NXP Semiconductors
Application Notes

Document Number: AN12218
Rev. 1, 10/2018

S32K1xx Bootloader

by: NXP Semiconductors

Contents

1. Introduction 1
2. Architecture description 1
2.1. Bootloader workflow overview 3
2.2. Communication handling overview 5
3. Building compatible applications 8
4. Using the bootloader 8
4.1. UART interface 9
5. Appendix A 12
5.1. On S32DS: 12
6. Revision History 14

1. Introduction

The following document describes the architecture and usage of the S32K1xx bootloader.

This bootloader supports Universal Asynchronous Receiver/Transmitter (UART) as communication interfaces and can be easily modified to support other kinds of communication interfaces.

2. Architecture description

The bootloader is organized in three layers:

- Bootloader – is incharge of starting the user application and polling for incoming data.
- Communication handling / Memory handling – is incharge of processing the received data and handling the writes to non-volatile memory.
- Microcontroller drivers – is in charge of handling all the low-level communication with the actual peripherals available on the microcontroller.

The following image showcases a diagram of the architecture of the bootloader:

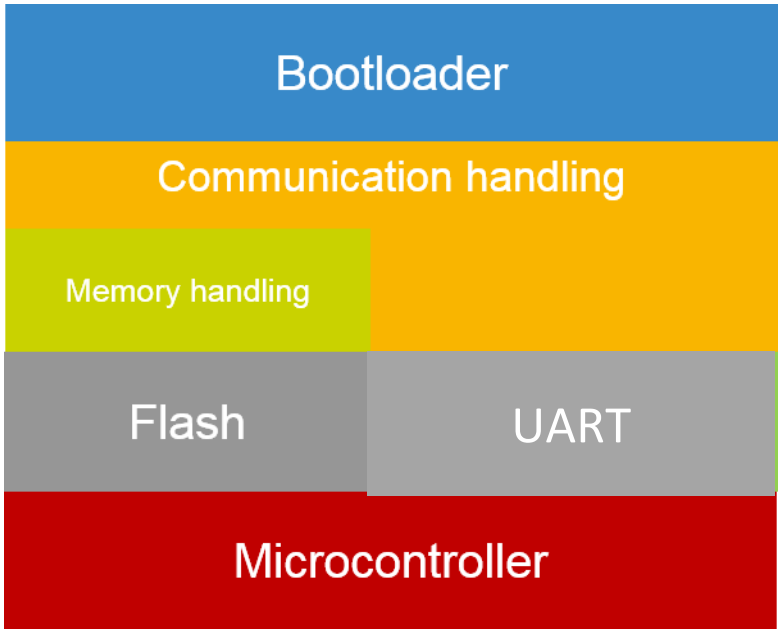


Figure 1. Bootloader architecture

The bootloader is placed in the D-Flash section. The application should be placed in the P-Flash section. The idea behind storing the bootloader into D-Flash section is to allow application to use the whole P-Flash without reserving a section for the bootloader. The following figure showcases the memory layout that the bootloader has and the application must follow.

