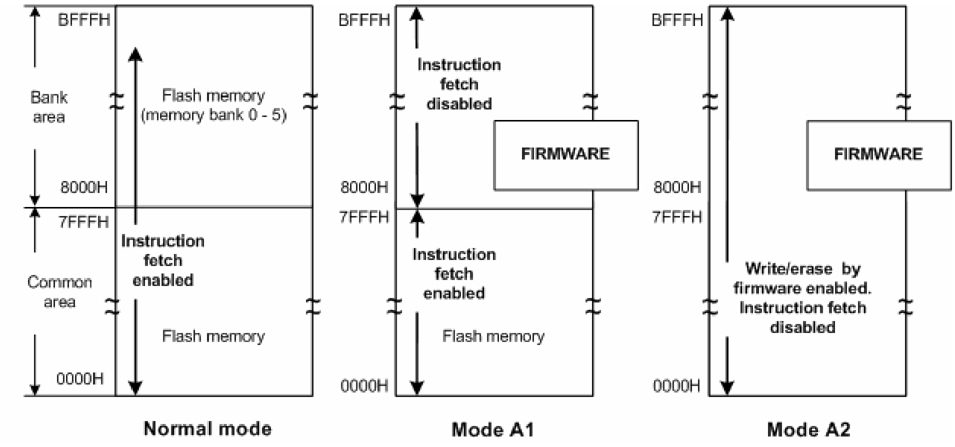

BFFFH
Bank area
8000H
7FFFH
Common area
0000H
Flash memory (memory bank 0 - 5)
Instruction fetch enabled
Flash memory
Normal mode
BFFFH
Instruction fetch disabled
FIRMWARE
8000H
7FFFH
Instruction fetch enabled
Flash memory
0000H
Mode A1
BFFFH
FIRMWARE
8000H
7FFFH
Write/erase by firmware enabled. Instruction fetch disabled
0000H
Mode A2

Figure 2-6 Memory access during self-programming

As you can see in the figure above, the bootloader/application(which will use the self-programming library) must be located within the common area, otherwise an instruction cannot be fetched in the Mode A1. The self-programming environment will be set up by the SelfLib_Open and SelfLib_Init function.

Activities during SelfLib_Open

- FLMD0 pin will be pulled up by the user defined general purpose port.
- Backup all registers from the register bank 3, which will be used by the firmware**Note**
- Backup user defined interrupt controller configuration and mask out all interrupts**Note**

Activities during SelfLib_Init

- Initialization of the pointer to the user defined data-buffer. This data-buffer will be used for data exchange between firmware and bootloader.

The SelfLib_Close function will close the self-programming environment.

Activities during SelfLib_Close

- FLMD0 pin will be pulled down by user defined general purpose port.
- Restore all registers from the register bank 3
- Restore user defined interrupt controller configuration

Note This feature is by default disabled within the bootloader!

The following figure illustrates the complete flash erase process.

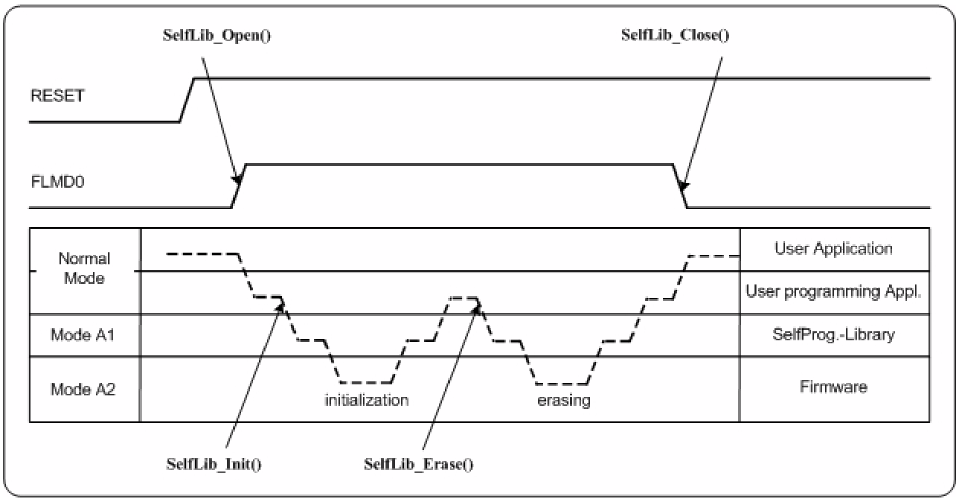


Figure 2-7 Mode transitions during self-programming

The following self-programming functions are used by the bootloader:

SelfLib_EepWrite	Write flash word on defined address.
SelfLib_Verify	Verify the voltage levels on the written block.
SelfLib_Erase	Erase flash block
SelfLib_ModeCheck	Check the voltage level on FLMD0 pin.
SelfLib_BlankCheck	Specified block will be blank checked.
SelfLib_SetInfo_SwapBootCluster	Inverts the current value of the boot flag inside the extra area[Note]
SelfLib_GetInfo_BootCluster	Read the boot flag , whether the two boot cluster are swapped

Note Security information is located within an extra area(boot-flag, chip erase protection, block erase protection, write protection, boot cluster protection). Security information can only be read/written using the self-programming functions.

Boot swap function

The self-programming environment allows the bootloader to be updated in a secure way by the boot swap function. Even if a power failure occurs during an update, the old bootloader can always be used until the boot swap is completed. The following figure illustrates the boot swap steps.

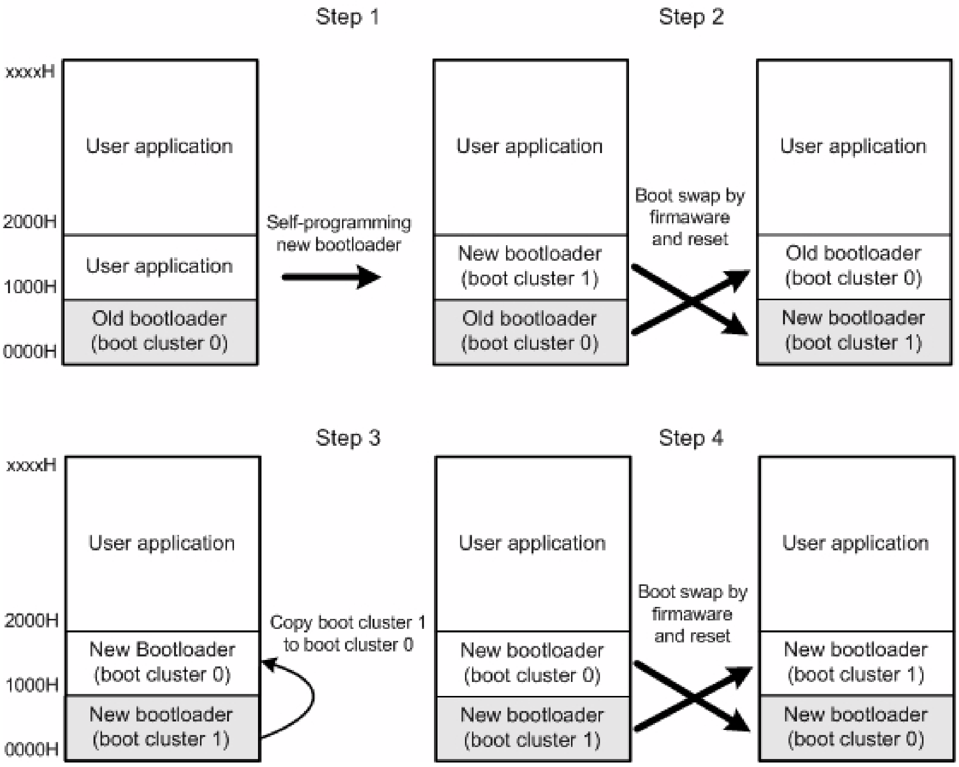

Step 1
Step 2
xxxxH
2000H
1000H
0000H
User application
User application
Old bootloader (boot cluster 0)
Self-programming new bootloader
New bootloader (boot cluster 1)
Boot swap by firmaware and reset
Old bootloader (boot cluster 0)
New bootloader (boot cluster 1)
Step 3
Step 4
New Bootloader (boot cluster 0)
New bootloader (boot cluster 1)
Copy boot cluster 1 to boot cluster 0
New bootloader (boot cluster 0)
New bootloader (boot cluster 1)

Figure 2-8 **Boot swap states**

Step 1:	Self-programming the new bootloader into the boot cluster 1
Step 2:	Execute the SelfLib_SetInfo_SwapBootCluster function to set the swap bit within the extra area. Force a hardware reset.
Step 3:	Copy the new bootloader into boot cluster 0.
Step 4:	Execute the SelfLib_SetInfo_SwapBootCluster function to reset the swap bit within the extra area. Force a hardware reset.

Note **After the set/reset of the boot swap flag above, the microcontroller must be reset to actively swap the physical addresses.**

Chapter 3 Bootloader

This chapter describes the features and the use of the bootloader.

3.1 Specification

Bootloader specification	
Frequency	12 MHz or 20 MHz(can be adapted by user)
UART6	Baud rate: 115200 bps or 57600 bps(can be adapted by user) Data bits: 8 Bit Stop bits: 1 Bit Parity: No parity Flow control: XON/XOFF
Supported file format	Intel-Hex-Standard or Intel-Hex-Extended Note 1
Update methods	Application or bootloader update. User can disable the bootloader update feature.
Interrupts	Interrupts are not allowed during the bootloader process Note 2
Watchdog	Will be stopped by bootloader to execute self programming functions, but can be used by the application.

Note 1. File must be sorted by addresses from low to high and the gaps must be filled
2. Interrupts within the application will be processed by virtual interrupt table

Software environment	
Software	**Version**
IAR C/C++ Compiler for NEC 78K0 and 78K0S	V4.40B (4.40.2.3)
IAR Assembler for NEC 78K0 and 78K0S	V4.40A (4.40.1.3)
IAR XLINK (Linker)	4.60C (4.60.3.0)
IAR Embedded Workbench	4.6B (4.6.2.0)
RealTerm Note	1.99.0.24

Note RealTerm is a serial terminal: http://realterm.sourceforge.net

Bootloader timing	
Conditions: Baud rate is 115200 bps, Intel-Hex-File size: 349KByte, Code size: 124KByte	
Receive buffer size	**Update time**
200Byte	6:50min
400Byte	4:10min

Bootloader size for banked model(optimized for size)		
Optimization level	**Bootloader update allowed**	**Bootloader update is not allowed**
None	4027Bytes	3607Bytes
Low	3902Bytes	3517Bytes
Medium	3710Bytes	3350Bytes
High	3540Bytes	3186Bytes

Bootloader size for non banked model(optimized for size)		
Optimization level	**Bootloader update allowed**	**Bootloader update is not allowed**
None	3969Bytes	3549Bytes
Low	3847Bytes	3462Bytes
Medium	3655Bytes	3295Bytes
High	3489Bytes	3135Bytes

3.2 Hardware requirements

The following figure shows the circuit which is required by the bootloader.

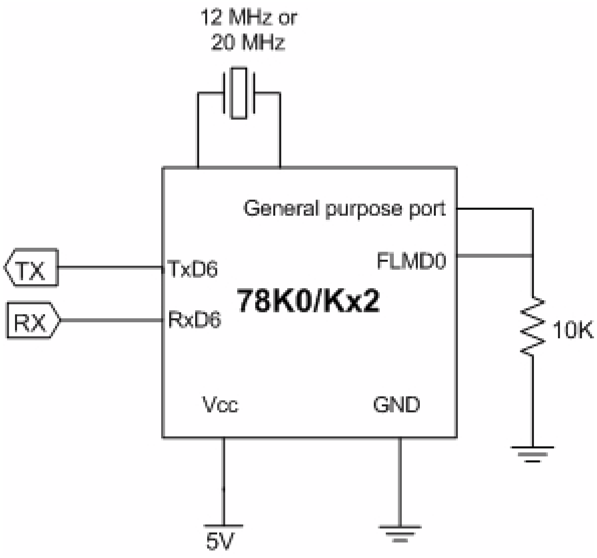


Figure 3-9 Hardware requirements for the bootloader

To use the bootloader, the FLMD0 pin is pulled-up by the user using the general purpose port.

