

KEA Bootloader

1 Introduction

This application note describes the architecture and usage of the KEA bootloader within the [KEA MCUs](#).

This bootloader supports SPI (Serial Peripheral Interface) and UART (Universal Asynchronous Receiver/Transmitter) 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 in charge of starting the user application and polling for incoming data
- Communication handling / Memory handling – is in charge 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:

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 SPI interface.....	9
4.2 UART interface.....	9
A Appendix A.....	11
A.1 On KDS.....	11
A.2 S32DS.....	16
B Appendix B.....	20
B.1 Linker file on KDS.....	20
B.2 Linker file on S32DS.....	23



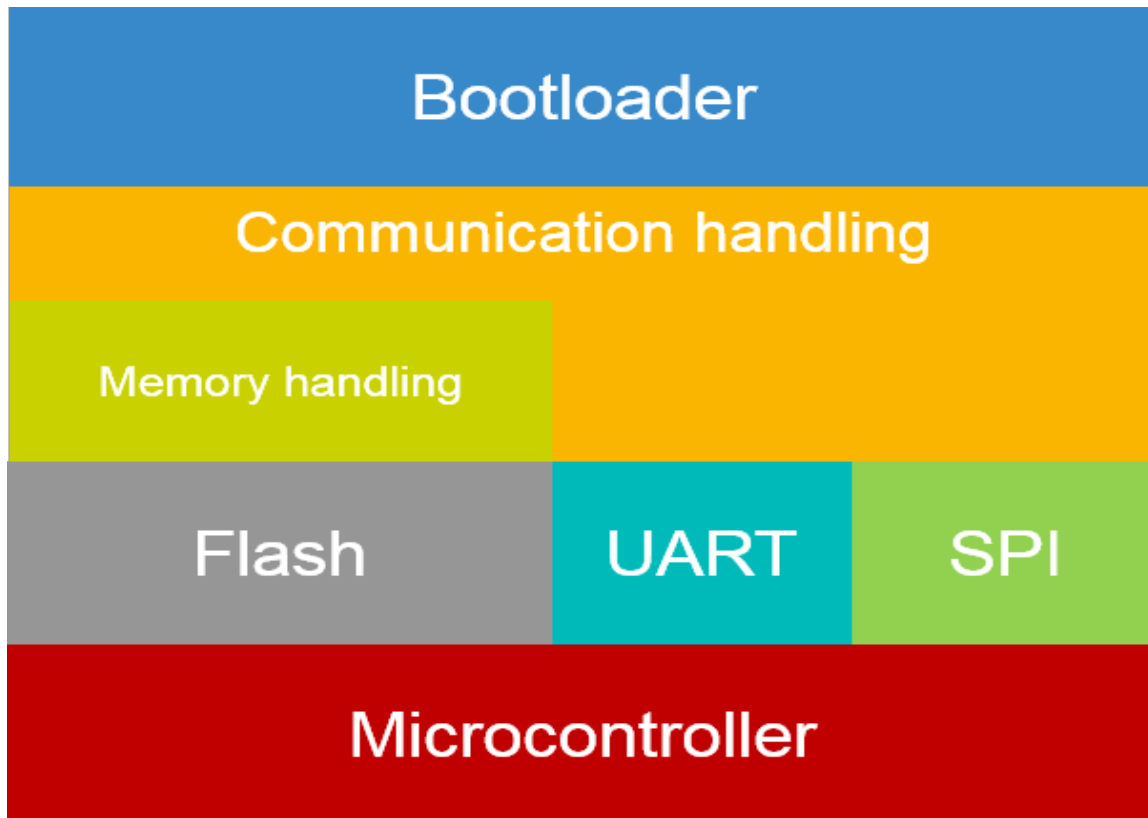


Figure 1. Bootloader Architecture

The bootloader is placed in the first 4 kB of flash memory. The current footprint of the bootloader is 3.25 kB the remaining .75 kB is reserved for future usage. This initial 4kB of memory is write protected from any attempt to overwrite the boot loader from the user application will be blocked by this protection mechanism (for more details regarding this protection mechanism please refer to the device reference manual chapter 18.3.6 Protection).