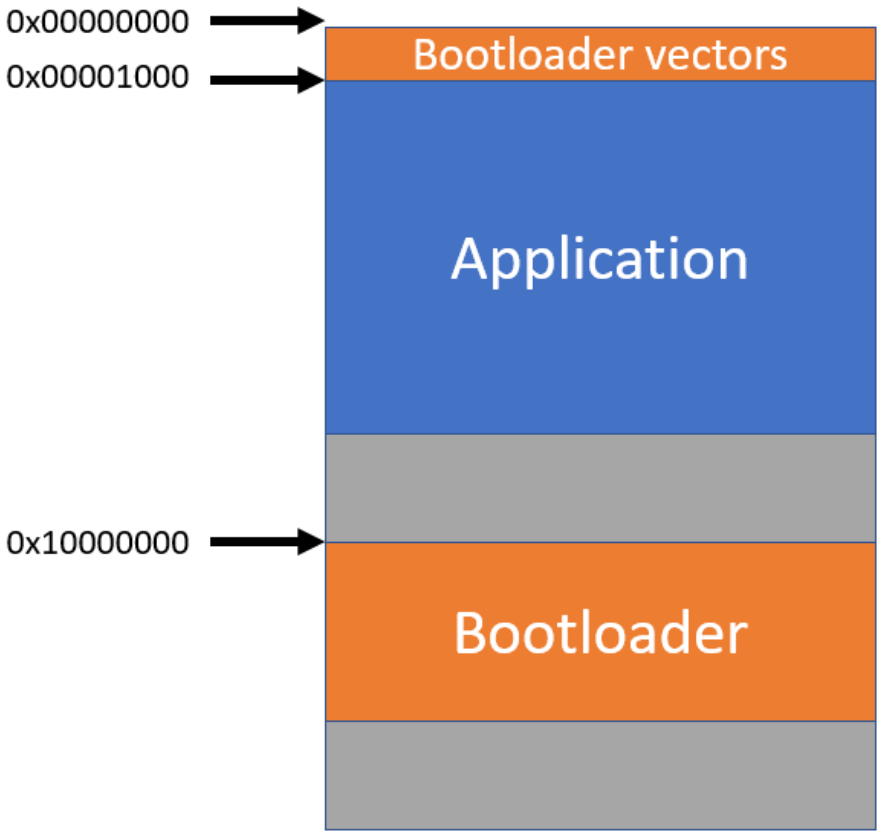


Figure 2. Memory layout

2.1. Bootloader workflow overview

The bootloader workflow can be observed in the image below.

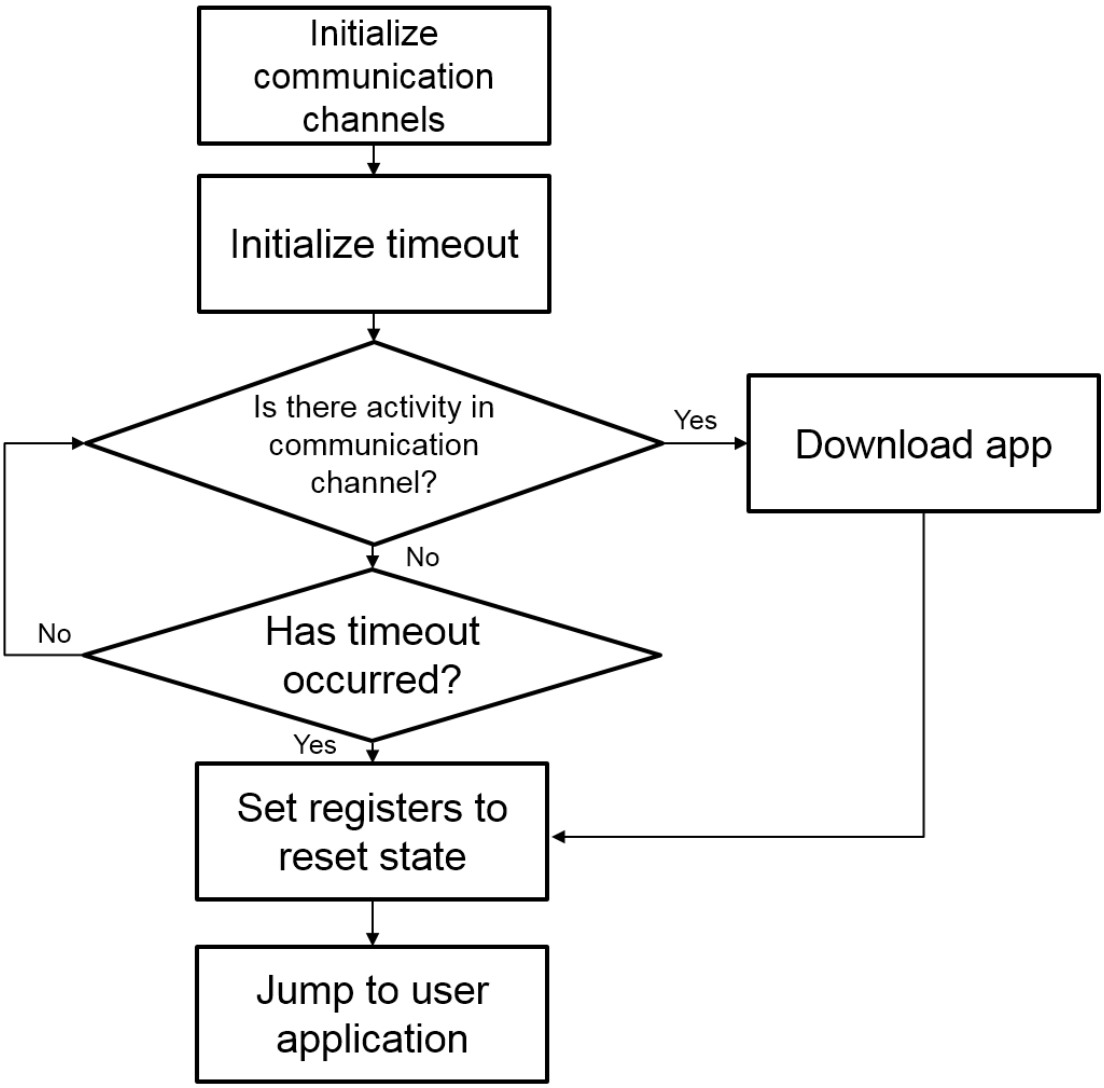


Figure 3. Bootloader workflow

The first step is to initialize the available communication channels, in this instance only is available, but if another communication channel is required its initialization routine should be called here.