3.3 Bootloader concept

3.3.1 Bootloader vs. Application

The use of a bootloader can restrict some microcontroller features(e.g. CALLT, CALLF) for the application. As you can see in the figure below the application does not have any access to the bootloader area(0x0000 - 0x0FFF). Following features can not be used/changed by the application:

CALLT	Can only be used by the bootloader
CALLF	Can only be used by the bootloader
Option byte	Is predefined by the bootloader
Interrupt vector table	Is predefined by the bootloader for the application(see below)

The following figure illustrates the differences between bootloader and application view.

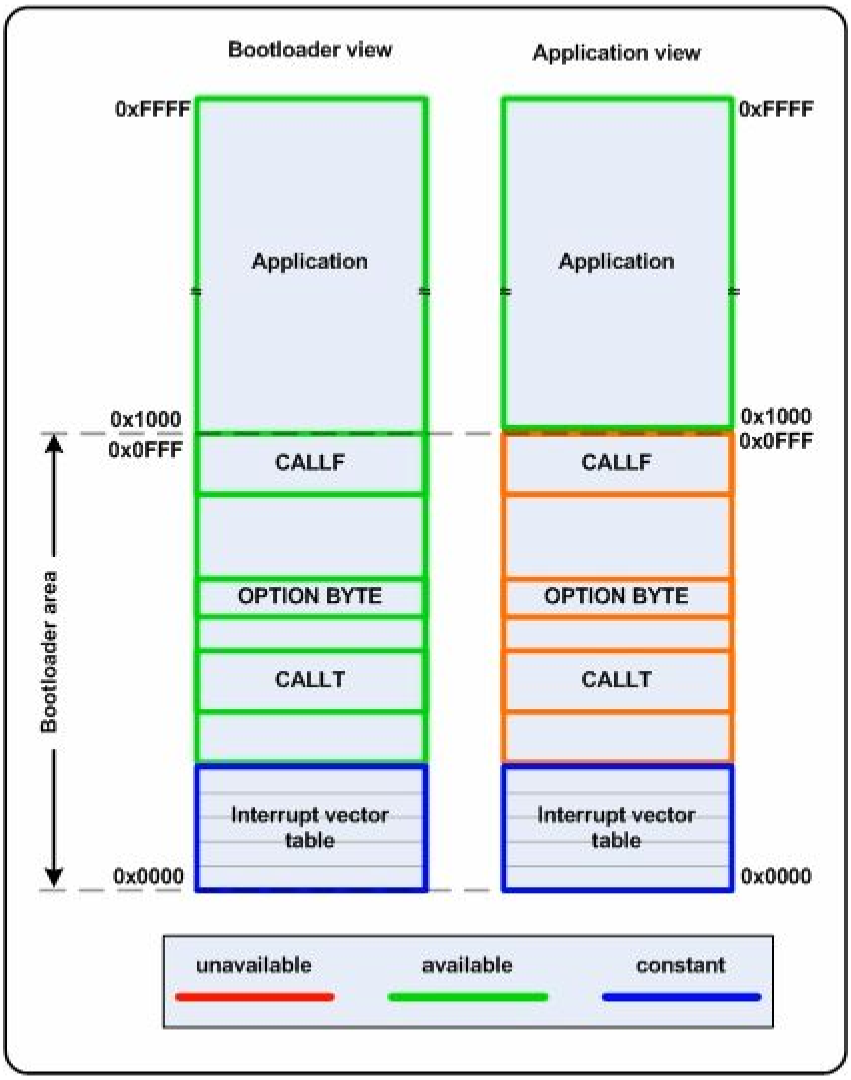


Figure 3-10 Bootlaoder vs. Application

The interrupt vector table must be constant, because an application will be compiled and linked independently from the bootloader, the link between interrupt vector(bootloader) and interrupt service routine(application) does not exist.

3.3.2 Virtual interrupt vectors

As previously described, the interrupt vector table in the bootloader area is constant and cannot be redefined by the application. For this reason, the bootloader runs in the polling mode with interrupts disabled and use of the interrupts can only be enabled for the application. The reset vector is the only vector used by the bootloader for start-up.

The interrupt handling for the application is handled by virtual interrupt vectors. The real interrupt vector table contains predefined addresses which point to the virtual interrupt vector table(located within the application area). Branching instructions are located within the virtual interrupt vector table, which executes a branch to the real interrupt service routine.

The following figure illustrates an interrupt flow for a watch timer.

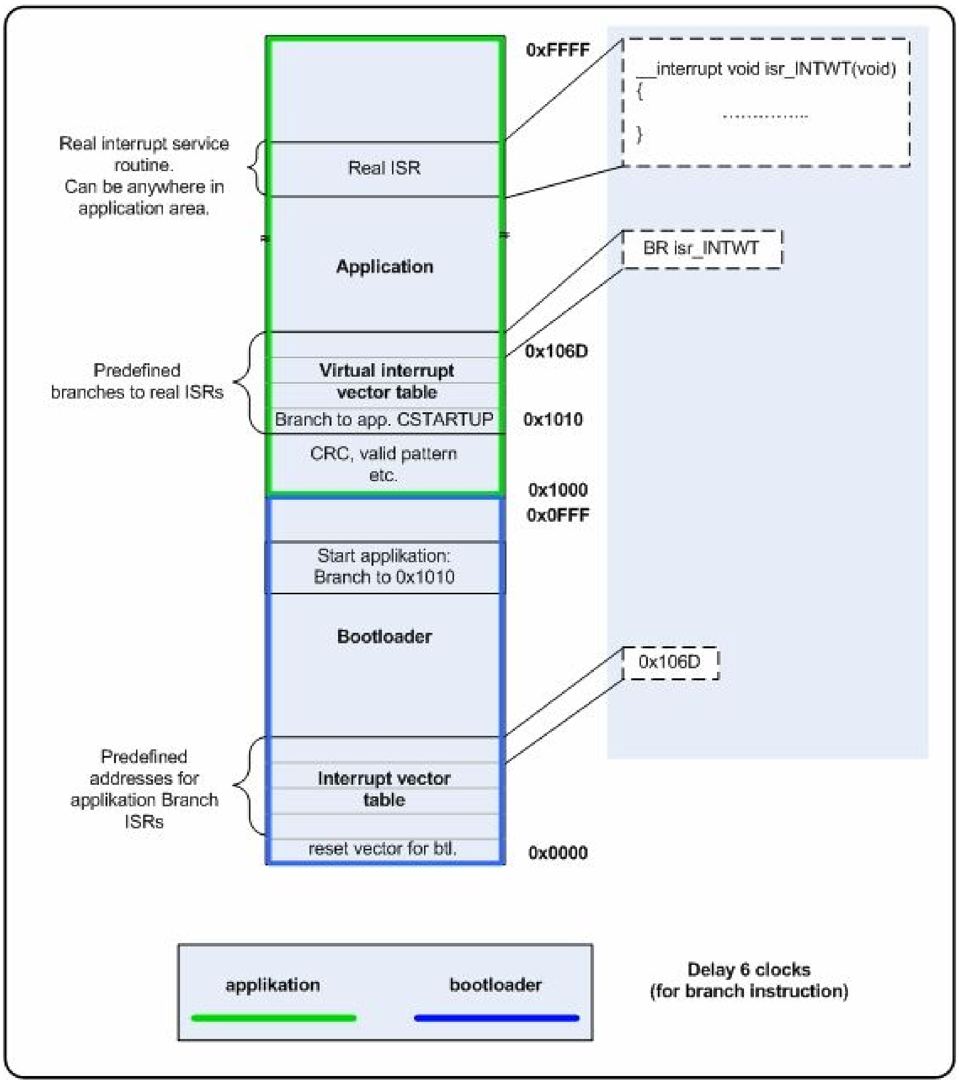

0xFFFF
__interrupt void isr_INTWT(void)
{
……………
}
Real interrupt service routine. Can be anywhere in application area.
Real ISR
Application
BR isr_INTWT
0x106D
Predefined branches to real ISRs
Virtual interrupt vector table
Branch to app. CSTARTUP
0x1010
CRC, valid pattern etc.
0x1000
0x0FFF
Start applikation: Branch to 0x1010
Bootloader
0x106D
Predefined addresses for applikation Branch ISRs
Interrupt vector table
reset vector for btl.
0x0000
applikation
bootloader
Delay 6 clocks (for branch instruction)

Figure 3-11 Interrupt handling with virtual interrupt vectors

The watch timer interrupt flow will be handled within 5 steps:

1. Interrupt triggered by watch timer
2. PSW(program status word) and PC(program counter) will be saved on the stack
3. PC will be set to the address of watch timer interrupt vector (PC = 0x106D).
4. On the address 0x106D is a branch instruction to the real interrupt service routine(PC = real interrupt service routine) located.
5. Real interrupt service routine is serviced and PC with PSW are restored from the stack.

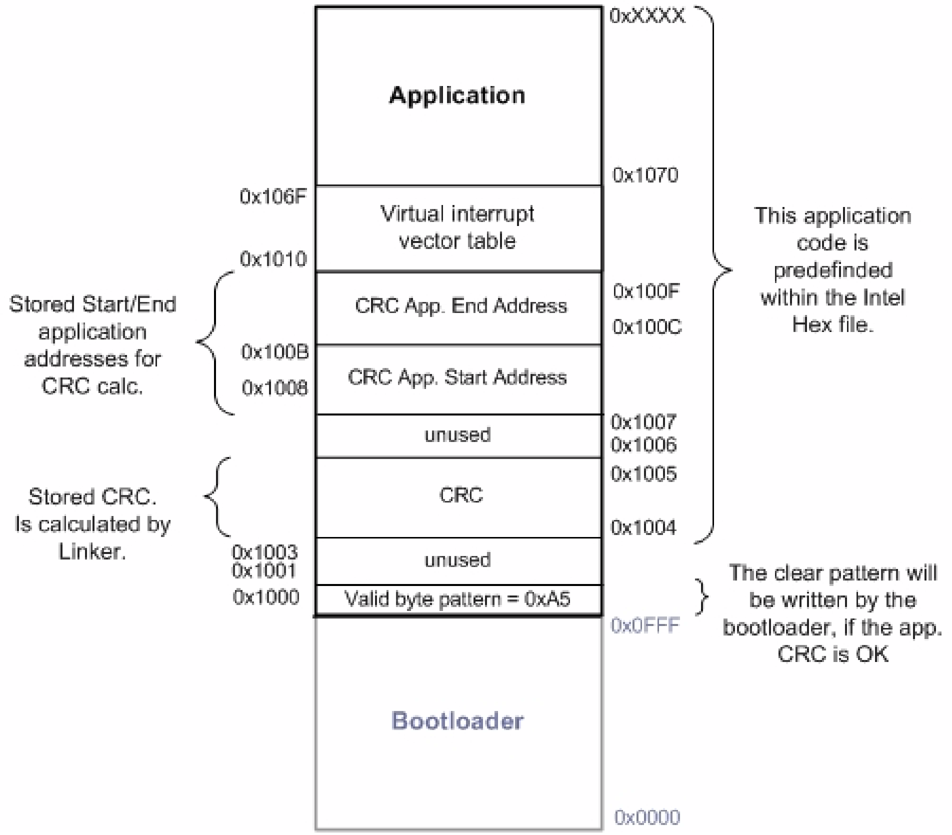
As described, the reset vector points to the bootloader CSTARTUP address. The application virtual reset vector is a branch to the application CSTARTUP, so if the bootloader transfers control to the application, a branch to the application virtual reset vector occurs.