To select the communication channel to be used simply modify ‘sources/drivers/inc/comm.h’ in line 11 to select the communication interface to use. Setting the preprocessor directive to 0 disables the communication interface and setting it to 1 enables it.

```
/* Define communication interfaces to use, 0-> Disable 1-> Enable */
#define UART_COMM                1
```

Communication interfaces can be enabled to work simultaneously, but since the bootloader is optimized for size, the bootloader’s linker file would have to be modified to accommodate the generated code. Therefore, it is recommended to use only one kind of communication at a time. If both interfaces are needed, the first one to detect activity in the bus will be used to download the application, by default the bootloader is set to work with UART communication only.

The second step is to initialize the timeout mechanism. After a reset, the microcontroller will poll the selected communication channel, if no activity was detected during the time allowed by the timeout mechanism the device will attempt to execute the last application loaded, if the device hasn't received an application it will get stuck in a loop. In order to attempt the download of an application another reset is required.

The timeout value is configurable and it is set by default to five seconds. Only one second multiples can be selected, in order to change the timeout value simply set the desired value in 'sources/drivers/inc/timeout.h' line 14.

```
/* Define timeout value, the base is 1s */
#define TIMEOUT_VAL             5
```

Once the timeout mechanism has been initialized the device starts polling for activity in the communication channel for the time allotted by the timeout value. If activity is detected in the communication channel the bootloader starts downloading the application via the selected communication channel (e.g. UART).

If a timeout occurs or an application is flashed to the device, the bootloader disables and sets all the registers that were modified to its reset state, this step is required to ensure the application starts executing on an environment close to out of reset state.

Once the registers have been set to its reset state the device attempts to jump to the user application.

2.2. Communication handling overview

The first step carried out by the communication handling routine is to obtain an SREC 'phrase' through the selected channel. A phrase is simply a line of the SREC file. Two lines (phrase) of an SREC file can be found below:

```
S00F000068656C6C6F20202020200000 3C
S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C600000386300002 6
```

Record type | Byte count | Address | Data | Checksum

The structure of an SREC line can be seen below:

S	Type	Byte Count	Address	Data	Checksum

The first two characters are sent in ASCII format, 'S' and SREC type (e.g. '0', '1'…'9'), the remaining data is converted to its hexadecimal representation and sent (instead of sending '0' and 'F' 0x0F is sent). For a detailed description of an SREC format please refer to the following webpage.

The phrase is received and stored in the following structure:

```
typedef union
{
	uint8_t Byte[MAX_PHSIZE_BP];		/* Byte level access to the Phrase */
	struct
	{
		char PhraseType;	/* Type of received record (e.g. S0, S1, S5, S9...) */
		uint8_t PhraseSize; /* Phrase size (address + data + checksum) */
		/* Address, depending on the type of record it might vary */
		uint8_t PhraseAddress[MAX_ADDRESS_BP]__attribute__ ((aligned (32)));
		/* Maximum 32 data bytes */
		uint8_t PhraseData[MAX_DATA_BP]__attribute__ ((aligned (32)));
		uint8_t PhraseCRC;  /* Checksum of size + address + data */
	}F;

}BootPhraseStruct;
```

This structure holds all the information provided by the SREC phrase, such as record type, byte count, address, data and cyclic redundancy check (CRC).

Once the structure has been populated it is checked to verify that it contains a valid record type (i.e. within ‘0’ and ‘9’), that its size is within the SREC maximum and also the CRC is computed with the received data and compared with the CRC that was received. If any of these conditions is not met, i.e. invalid record type, invalid record size or CRC does not match, an ERR_CRC (0x41) signal is sent back to the device that is sending the data. If everything is received without issues the received data is processed and an ERR_OK (0x45) signal is sent as an acknowledge.

If the type of record received carries the data to write to the microcontroller (either ‘1’, ‘2’ or ‘3’) then the data is processed and written by the memory handling layer.

This process is repeated until the termination record is received (either ‘7’, ‘8’ or ‘9’), once this record is received the communication handling routine ends and returns to the bootloader.

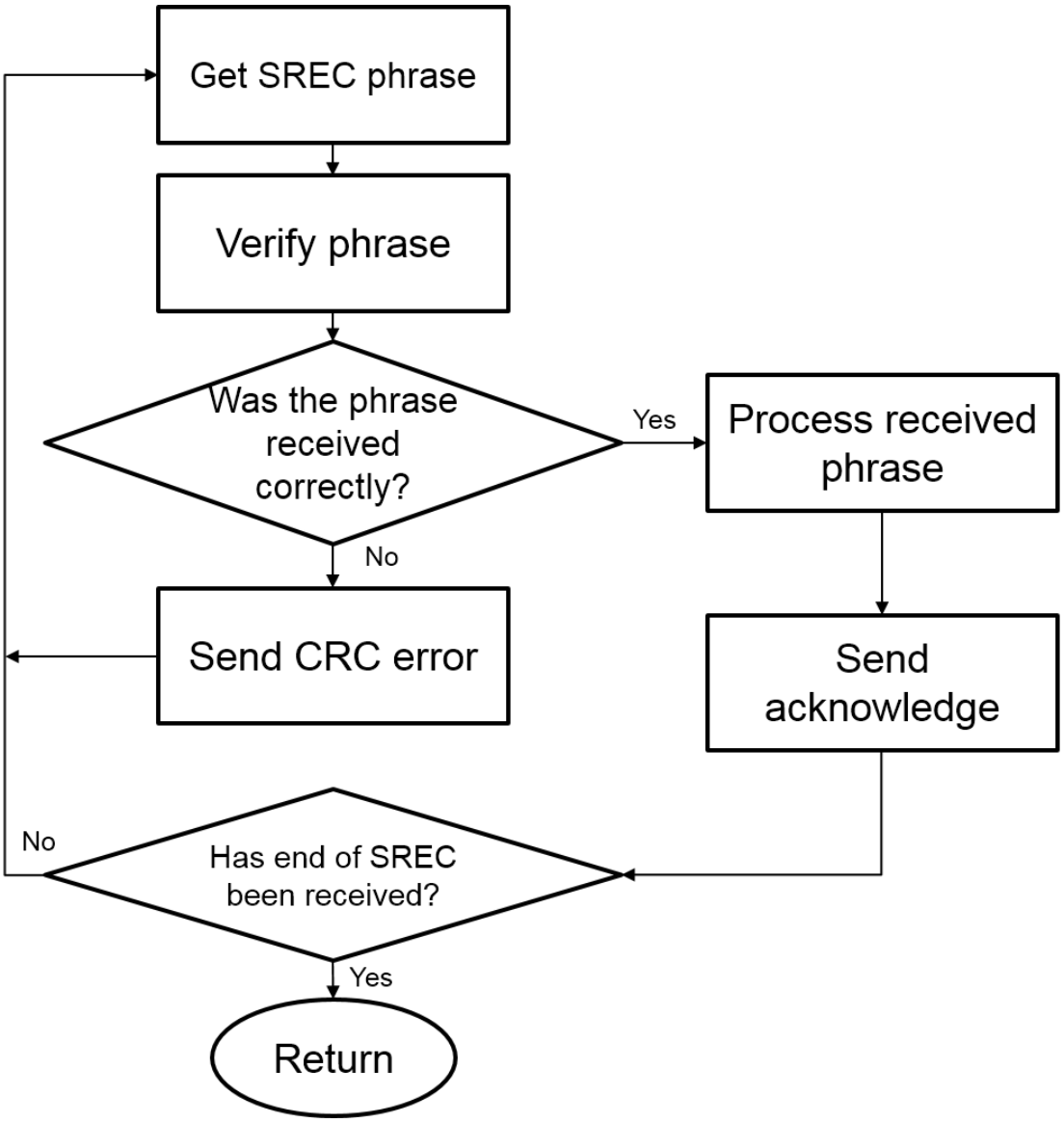


Figure 4. Communication handling workflow

2.2.1. UART communication

When using UART communication to receive, the following data flow is expected:

1. First an ‘S’ (0x53) must be send to signal the beginning of an SREC phrase.
2. The next expected byte signals the type of SREC to be received, this is also send in ASCII format, e.g. ‘0’ (0x30), ‘1’ (0x31), ‘2’ (0x32), …’9’ (0x39).
3. The rest of the SREC phrase (byte count, address, data and checksum) is converted from ASCII to its hexadecimal representation, e.g. from “AF” (0x41 and 0x46) to 0xAF. Once this is

performed the data is sent over UART and the get_phrase routine stops until it has received all the bytes signaled by the “byte count” field on the SREC phrase.