3.3.3 Update methods

The bootloader supports two update methods, which are available for the user.

Application update

This update method enables the user to update their application. Before going into the details of this update method lets look at the application header.



0xXXXX
Application
0x1070
0x106F
Virtual interrupt vector table
This application code is predefinded within the Intel Hex file.
0x1010
Stored Start/End application addresses for CRC calc.
CRC App. End Address
0x100F
0x100C
0x100B
0x1008
CRC App. Start Address
unused
0x1007
0x1006
0x1005
Stored CRC. Is calculated by Linker.
CRC
0x1004
0x1003
0x1001
unused
0x1000
Valid byte pattern = 0xA5
The clear pattern will be written by the bootloader, if the app. CRC is OK
0x0FFF
Bootloader
0x0000

Figure 3-12 Memory structure of the application

The first entry within the application header is the valid byte pattern. To prove the validity of the application, the bootloader verifies this byte and starts the application if valid, or updates this if not valid.

Content of the valid byte	Validity of the application
0xA5	The application is valid and can be started.
other	Invalid application. Update essential

The next entry within the application header is the calculated CRC located on the addresses from 1004H to 1005H. This CRC is calculated by the Linker. The address boundaries for the CRC calculation are stored within the next application header entries "CRC App. Start Address" and "CRC App. End Address". The following figure illustrates a complete application update flow.

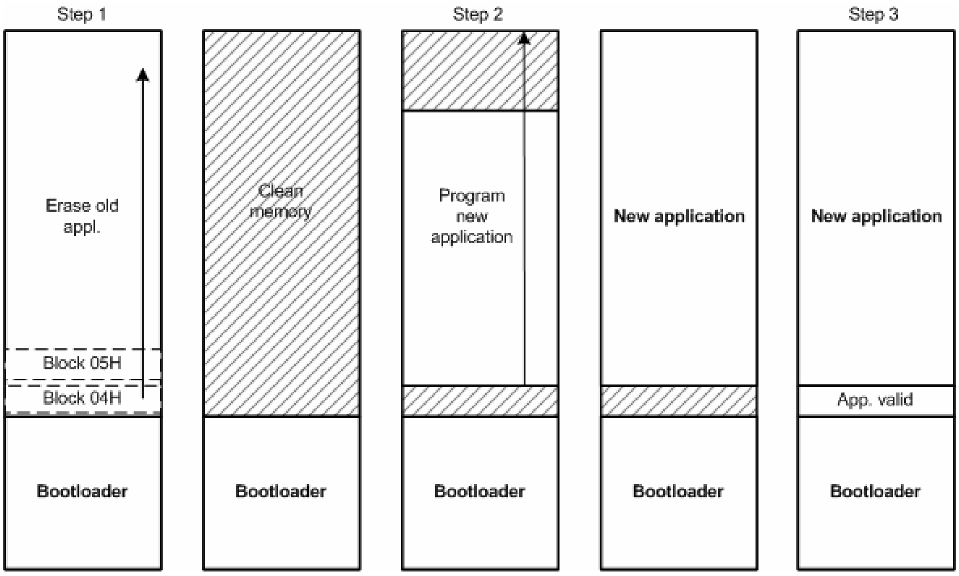


Figure 3-13 Application programming flow

Before a new application can be written into the memory, the full application area must be erased(Step 1). An erasing procedure begins with the lowest block number(Block 04H), due to the fact that the valid byte pattern must be cleared first. The next step within the update flow is to program the new application into the flash memory(Step 2). In this case the received Intel-Hex file will be decoded and the application code will be written into the flash memory. The final procedure is the CRC check of the application(Step 3). The CRC calculation begins from the stored address within the application header "CRC App. Start Address" and end at the address "CRC App. End Address". If the calculated CRC is equal to the stored CRC within the application header the valid byte pattern will be written and after this step the application is valid and can be started.

Bootloader update

This method allows the bootloader code to be updated in asecure way. The new received bootloader code will be written into the boot cluster 1 and the bootloader CRC is stored at the addresses from 0x1FFE to 0x1FFF as showing in the following figure.

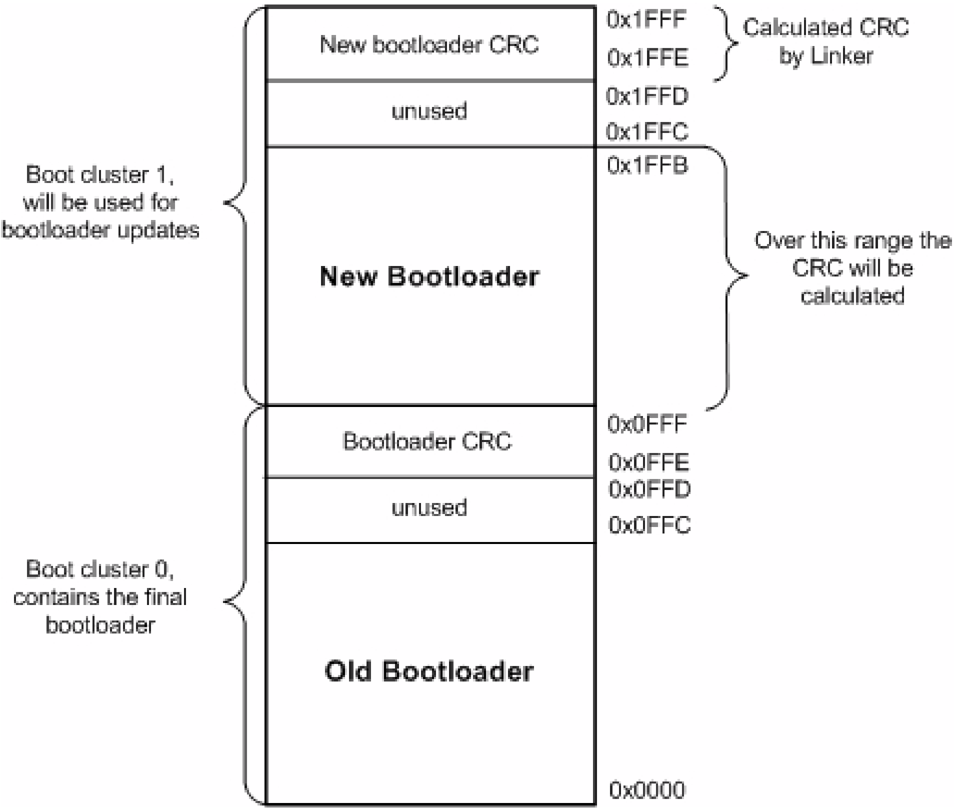


Figure 3-14 Memory structure of the bootloader

In contrast to the application, the bootloader contains always the same address boundaries(from 0x1000 to 0x1FFB) for the CRC calculation. Therefore the Sart- and End-Addresses of the bootloader are not stored. The new received bootloader will be written into the boot cluster 1. The following steps describes a bootloader update flow:

1. **Erasing the application area(boot cluster 1):** The first step during the update is to erase the application area(boot cluster 1). This area will be blank checked and erased if necessary.
2. **Receive the new bootloader and write it into the boot cluster 1:** The received bootloader will be written into the boot cluster 1. The updated memory areas will be internal verified, whether the voltage levels are correct. After successful write cycle, a CRC check will be performed. If all checks was ok, the boot swap flag will be set by the self programming library and a hardware reset will be generated.
3. **Copy new bootloader into the bootcluster 0:** The new bootloader will start-up and check the boot swap flag. If the boot swap flag is set, the bootloader will copy itself to the bootcluster 0. After this copy, the CRC check will be performed. If all checks was ok, the boot swap flag will be reset by the self programming library and a hardware reset will be generated.
4. After the last reset the bootloader was successful updated.

In the following figure are this steps illustrated.

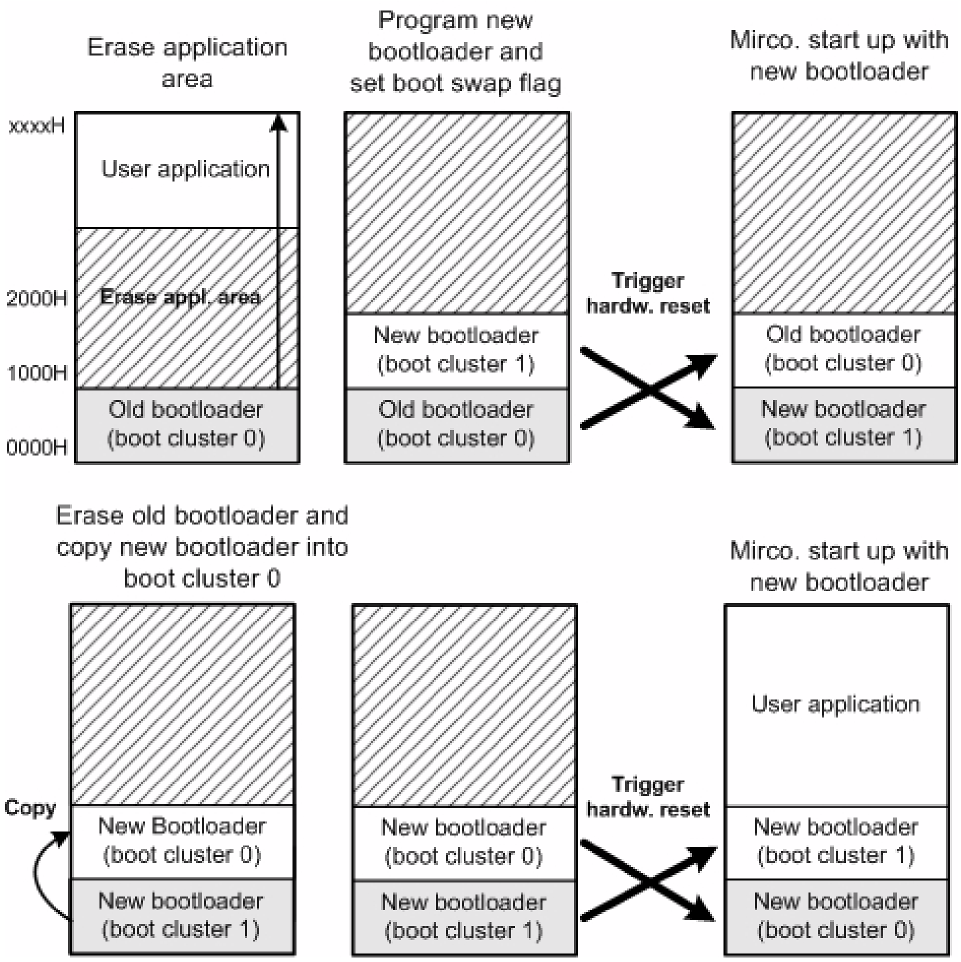

Erase application area
xxxxH
2000H
1000H
0000H
User application
Erase appl. area
Old bootloader (boot cluster 0)
Program new bootloader and set boot swap flag
New bootloader (boot cluster 1)
Old bootloader (boot cluster 0)
Trigger hardw. reset
Mirco. start up with new bootloader
Old bootloader (boot cluster 0)
New bootloader (boot cluster 1)
Erase old bootloader and copy new bootloader into boot cluster 0
Copy
New Bootloader (boot cluster 0)
New bootloader (boot cluster 1)
New bootloader (boot cluster 0)
New bootloader (boot cluster 1)
Trigger hardw. reset
Mirco. start up with new bootloader
User application
New bootloader (boot cluster 1)
New bootloader (boot cluster 0)

Figure 3-15 Bootloader update states

3.4 Software architecture

The software architecture is modular, so modules like communications or file decoders can be exchanged. The following figure shows the general bootloader concept.