4. As a final step the received data is verified and an error signal is sent back to signal if the data was received correctly, The ERR_CRC (0x41) signal is sent if an error was detected and the ERR_OK (0x45) signal is sent otherwise.
    a. If the master receives:
        i. An ERR_OK signal, the next SREC phrase is sent.
        ii. An ERR_CRC signal, the same SREC phrase is sent again until the slave receives it correctly.

After this process the device is ready to receive the next SREC phrase until the end of the file has been reached.

3. Building compatible applications

The application should start at 0x1000 (4 kB) of flash and its vector table should be placed at this address.
An easy and quick way to compile an application compatible with this bootloader is to simply add an offset of 4 kB to the memory section of the linker file, some examples on two IDEs are shown below.

On S32DS

```
/* Specify the memory areas */
MEMORY
{
  /* Flash */
  m_interrupts          (RX)  : ORIGIN = 0x00001000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x00001400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x00001410, LENGTH = 0x0017EBF0

  /* SRAM_L */
  m_data                (RW)  : ORIGIN = 0x1FFE0000, LENGTH = 0x00020000

  /* SRAM_U */
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x0001F000
}
```

4. Using the bootloader

The example software contained in this application note is for the S32K148 EVB. However, it can be easily migrated to other S32K1xx devices. The bootloader expects the image to load in SREC format, for instructions on how to generate an SREC file on S32DS please refer to Appendix A. Precompiled example SREC file is available within this application note package. It is a simple routine expected to run on the S32K148EVB

NOTE:

Some IDE's place the name of the project in the first SREC phrase (S0), this can cause issues with the bootloader whenever the project name exceeds 27 characters. The maximum data per phrase is 32 bytes, but the IDE appends the string '.srec' to the project name, hence the 27 characters as maximum allowed.

The bootloader supports one communication channel:

- UART

4.1. UART interface

While using the UART interface simply open the java application located in 'Java interface/' and follow these steps:

1. Select communication port.
2. Select baudrate, the default baudrate is 19200.
3. Select SREC file to send.
4. Click download, and the SREC file will be sent line after line.

NOTE

Java JRE 8 32-bit version must be installed before trying to use the interface.

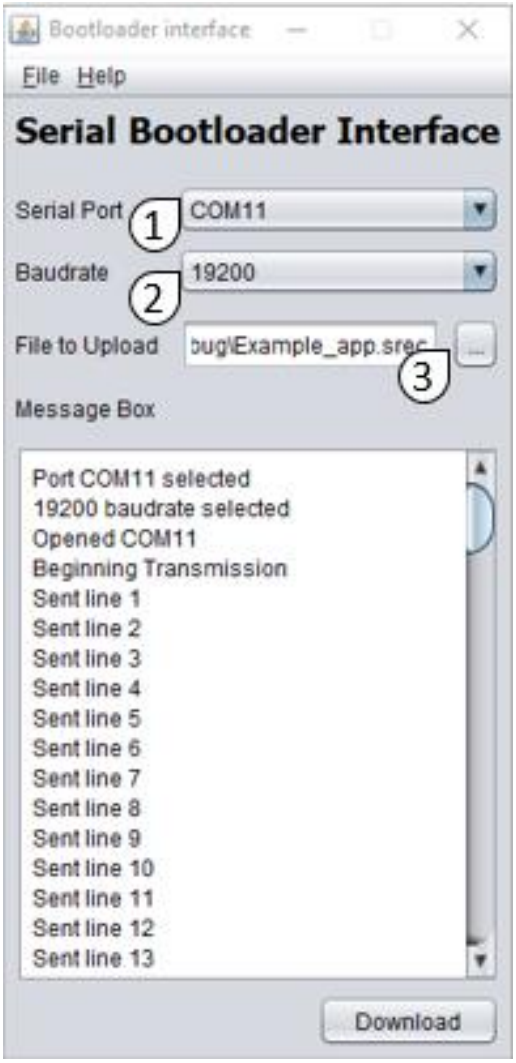


Figure 5. UART Bootloader Interface steps

Once the whole SREC file has been sent the java interface will close the port and the application should start execution on the target board.

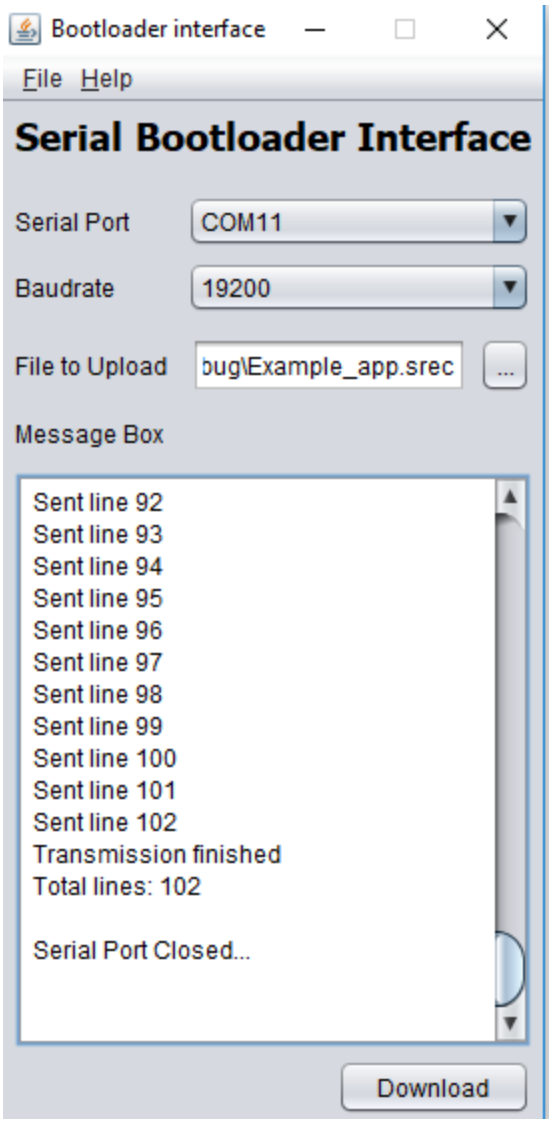
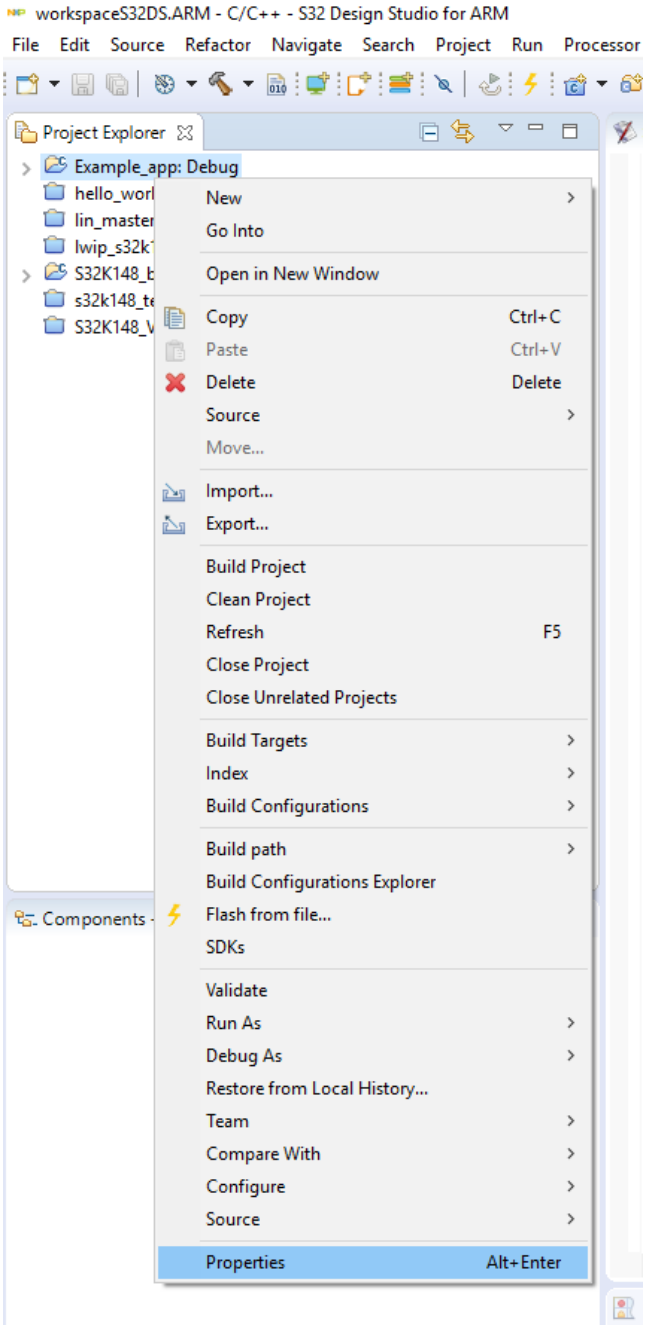