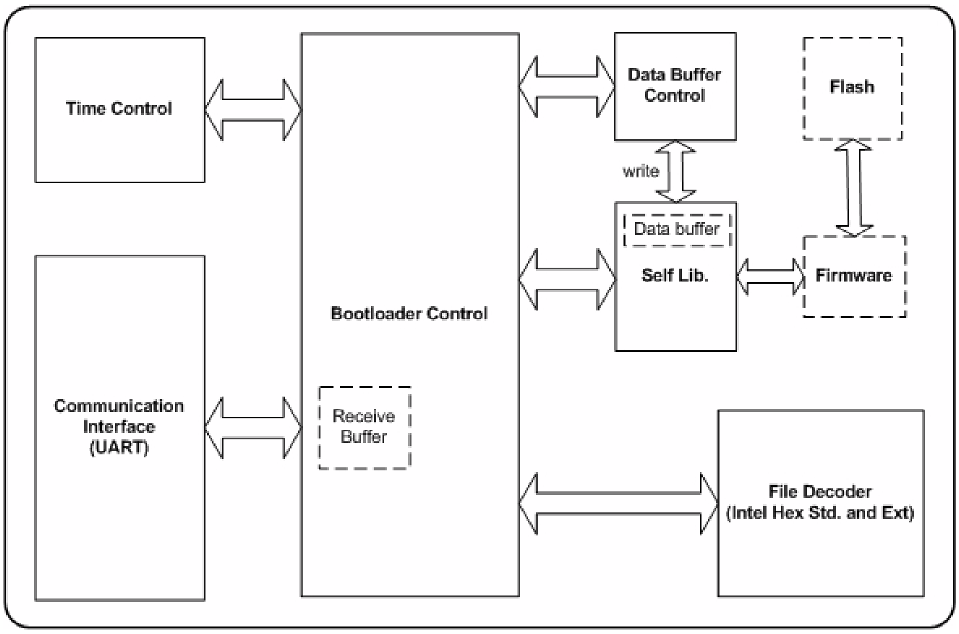


Figure 3-16 Bootloader concept with all modules

The bootloader control is a central connection unit between all modules, which controls all data transfer. The bootloader control operates in the polling mode and all interrupts are disabled. The module interface specifications are located within the chapter "Interface specification".

3.4.1 Time Control module

The bootloader runs with the watchdog disabled, so all critical loops must be controlled by the time control module. The following list shows the cases where timeout detection is used.

- Communication interface send a message
- Communication interface waits for a byte
- Communication interface send a XOFF or XON flow control byte [Note]

The control flow will be checked by the timeout detection, because there can be other interfaces like CAN, where the data transfer is priority controlled. So it can be that the priority is low and the flow control byte can not be sent. In this case the micro will hang. On this reason the flow must be controlled by the time control module.

Note XOFF and XON flow control bytes are used for UART communication. Different flow control signals may be used for other communication interfaces.

3.4.2 Communication Interface module

This module receives and transmits data from/to the host. The serial interface (UART6) can be exchanged by the user (e.g. for CAN). To prevent a receive overflow the interface operates with XON/XOFF protocol. The modular concept allows other flow controls like hardware handshake to be used.

3.4.3 Data Buffer Control module

The self programming library uses a data buffer for data exchange with the firmware.

For example.

The bootloader writes 4 Bytes data into the data buffer and execute the function SelfLib_Write(). This data will be written into flash on a defined address by the firmware.

As you can see in the example above the data buffer must be correctly filled, so the firmware handles the data writes into the flash. The following firmware conditions for write process are controlled by this module:

- **First byte address must be on modulo 4 address**
- **Data buffer content must not overlap over two flash blocks**
- **The byte count within the data buffer must be modulo 4**

3.4.4 File Decoder module

The File Decoder module decodes the received bytes from the update-file. It signals if there are bytes to write into the flash. The following file formats can be decoded:

- Intel-Hex Standard
- Intel-Hex Extended

There are restrictions which must be observed:

1. **The update-file must be sorted by addresses from low to high addresses**
2. **Gaps between addresses must be filled(for CRC calculation).**

3.5 Bootloader implementation

This chapter describes how the bootloader is implemented. The following figure shows the general bootloader flow and the interactions with the user.

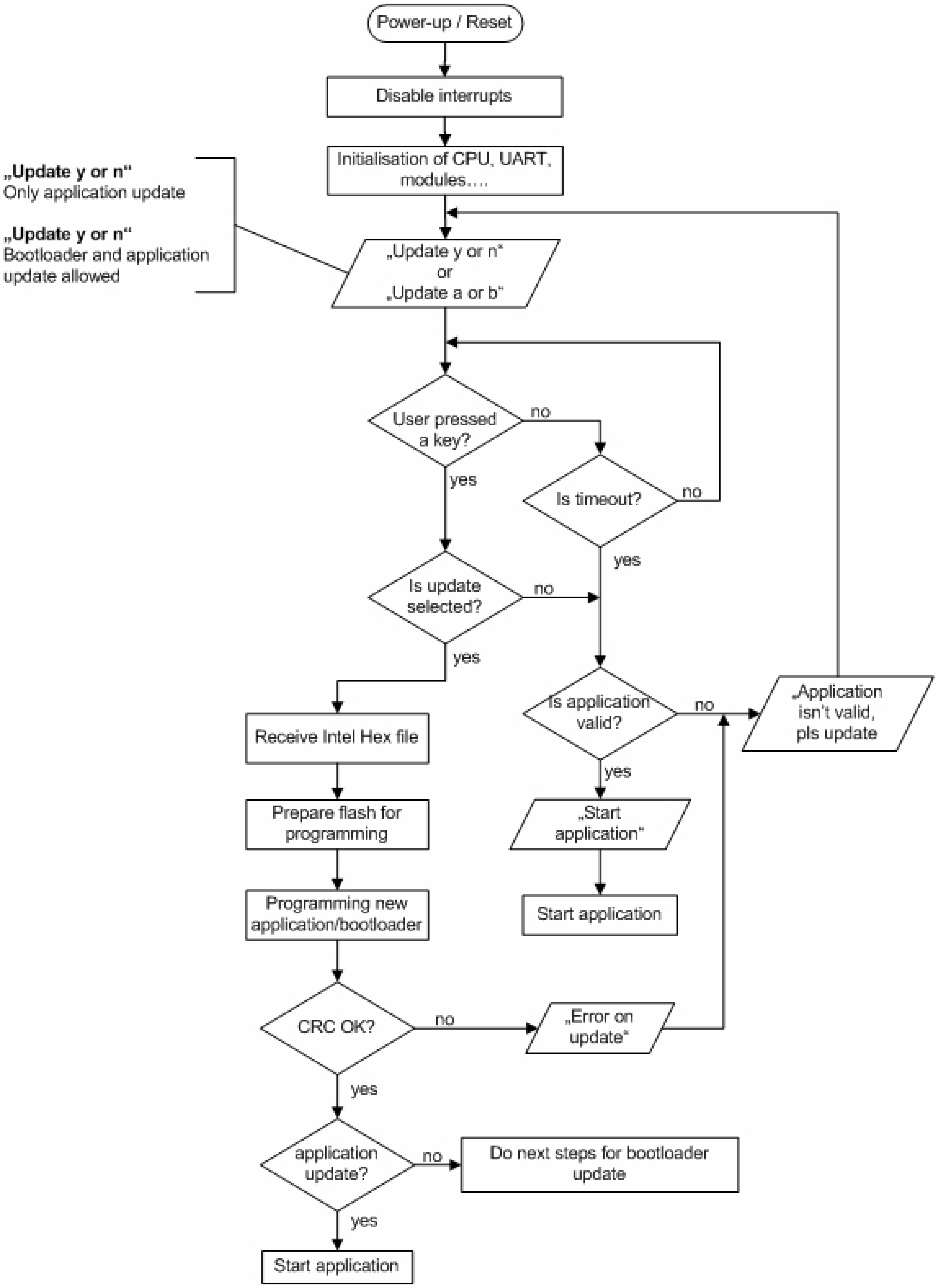


Figure 3-17 General bootloader flow

As you can see in the figure above the user can configure the bootloader for two update flows. The first update("Update y or n") in the figure aonly allows updates to the application. So if the Intel-Hex file contains addresses which are located within the bootloader area, the update will be cancelled. The second update ("Update a or b") flow allows both application and bootloader updates.

3.5.1 Watchdog

The bootloader is implemented without the use of the watchdog(watchdog is stopped). The reason behind this is, because the Self-programming set info functions(e.g. set boot swap flag) needs up to 700µs execution time and the max. overflow time for the watchdog is 500µs(watchdog would reset the microcontroller during set info functions). The watchdog can be stopped by the internal low-speed oscillator. The configuration for the internal low speed oscillator is set by the Option byte to enable stop/start control by software. This allows the usage of the watchdog within the application.

3.5.2 Timeout detection with polling

The bootloader runs without activation of the watchdog, so that all critical loops must be controlled by timeout detection. The following figure illustrates a timeout detection flow.

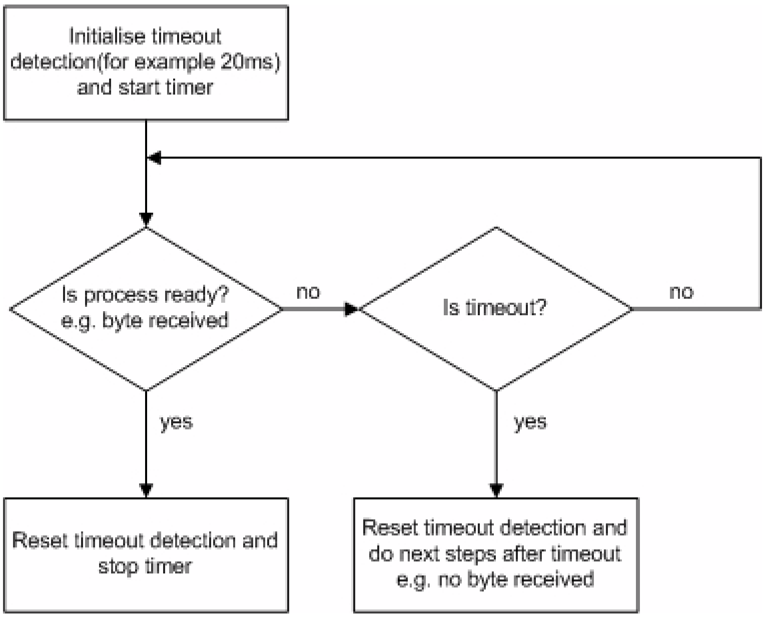

Initialise timeout detection(for example 20ms) and start timer
Is process ready? e.g. byte received
no
Is timeout?
no
yes
yes
Reset timeout detection and stop timer
Reset timeout detection and do next steps after timeout e.g. no byte received

Figure 3-18 General timeout detection flow

3.5.3 Receive flow

The receive flow of the Intel-Hex file will be controlled by the XON/XOFF protocol. During the file transfer the bootloader will receive the bytes until the receive buffer is full. Then the bootloader sends a XOFF byte and receive the last bytes(after XOFF). Therefore the receive buffer is divided into two sections as shown in the following figure:

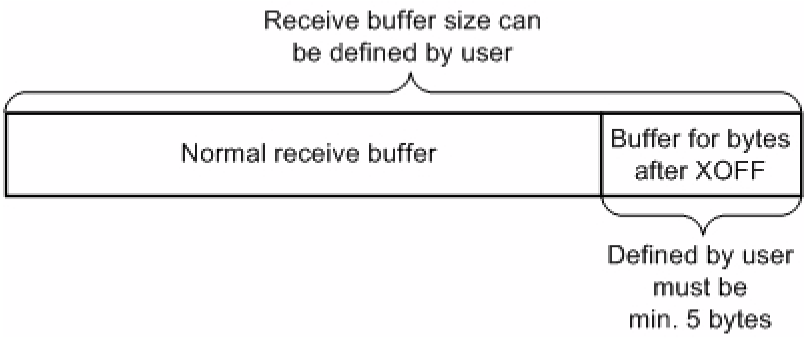

Receive buffer size can be defined by user
Normal receive buffer
Buffer for bytes after XOFF
Defined by user must be min. 5 bytes

Figure 3-19 Buffer architecture

The following figure illustrates the complete receive flow.