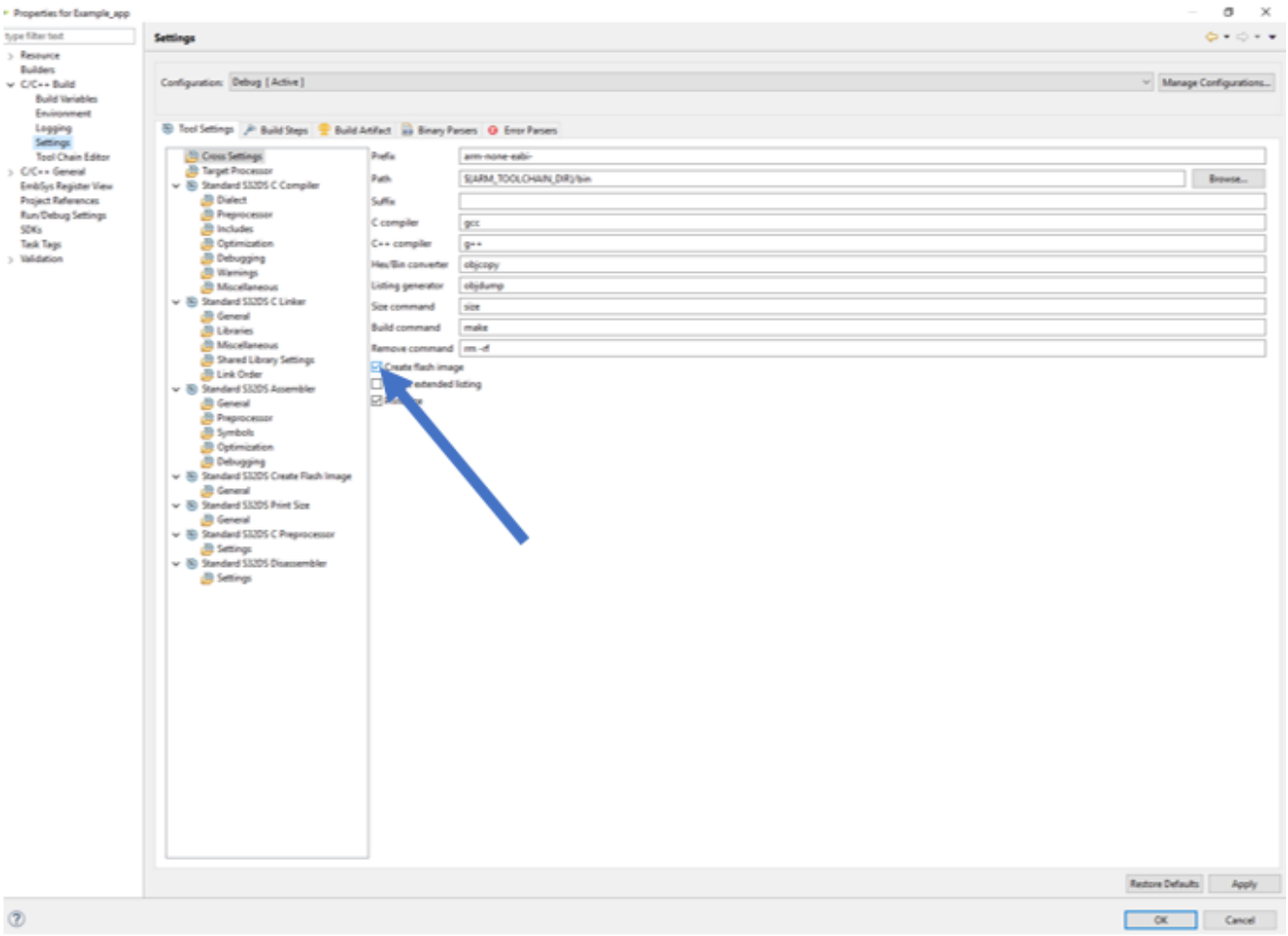