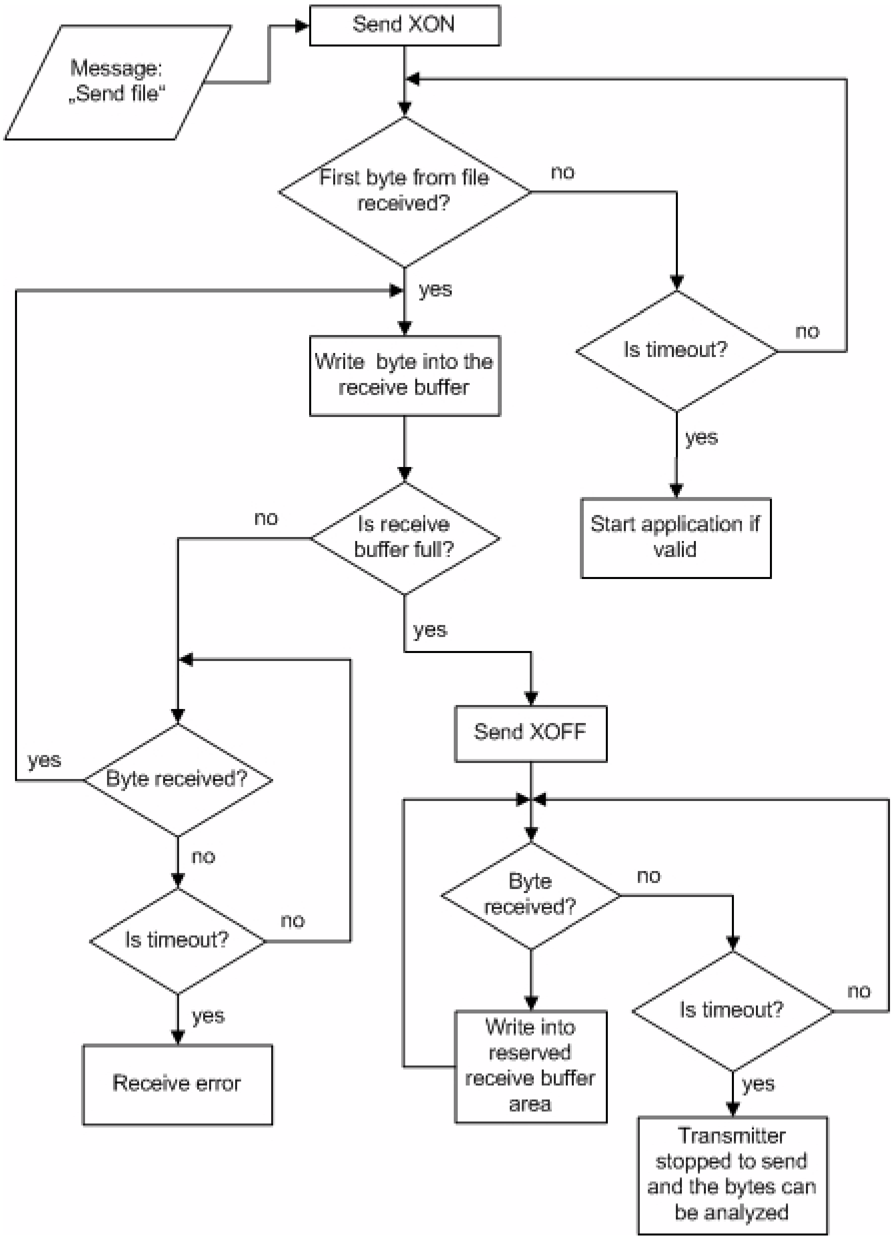

Message: „Send file“
Send XON
First byte from file received?
no
yes
Is timeout?
no
yes
Start application if valid
Write byte into the receive buffer
Is receive buffer full?
no
yes
Send XOFF
yes
Byte received?
no
Is timeout?
no
yes
Receive error
Byte received?
no
Write into reserved receive buffer area
Is timeout?
no
yes
Transmitter stopped to send and the bytes can be analyzed

Figure 3-20 Receive flow controlled by XON/XOFF

3.5.4 Error handler

An error handler is implemented within the bootloader, which will send an error message to the user terminal if an error occurs.

The following table illustrates all defined errors.

Error Code	Error description
01	Application update is selected but the hex file is within the bootloader area. Please check the address ranges of the application.
02	Interface does not send "XOFF". Please check the UART connection.
03	Intel-Hex file error. Please check the Intel-Hex-File.
04	Timeout during hex file receive. Please check the Intel-Hex file whether a EOF tag is written or increase the timeout factor RX_BYTE_TIMEOUT_HEX_2MS.
05	Error during receive. Please check the UART connection.
06	Data buffer write error. Please check the Intel-Hex file whether the addresses are sorted.
07	CRC error on written application. The calculated CRC on the written application is not equal to the stored CRC. Please check the Intel-Hex file whether there are address gaps between the application.
08	CRC error on written bootloader. The calculated CRC on the written bootloader is not equal to the stored CRC. Please check the Intel-Hex file whether there are address gaps between the bootloader.
09	Error on swap flag set. Please check whether all interrupts are disabled and masked out. Check the application area whether it is protected.
10	Error on flash verify. Please check whether all interrupts are disabled and masked out. Check the application area whether it is protected.
11	Error on flash write. Please check whether all interrupts are disabled and masked out. Check the application area whether it is protected.
12	Error on flash erase. Please check whether all interrupts are disabled and masked out. Check the application area whether it is protected.
13	Error on FLMD0 check. Please check the port connection to the FLMD0 pin. Check whether the defined port is correct defined within the header file spl78k0_kx2_user.h (FLMD0_CTRL_PORT_HIGH and FLMD0_CTRL_PORT_LOW).
14	Error on copy boot cluster from 0 to 1. Please check the boot cluster 1 area, whether it is protected and check whether all interrupts are disabled and masked out.
15	Error on swap flag read.
16	The received Intel-Hex file has addresses, which are over the defined application end block. See definition(LAST_APP_BLOCK) within the spl78k0_kx2_user.h file.
17	Address of the new bootloader is bigger than address 0x1000. Please check the Intel-hex-File.
18	Bootloader update not allowed. The bootloader allows only the application update defined within the header file spl78k0_kx2_user.h (BOOTL_UPDATE_ALLOWED). Check the Intel-Hex file whether there are addresses within the bootloader area.
19	User selected a bootloader update, but the Intel Hex file is within the application area. Please check the Intel-Hex file.

3.6 Bootloader configuration

There are three header files for the bootloader configuration as described below.

bl78k0_kx2_user.h

Within this header file are general definitions for the bootloader. Listed below are defines with the functionality:

Definition	Description
BOOTLOADER_VERSION	Bootoader version as string.
BOOTL_UPDATE_ALLOWED	Defines whether the bootloader update is allowed or not. If it is commented out the user can only update the application update(Terminal prompt: Update y or no?).
STOP_RX_WITH_TIMEOUT	General the bootloader send a XOFF control byte if the receive buffer is full. In the case of the UART interface the interface cannot wait until the XOFF byte was send, therefore the XOFF byte will be written into the send buffer and the following bytes will be received(after XOFF). Otherwise the interface would get a receive overrun error. This feature can only be used at interfaces with hardware handshake, were the stop to send signal is very fast.
CLOCK_FREQUENCY	Clock frequency of the used oscillator. Supported are the following frequencies: CLOCK_FREQUENCY = 12 CLOCK_FREQUENCY = 20
IMS_REGISTER_VALUE	Set the value for the IMS register(internal memory size switching register). Within the application the IMS must not be changed.
IXS_REGISTER_VALUE	Set the value for the IXS register(internal expansion RAM size switching register). Within the application the IXS must not be changed.
CODEBANK_BANKS_USER	Count of memory banks. Comment it out, if non banked model is used. E.g. CODEBANK_BANKS_USER = 5
FLASH_END_ADDR	This is an end address of the flash, if the non banked model is chosen. Comment it out, if the banked model is used. E.g. For (64KByte non banked) FLASH_END_ADDR = 0xEFFF
FILL_BYTE	This is a byte pattern, which will be written into the data buffer address gaps.
LAST_APP_BLOCK	This is a define for the last application block. The bootloader will erase the flash memory until this block number during update.
BYTES_AFTER_RX_STOP	This is the reserved bytes count for the following bytes after XOFF message.
RX_BUFFER_SIZE	Receive buffer size definition.
STOP_WATCHDOG	This is a macro to stop a watchdog(the internal low-speed oscillator will be stopped). Change this macro only if the watchdog will be stopped by other method.
START_WATCHDOG	This is a macro to start a watchdog(the internal low-speed oscillator will be started). Change this macro only if the watchdog will be stopped by other method.

The following two tables contains time factors(K x 2ms or K x 50ms) for timeout detection within the bootloader flow.

Definition (2ms factors)	Description
RX_BYTE_TIMEOUT_HEX_2MS	This factor is for the timeout detection during file transfer. For example: 70 * 2ms = 140ms. Timeout will be occurred, if the bootloader does not receive any byte within 140ms.
READY_TO_RX_TIMEOUT	This factor is for the timeout detection during XON send (2 x 2ms = 4ms).
STOP_RX_TIMEOUT	This factor is for the timeout detection during XOFF send (2 x 2ms = 4ms).

Definition (50ms factors)	Description
ILLEGAL_BYTE_TIMEOUT	At the start-up the bootloader send a XON byte to the host. After this the bootloader wait for this timeout (2 x 50ms = 100ms) whether a file transfer is active. If a byte is received during 100ms the bootloader wait on illegal bytes(see ILLEGAL_SEND_TIMEOUT).
ILLEGAL_SEND_TIMEOUT	If an illegal file transfer is detected(see ILLEGAL_BYTE_TIMEOUT) the bootloader will receive illegal bytes until the file transfer has finished(no byte may received until 14 x 50ms = 700ms).
MESSAGE_TIMEOUT_K_50MS	Within this timeout the interface must send a message otherwise a timeout will be occurred(40 x 50ms = 2s).
RECEIVE_TIMEOUT_K_50MS	This timeout factor(60 x 50ms = 3s) is for the user prompt ('y' or 'n'). If the user does not press a key a timeout will be occurred(application will be started).
FIRST_BYTE_HEX_TIMEOUT	This timeout factor(200 x 50ms = 10s) is for the wait loop on the Intel-Hex file.

bl78k0_kx2_uart.h

Within this header file are definitions for the for the UART.

Definition	Description
BAUDRATE	This is a define for the Baud rate. Supported are following baud rates: Baud rate = 115200 Baud rate = 57600

spl78k0_kx2_user.h

Within this header file are definitions for self-programming.

Definition	Description
DATA_BUFFER_SIZE	This is a definition buffer size definition, which will be used for data exchange between firmware and bootloader.
FLMD0_CTRL_PORT_HIGH	Definition of the port, which will pull-up the FLMD0 pin.
FLMD0_CTRL_PORT_LOW	Definition of the port, which will pull-down the FLMD0 pin.

bl78k0_kx2_main.c

The OPTION BYTE must be configured for application requirements(e.g. the watchdog timer interval).

Attention: The Internal low-speed oscillator must be configured in the following mode: Can be stopped by software.

Example:

```
#pragma location = "OPTBYTE"
__root const u08 opbyte[]={0x7E,0x00,0x00,0x00,0x00};
```

3.6.1 Bootloader adaptation

The XCL and header files of the bootloader are configured for the µPD78F0547 microcontroller. The following entries within the XCL and header files must be adapted by the user for other microcontroller of the 78K0/Kx2 family.

bl78k0_kx2_user.h

1. Set the values for the IMS(IMS_REGISTER_VALUE) and IXS (IXS_REGISTER_VALUE) registers. Within the application the IMS and IXS registers must not be changed.
2. For the banked memory model the user must set the banks count or comment it out, if non banked model will be used. E.g. CODEBANK_BANKS_USER = 6
3. If the non banked model will be used, the user must set the flash end address(FLASH_END_ADDR). E.g. For 64KByte flash memory FLASH_END_ADDR=0xEFFF

XCL-File

Following segments must be added for the self-programming environment:

Location of the bootloader:
-Z(CODE)BCLUST0=0086-0FFF

Location of the register bank 3:
-Z(DATA)RB3REGS=FEE0-FEE7

Location of the register bank 2:
-Z(DATA)RB2REGS=FEE8-FEEF

Location of the register bank 1:
-Z(DATA)RB1REGS=FEF0-FEF7

Location of the register bank 0:
-Z(DATA)RB0REGS=FEF8-FEFF

Location of the data buffer:
-Z(DATA)DS_DBF=FB00-FC00

Location of the work area for the self-programming environment:
-Z(DATA)DS_ERAM=FE20-FE83

The CALLF segment can be commented out, because the bootloader does not uses this feature:

```
//-Z(CODE)FCODE=0800-0FFF
```

The following code segments must be located within the address range of 0086 to 0FFB. Because the addresses 0080 to 0084 are for the option byte and 0FFC-0FFF are for the bootloader CRC:

The Start-up, Runtime-library, Non banked and Interrupt functions code segment:

```
-Z(CODE)RCODE,CODE=0086-0FFB
```

Data initialize segments:

```
-Z(CODE)NEAR_ID,SADDR_ID,DIFUNCT=0086-0FFB
```

Location for constants and switch table:

```
-Z(CODE)CONST,SWITCH=0086-0FFB
```

The short address data segments must be located from the address FE84, because the DS_ERAM segment of the self-programming environment is located up to the address FE83:

```
-Z(DATA)SADDR_I,SADDR_Z,SADDR_N,WRKSEG=FE84-FEDF
```

Following segments must be added for address translation and CRC calculation:

This entry will fill the unused bootloader code within the address range of 0000 to 0FFB with the 0xFF pattern:

```
-h(CODE)0-0FFB
-HFF
```

The following segment is the location for the 2Byte CRC:

```
-Z(CODE)CHECKSUM=0FFE-0FFF
```

The following entry will calculate a 2Byte CRC over the code range of 0000 to 0FFB:

```
-J2,crc16,,,,1,0=(CODE)0-0FFB
```

Chapter 4 Application adaption

This chapter describes how the user can configure the application for use with the bootloader.

4.1 Modify XCL-file

First of all the XCL file must be adapted by the user as described below.

Define two code segments(START_ADDR, END_ADDR), which will contain the start and the end address boundaries for the CRC calculation.

```
-Z(CODE)START_ADDR=1008-100B
-Z(CODE)END_ADDR=100C-100F
```

As described before, the application can not uses CALLT, CALLF... features. Delete the following listed segments from the XCL file:

```
-Z(CODE)INTVEC=0000-003F
-Z(CODE)CLTVEC=0040-007D
-Z(CODE)OPTBYTE=0080-0081
-Z(CODE)SECUID=0084-008E
-Z(CODE)FCODE=0800-0FFF
```

The virtual interrupt vector table must be located within the application area, so that a the following segment must be defined:

```
-Z(CODE)VINTVEC=1010-106F
```

For the CRC calculation the application area must be filled with a byte pattern(-HFF -> 0xFF). The -h option defines the address area, which must be filled. The begin(e.g. 1010) and the end(e.g. 1FFF) addresses must be adapted to the application.

```
-h(CODE)1010-1FFF // 0x1010 until application end
```

```
-HFF
```

If the application is implemented with banking usage, the banks must be filled separate.

For example:

```
-h(CODE)01010-7FFF
-h(CODE)08000-0BFFF
-h(CODE)18000-1BFFF
-h(CODE)28000-2BFFF
-HFF
```

The following segment defines the location for the calculated CRC of the Linker (this segment must be added).

```
-Z(CODE)CHECKSUM=1004-1005
```

The following option defines the CRC calculation. The address ranges must be equal to the address ranges of the fill option(see above –h option).

```
-J2,crc16,,,,1,0=(CODE)1010-1FFF
```

Change the start addresses of other code segments to ensure that they are over the end address of the VINTVEC segment.

For example:

```
OLD: -Z(CODE)RCODE,CODE=0257-7FFF
NEW: -Z(CODE)RCODE,CODE=1070-7FFF
```

If the bootloader is compiled for the banking mode, the addresses must be translated. This will be done by the following options:

```
-M18000-1BFFF=0C000-0FFFF
-M28000-2BFFF=10000-13FFF
-M38000-3BFFF=14000-17FFF
-M48000-4BFFF=18000-1BFFF
-M58000-5BFFF=1C000-1FFFF
```

4.2 Add/configure predefined files

There are three predefined files, which must be added to the application project:

- cstartup.s26
- virtual_irq_table.asm
- app_bootl_def.h

virtual_irq_table.asm

This file contains predefined branches to the interrupt service routines and must not be changed by the user.

cstartup.s26

This file will replacing the standard cstartup file. The difference between this file and the standard file is that the address for the reset interrupt vector will not be written(because it would be on the address 0x0000, where the bootloader is located). Change this file only if it is necessary for the application.

app_bootl_def.h

This file must be adapted by the user for the application.

1. Modify the start and the end addresses for the application. These addresses must agree with addresses defined for CRC calculation within the XCL file (-J2,crc16,,,,1,0=(CODE)1010-1FFF and -h(CODE)1010-1FFF).

```
#define START_ADDR_APP 0x1010

#define END_ADDR_APP 0x1FFF
```

2. Comment out interrupt services, which are not used by the application.

```
//#define INTCK2_isr_used
//#define INTLVI_isr_used
#define INTP0_isr_used
#define INTP1_isr_used
//#define INTP2_isr_used
...........
```

If an interrupt service is defined(e.g. #define INTP0_isr_used) the interrupt service routine will be written as follows:

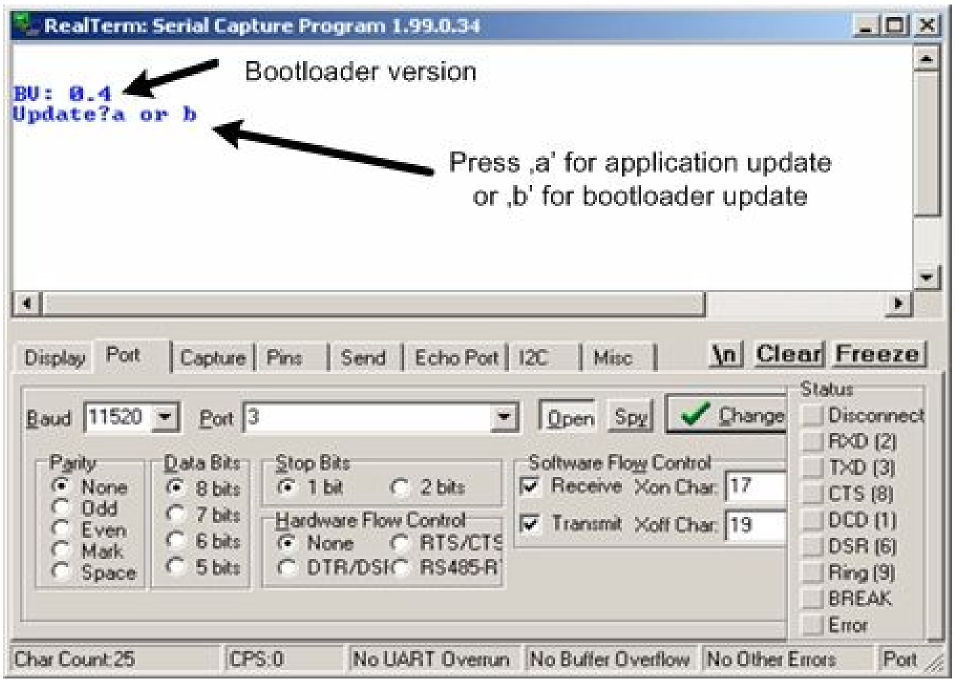
```
__interrupt void INTP0_isr(void)
{
............
}
```

The pragma directive(#pragma vector = INTP0_vect) before the interrupt service routine must be erased.

If an interrupt service is commented out, it will get an RETI instruction. This prevents an illegal branch, for example if the interrupts are not clearly disabled.

Chapter 5 Application example

The following example illustrates a bootloader update.



```
BV: 0.4
Update?a or b
```

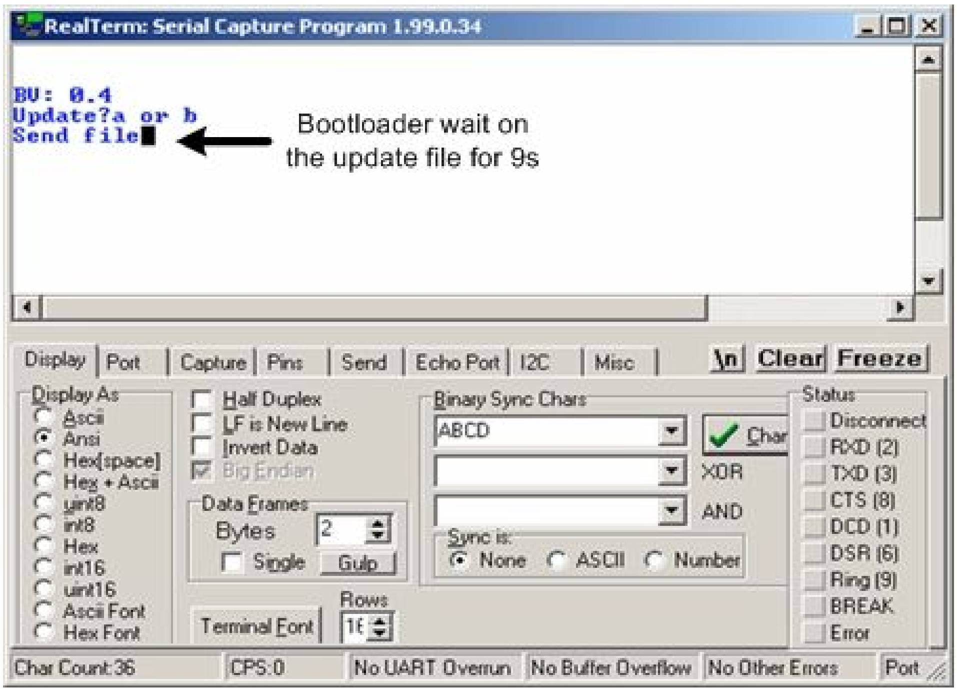
Bootloader version

Press ,a' for application update
or ,b' for bootloader update

Baud 11520
Port 3
Parity: ● None
Data Bits: ● 8 bits
Stop Bits: ● 1 bit
Hardware Flow Control: ● None
Software Flow Control: ☑ Receive Xon Char: 17
☑ Transmit Xoff Char: 19
Char Count:25
CPS:0