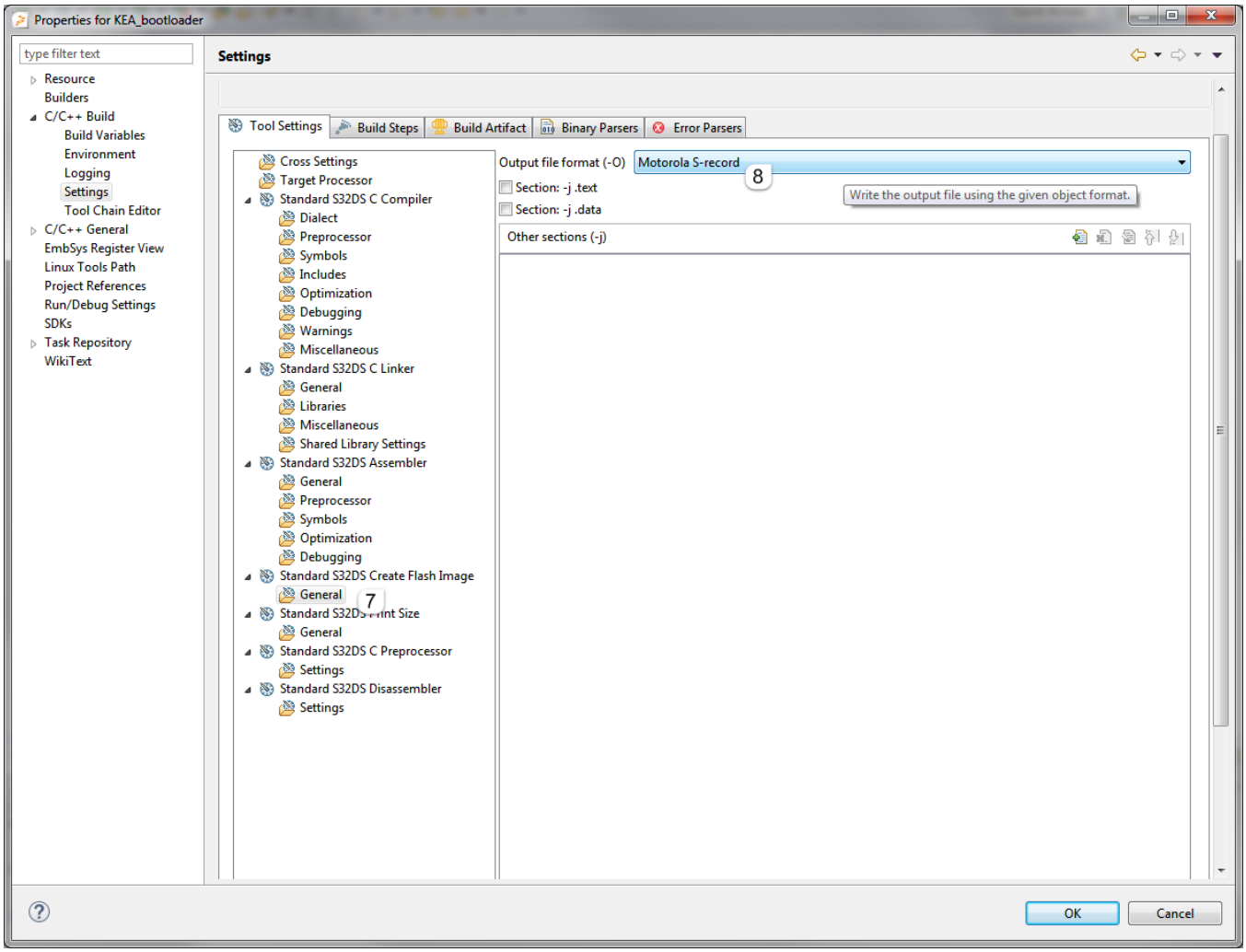
Figure 6. UART Bootloader Interface complete

5. Appendix A

5.1. On S32DS:




Properties for Example_app
Settings
Configuration: Debug [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
Cross Settings
Prefix
arm-none-eabi-
Path
Browse...
Suffix
C compiler
gcc
C++ compiler
g++
Hex/Bin converter
objcopy
Listing generator
objdump
Size command
size
Build command
make
Remove command
rm -rf
Create flash image
Restore Defaults
Apply
OK
Cancel



6. Revision History

Revision Number	Date	Substantive changes
0	07/2018	Initial release
1	10/2018	Added associated software

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C 5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM, AMBA, ARM Powered, Artisan, Cortex, Jazelle, Keil, SecurCore, Thumb, TrustZone, and µVision are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. ARM7, ARM9, ARM11, big.LITTLE, CoreLink, CoreSight, DesignStart, Mali, mbed, NEON, POP, Sensinode, Socrates, ULINK and Versatile are trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2018 NXP B.V.

Document Number: AN12218
Rev. 1
10/2018