Figure 5-21 User prompt for update

As you can see in the figure the bootloader version is 0.4 and the update method allows to update the bootloader and the application. If the user does not press any key, the bootloader would start the application. For example the user typed 'b':



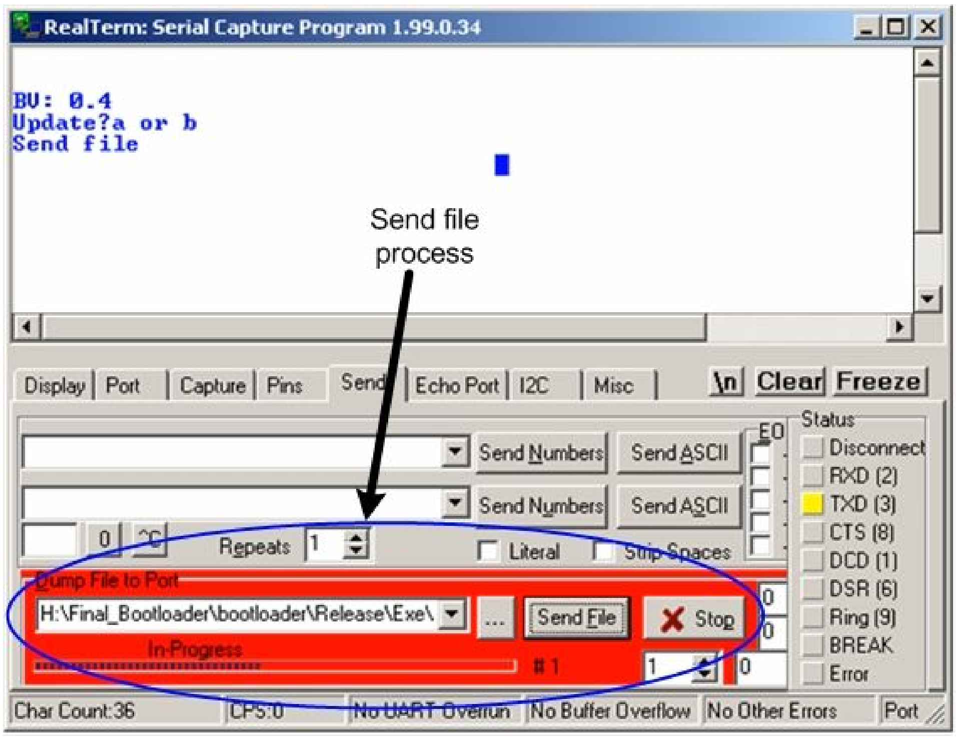
```
BV: 0.4
Update?a or b
Send file█
```

Bootloader wait on
the update file for 9s

Display As: ● Ansi
☐ Half Duplex
☐ LF is New Line
☐ Invert Data
☑ Big Endian
Data Frames Bytes 2
☐ Single
Binary Sync Chars: ABCD
Sync is: ● None
Rows 16
Char Count:36
CPS:0

Figure 5-22 Bootloader waits on the update file

The send file process was triggered by the user. The bootloader does not send any message to the user, that the file transfer was detected.



```
BV: 0.4
Update?a or b
Send file
```

Send file process

Dump File to Port: H:\Final_Bootloader\bootloader\Release\Exe\

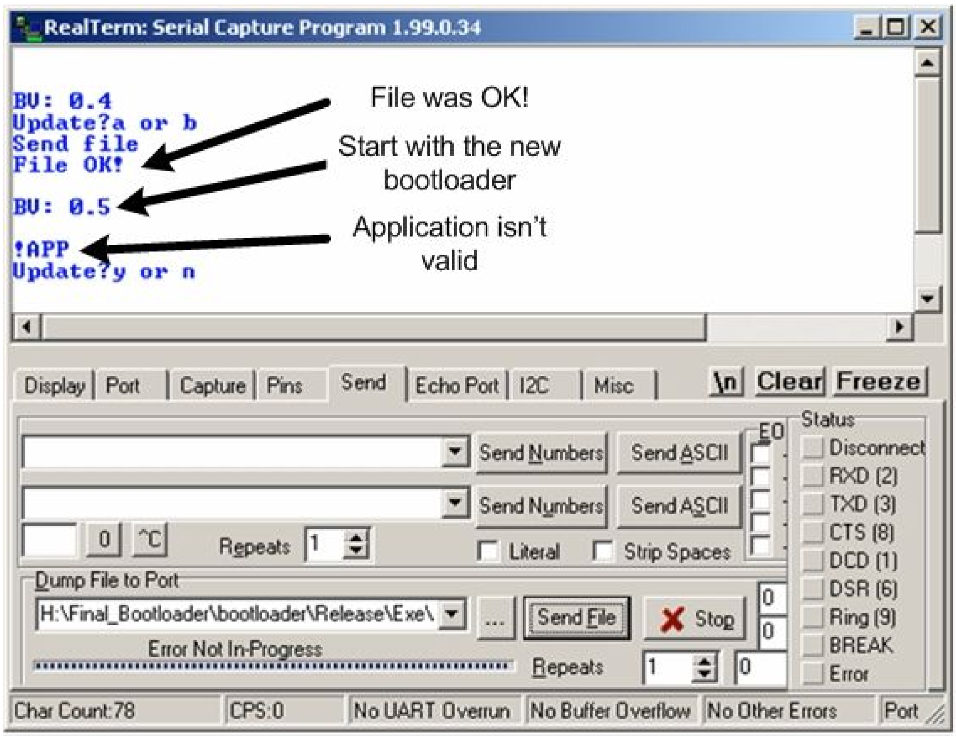
In-Progress

Repeats 1

Char Count:36

Figure 5-23 Triggered file send process

If the Intel-Hex file was successfully transmitted and the CRC check was OK, the bootloader will send a "File OK!" message.



```
BV: 0.4
Update?a or b
Send file
File OK!

BV: 0.5

!APP
Update?y or n
```

File was OK!

Start with the new bootloader

Application isn't valid

Dump File to Port: H:\Final_Bootloader\bootloader\Release\Exe\

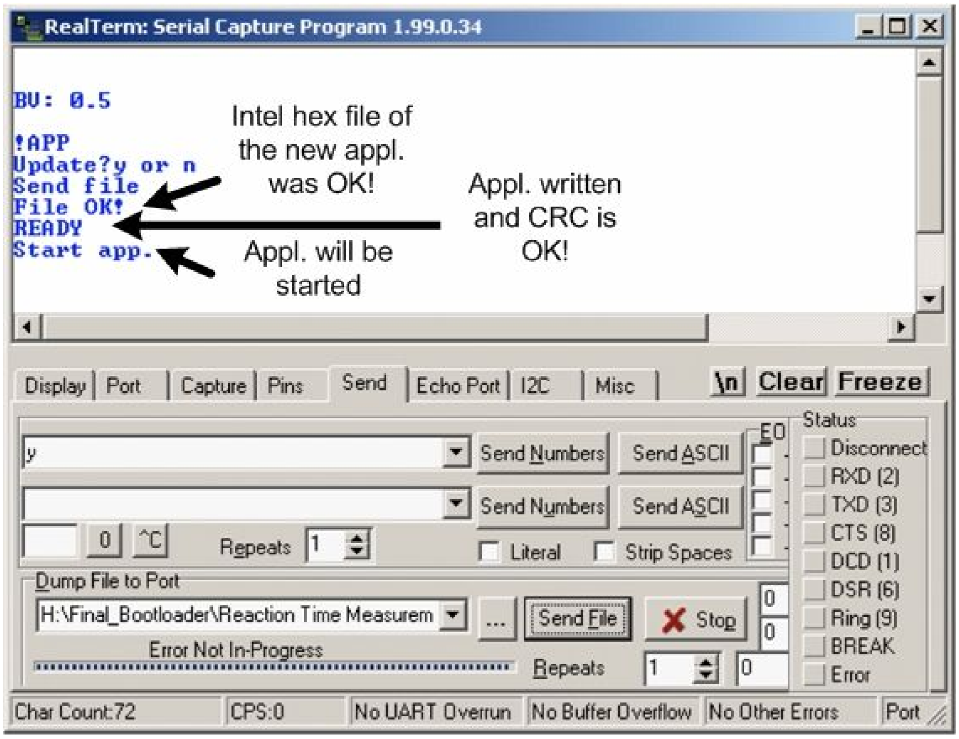
Error Not In-Progress

Repeats 1

Char Count:78

Figure 5-24 Bootloader update has finished

After successful update the bootloader start-up with a new version(BV: 0.5). During the bootloader update the application valid byte pattern will be cleared, so that the application is not valid and sends the "!APP" message. The new bootloader only allows application updates("Update? y or n"). If the new application was written, the bootloader sends the "READY" message and starts the application.



```
BV: 0.5

!APP
Update?y or n
Send file
File OK!
READY
Start app.
```

Intel hex file of the new appl. was OK!

Appl. written and CRC is OK!

Appl. will be started

y

H:\Final_Bootloader\Reaction Time Measurem

Error Not In-Progress

Char Count:72 CPS:0 No UART Overrun No Buffer Overflow No Other Errors

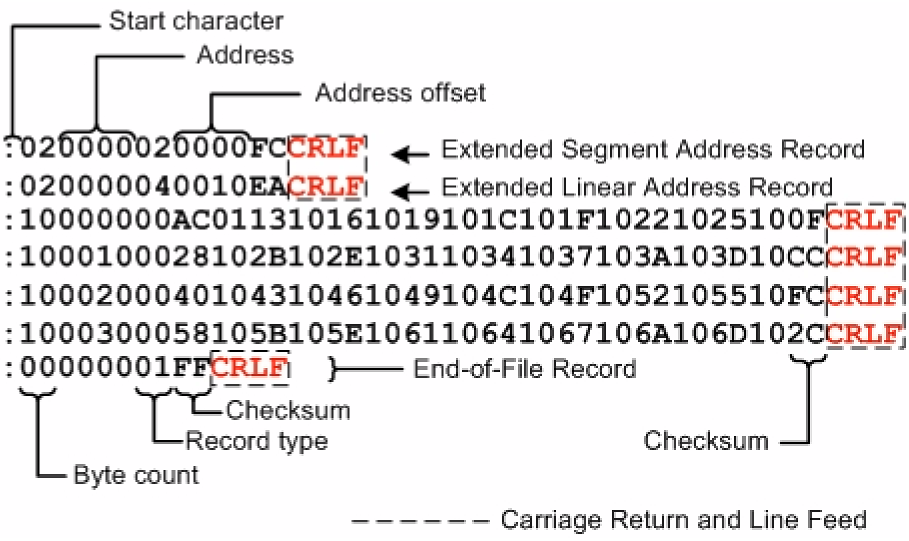
Figure 5-25 Application update and start

Chapter 6 Interface specification

This chapter describes the Intel-Hex file and modules specification.

6.1 Intel Hex File format

Intel-Hex File consists of records which has a header of 9-character (4-field) prefix that defines the start of record, byte count, address offset, and record type, and a 2-character checksum suffix. The following figure illustrates some sample records.

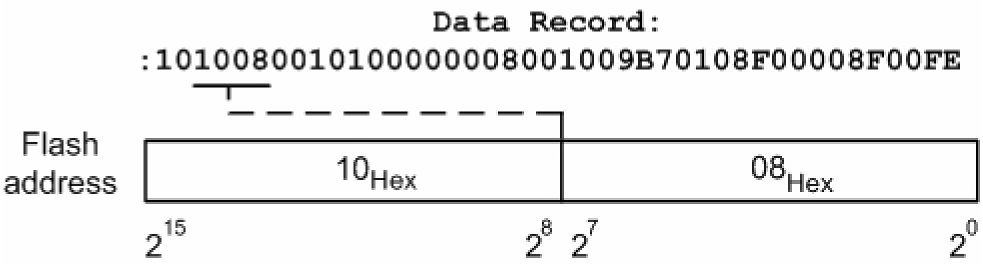


```
:020000020000FCCRLF
:020000040010EACRLF
:10000000AC0113101610191O1C101F10221025100FCRLF
:100010002810 2B102E10311034103710 3A103D10CCCRLF
:10002000401043104610491O4C104F1052105510FCCRLF
:10003000581O5B105E10611064106710 6A106D102CCRLF
:00000001FFCRLF
```

Figure 6-26 Intel-Hex file format

The differences between "Standard-Intel-Hex" and "Extended-Intel-Hex" are the address ranges. Extended Intel-Hex has 32 bit address range and Standard Intel-Hex has 16 Bit address range.

For the address calculation within the Standard Intel-Hex file is only the address field within the header necessary. The following figure shows the address calculation within the Standard Intel-Hex file:



```
Data Record:
:10100800101000000008001009B70108F00008F00FE
```

Figure 6-27 Standard-Intel-Hex file addressing

For 32Bit addressing within the Extended-Intel-Hex file are two additively records essential(Linear Address offset and Segment address offset):

```
:02 0000 04 0010 EA          Linear Address Offset
:02 0000 02 1230 BA          Segment Address Offset
:10 0045 00 55AA FF ..... BC        Data Record
```

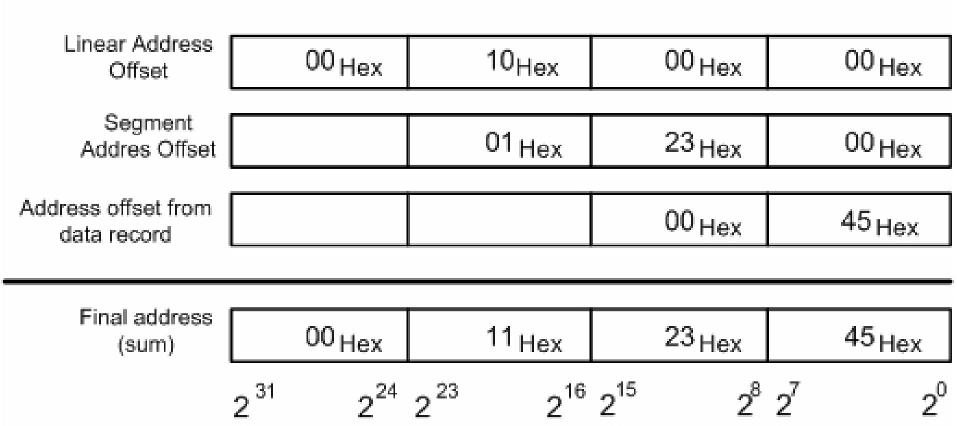


Figure 6-28 Extended-Intel-Hex file addressing

There are six record types defined for the Intel-Hex file format. Every record will be secured by a checksum, which is a two's complement of the fields byte count, address, record type and data bytes.

00-Data Record

The data record contains data to write into the flash. The data count will be defined by the byte count field and the address for the first byte by the address field(see above for address calculation).

Start character	Byte count	Address	Record type	Data	Checksum
1 Byte ':'	1 Byte	2 Byte	1 Byte '00'	xx Byte count	1 Byte

01-End of File Record

The end-of-file record represents the end of an Intel-Hex file.

Start character	Byte count	Address	Record type	Checksum
1 Byte ':'	1 Byte '00'	2 Byte '0000'	1 Byte '01'	1 Byte 'FF'

02-Extended Segment Address Record

This record type defines bits 4 to 19 of the segment address(see above).

Start character	Byte count	Address	Record type	Data	Checksum
1 Byte ':'	1 Byte '02'	2 Byte '0000'	1 Byte '02'	2 Byte 'ext. addr.'	1 Byte

03-Start Segment Address Record

This record, which specifies bits 4-19 of the execution start address will be ignored by the bootloader.

Start character	Byte count	Address	Record type	Data	Checksum
1 Byte ':'	1 Byte '04'	2 Byte '0000'	1 Byte '03'	4 Byte 'addr.'	1 Byte

04-Extended Linear Address Record

This record specifies bits 16-31 of the destination address(see above).

Start character	Byte count	Address	Record type	Data	Checksum
1 Byte ':'	1 Byte '02'	2 Byte '0000'	1 Byte '04'	2 Byte 'ext. lin. addr.'	1 Byte

05-Start Linear Address Record

This record, which specifies bits 16-31 of the execution start address will be ignored by the bootloader.

Start character	Byte count	Address	Record type	Data	Checksum
1 Byte ':'	1 Byte '04'	2 Byte '0000'	1 Byte '05'	4 Byte 'addr.'	1 Byte

6.2 Interface specification

This chapter describes the interface specification. The C files can be exchanged by the user and only the function names must be equal(the bootloader binds the function with **extern** keyword).

6.2.1 Communication interface

void ifaceInit(void)

Description:

This function initializes the communication interface.

Precondition:

None

Postcondition:

The communication interface is initialized and can be used.

Parameter:

None

Return:

None

void readyToRx(void)

Description:

This function signals the host that it is ready to receive(Uart: Xon message(11 hex) will be send to the host).

Precondition:

None

Postcondition:

The host is informed that the bootloader is ready to receive. The bootloader must analyze with the readyToRxEnabled function, whether the host has acknowledged.

Parameter:

None

Return:

None

u08 readyToRxEnabled(void)

Description:

This function checks the host reaction, whether an acknowledge is received(after readyToRx).

Precondition:

The function readyToRx was called.

Postcondition:

The host has acknowledged or not acknowledged. If the host has acknowledged, all acknowledge flags must be erased.

Parameter:

none

Return:

- 0_{HEX} Host has not acknowledged.
- 1_{HEX} Host has acknowledged.

void stopRx(void)

Description:

This function signals the host that it must stop the file transfer(Uart: Xoff message (13 hex) will be send to the host).

Precondition:

None

Postcondition:

The host is informed that the bootloader is not ready to receive. The bootloader must analyze with the stopRxEnabled function, whether the host has acknowledged.

Parameter:

None

Return:

None

u08 stopRxEnabled(void)

Description:

This function checks the host reaction, whether an acknowledge is received(after stopRx).

Precondition:

The stopRx function was called.

Postcondition:

The host has acknowledged or not yet. If the host has acknowledged, all acknowledge flags must be erased.

Parameter:

None

Return:

- 0_{HEX} Host has not acknowledged.
- 1_{HEX} Host has acknowledged.

void sendByte(u08 txData)

Description:

This function write a byte into the transmit buffer.

Precondition:

The interface status was checked by the txStatus, that it is not busy.

Postcondition:

The byte is written into the transmit buffer.

Parameter:

- 1Byte - This byte will be written into the transmit buffer.

Return:

None

u08 txStatus(void)

Description:

This function check the interface status, whether it is busy or not.

Precondition:

A byte was written into the transmit buffer by the sendByte function.

Postcondition:

If the byte was transmitted, all transmit ready flags must be erased(e.g. tx interrupt request bit).

Parameter:

None

Return:

- 0_{Hex} Transmit interface is busy
- 1_{Hex} Transmit interface is ready-to-transmit

__callt u08 byteReceived(void)

Description:

The function checks the receive status.

Precondition:

The host was informed by the readyToRx function, that it is receive ready.

Postcondition:

If a byte was received, all receive flags must be erased.

Parameter:

None

Return:

- 00_{Hex}: Nothing received
- 01_{Hex}: Byte received

__callt u08 rxError(void)

Description:

This function checks the receiver, whether an error occurs during receive.

Precondition:

Byte was received(see byteReceived).

Postcondition:

If an error is occurred, all error flags must be cleared.

Parameter:

None

Return:

- 00_{Hex}: Error is occurred during receive routine.
- 01_{Hex}: No error.

__callt u08 getRxByte(void)

Description:

This function return the received byte.

Precondition:

Byte was received(see byteReceived()) and no error occurred during receive routine(see rxError()).

Postcondition:

Received byte was read from the receive buffer.

Parameter:

None

Return:

- 1 Byte : Received byte.

void resetRxErrorFlags(void)

Description:

This function reset all receive error flags of the receive interface.

Precondition:

None

Postcondition:

The receive interface is receive ready.

Parameter:

None

Return:

None

void resetInterface(void)

Description:

This function reset the communication interface.

Precondition:

None

Postcondition:

None

Parameter:

None

Return:

None

6.2.2 File decoder interface

__callt u08 decodeReceivedBytes(u08 rx_byte)

Description:

This function decode the received byte.

Precondition:

None

Postcondition:

None

Parameter:

- 1 Byte: Received byte from file, which must be checked.

Return:

- 01_{Hex}: File error
- 02_{Hex}: Bytes is checked, but do not write it into the data buffer
- 03_{Hex}: Write the checked byte into the data buffer

__callt void resetFileDecoder(void)

Description:

This function reset the file decoder.

Precondition:

None

Postcondition:

None

Parameter:

None

Return:

None

u08 isEOF(void)

Description:

This function checks, whether the file end is reached.

Precondition:

The host do not send any bytes from the file(timeout is occurred).

Postcondition:

None

Parameter:

None

Return:

- 00_{Hex}: Is not file end.
- 01_{Hex}: The file end is reached.

u32 getAddress(void)

Description:

This function return the flash address location for the byte, which must be written into the data buffer.

Precondition:

The decodeReceivedBytes() function returned 03_{Hex}.

Postcondition:

None

Parameter:

None

Return:

- 4Byte - Flash address of the byte, which must be written into the data buffer.

u08 getWriteByte(void)

Description:

This function returns the byte, which must be written into the data buffer.

Precondition:

The decodeReceivedBytes() function returned 03_{Hex}.

Postcondition:

None

Parameter:

None

Return:

- 1Byte - The current byte, which must be written into the data buffer.

6.2.3 Time control interface

void init_timer_50ms(void)

Description:

This function will be called after the microcontroller reset. The timer will be initialized for 50ms interval.

Precondition:

None

Postcondition:

The timer is ready for operation.

Parameter:

None

Return:

None

__callt void setTimerIntervall_2ms(void)

Description:

This function set the timer on the 2ms interval.

Precondition:

Timer was initialized by the init_timer_50ms() function.

Postcondition:

None

Parameter:

None

Return:

None

__callt void setTimerIntervall_50ms(void)

Description:

This function set the timer on the 50ms interval.

Precondition:

Timer was initialized by the init_timer_50ms() function.

Postcondition:

None

Parameter:

None

Return:

None

__callt void initTimeoutDetect(u08 timer_factor)

Description:

This function initialize the timeout detection for a given time(time_factor x 2ms or time_factor x 50ms).

Precondition:

Timer was initialized by the function init_timer_50ms().

Postcondition:

Timeout detection is initialized and the timer is started.

Parameter:

- 1Byte: This is a factor for the timer e.g. if the timer is initialized by the function setTimerIntervall_2ms() and the factor is timer_factor = 10 a timeout will occur after 10 x 2ms = 20ms.

Return:

None

__callt void resetTimeoutDetect(void)

Description:

This function reset the timer and the timeout detection.

Precondition:

None

Postcondition:

Timeout detection can be used for other process.

Parameter:

None

Return:

None

__callt u08 isTimeout(void)

Description:

This function checks the timeout.

Precondition:

The timeout factor was set by the initTimeoutDetect() function.

Postcondition:

None

Parameter:

None

Return:

- 00Hex: No timeout
- 01Hex: Timeout occurred

Revision History

This revision list shows all functional changes compared to the previous manual version EASE-UM-0004-0.2 (date published 16/09/05).

Chapter	Page	Description

Index