The remaining 124 kB available on the device are intended to be used by the user application. The following diagram showcases the memory layout that the bootloader has and the application must follow:

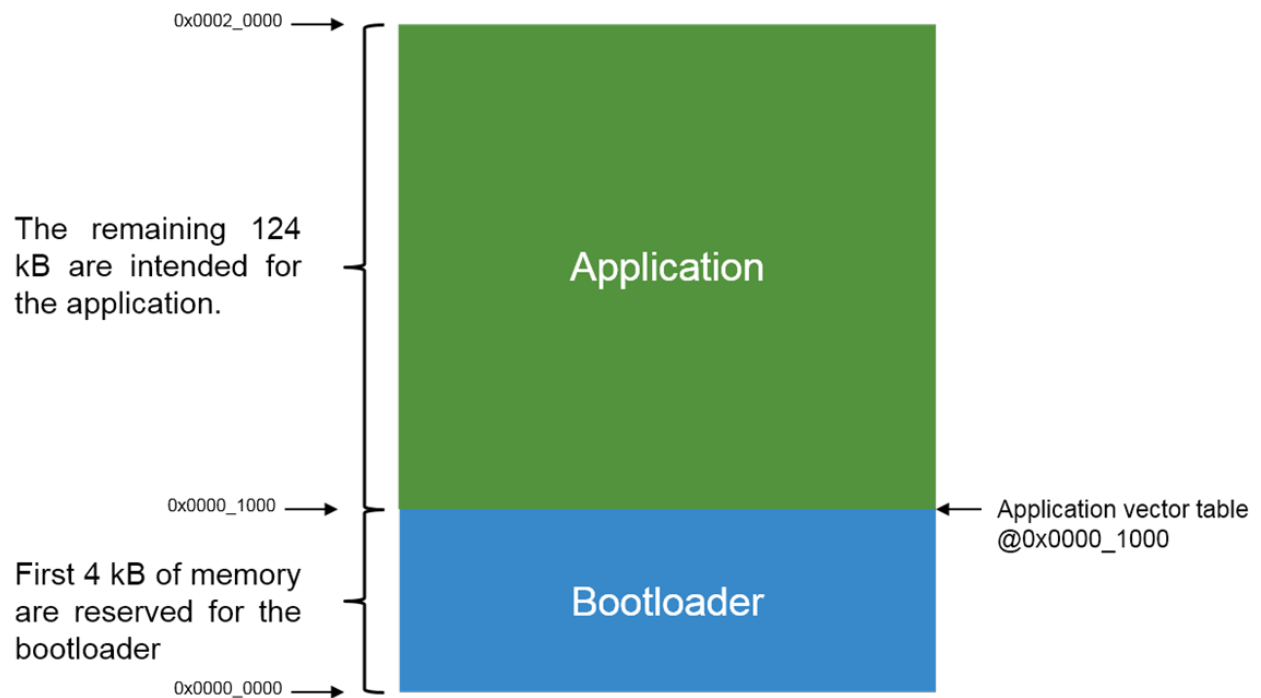


Figure 2. Memory layout

As displayed in the above figure, the bootloader expects the application vector table to be located at the end of the first 4 kB of flash (i.e. 0x1000), this is required since the application stack pointer and entry point are taken from this vector table.

2.1 Bootloader workflow overview

The bootloader workflow can be observed in the figure below:

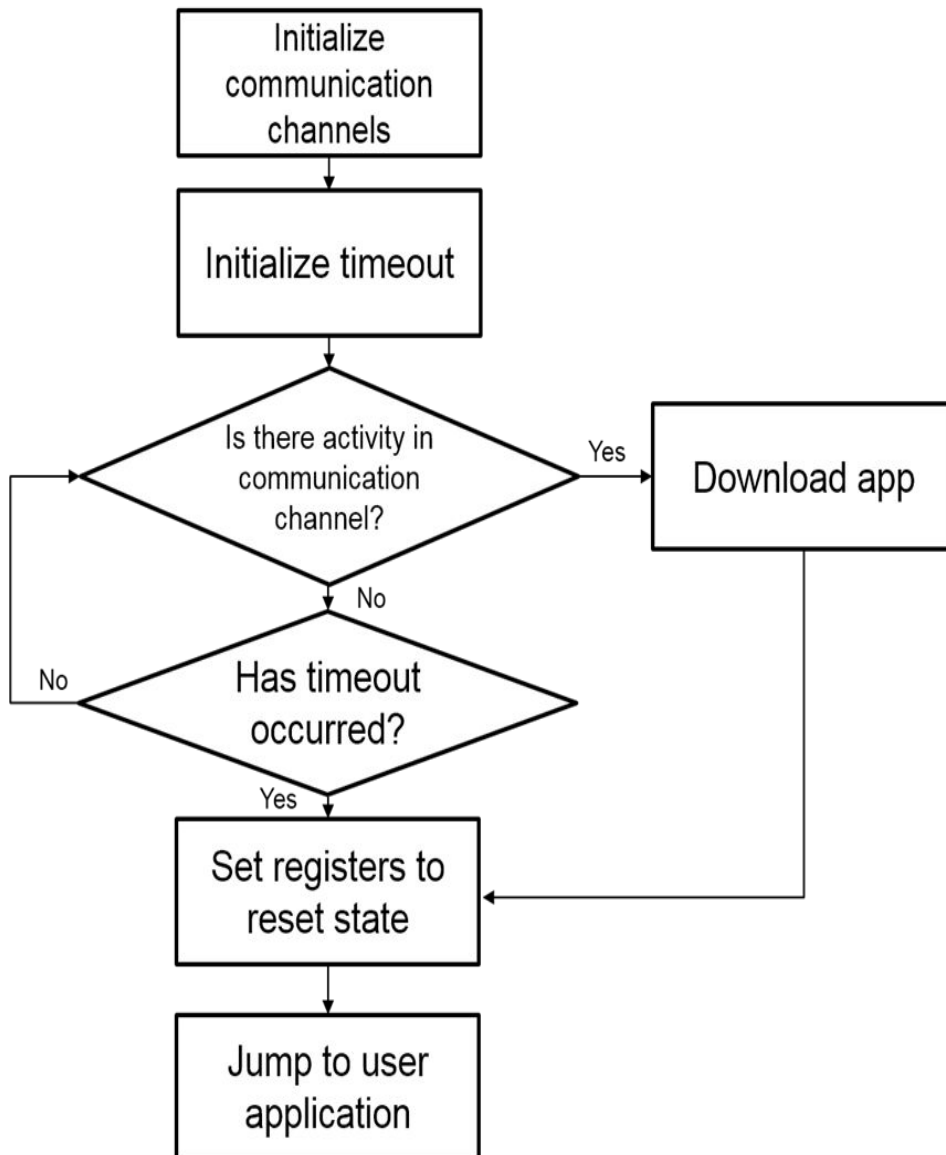


Figure 3. Bootloader workflow

The first step is to initialize the available communication channels, in this instance only UART and SPI are 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  
  
#define SPI_COMM       0
```

Both 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, in case both interfaces detect activity at the same time UART communication has a higher priority and will be used instead of SPI.

By default the bootloader is set to work with UART communication only.

The second step is to initialize the timeout mechanism. After 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 did not received an application it would 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 or SPI).

If a timeout occurs or an application is flashed to the device, the bootloader disables and set all the registers that are modified to its reset state. This step is required to ensure that, 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 residing on 0x1000 address.

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:

```
S00F000068656C6C6F202020202000003C
S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C6000003863000026
```

Record type Byte count Address Data Checksum

The structure of an SREC line can be seen below:

S	Type	Byte Count	Address	Data	Checksum
---	------	------------	---------	------	----------

Figure 4. SREC phrase structure

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: [https://en.wikipedia.org/wiki/SREC_\(file_format\)](https://en.wikipedia.org/wiki/SREC_(file_format))

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'), 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 are 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.

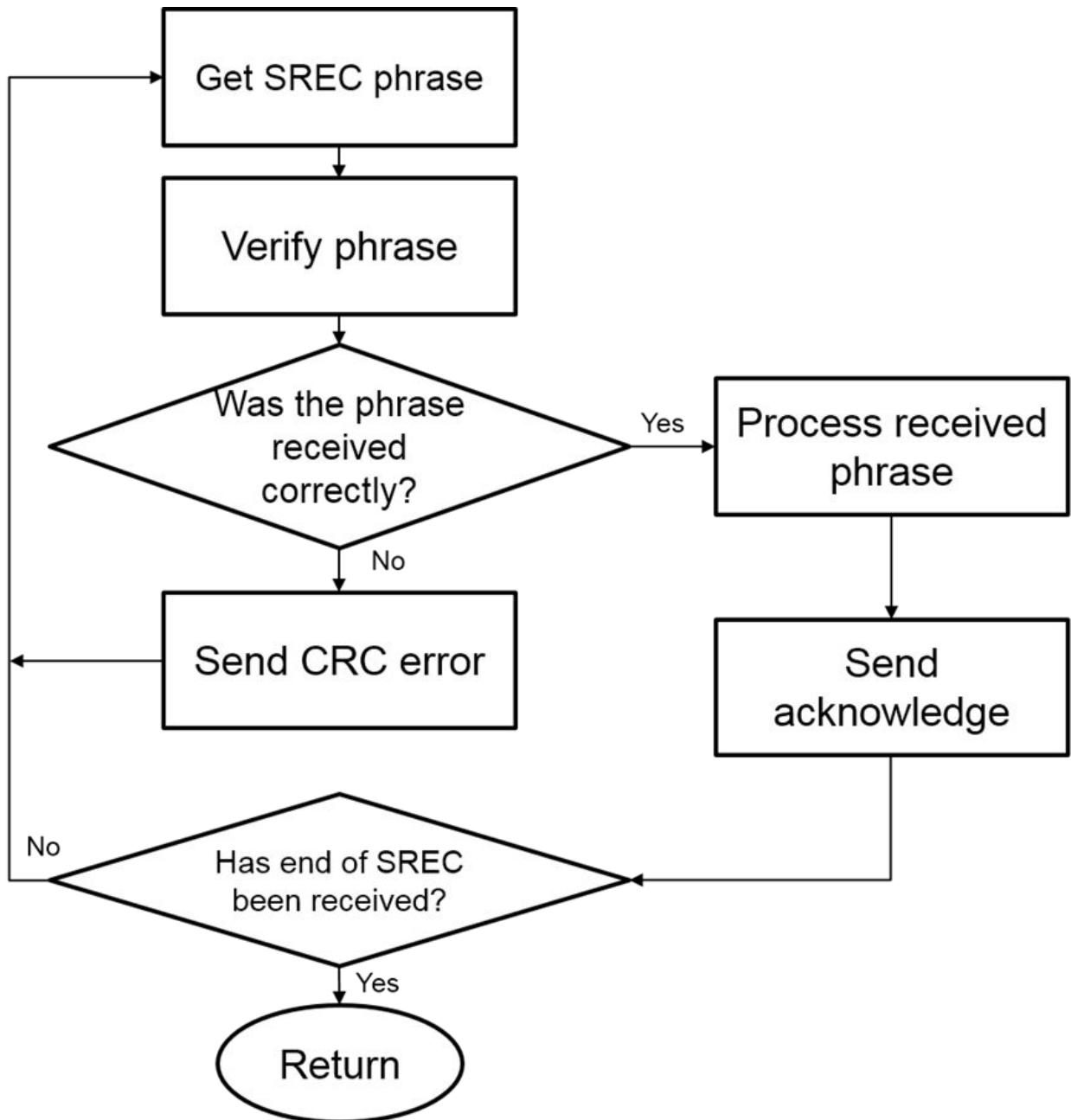


Figure 5. Communication handling workflow

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 KDS:

MEMORY

```
{
m_interrupts      (RX)      : ORIGIN = 0x00001000,
    LENGTH = 0x00000100

    m_flash_config      (RX)      : ORIGIN = 0x00001400, LENGTH =
0x00000010

    m_text              (RX)
: ORIGIN = 0x00001410, LENGTH = 0x0001DBF0

    m_data
(RW)      : ORIGIN = 0x1FFFF000, LENGTH =
0x00004000
}
```

On S32DS:

MEMORY

```
{
FLASH_1          (RX)      : ORIGIN = 0x00001000, LENGTH = 0x00000100

    FLASH_CONFIG      (RX)      : ORIGIN = 0x00001400, LENGTH = 0x00000010
FLASH_2          (RX)      : ORIGIN =      0x00001410, LENGTH = 0x0001DBF0
SRAM              (RW)      : ORIGIN = 0x1FFFF000, LENGTH = 0x00004000
}
```

The best way to adapt an application to work with the bootloader is to eliminate the flash_config memory section of the linker file (Project_Settings/Linker_Files/SKEAZ128xxx4_flash.ld) as well as the actual configuration in the startup file (Project_Settings/Startup_Code/startup_SKEAZ1284.S line 100), there is a sample application attached within this AN package. Also [Appendix B](#) on page 20 features an example linker file for both S32DS and KDS.

As long as the vector table is located at 0x1000 (4 kB) and the application memory space does not overlap with the bootloader's (0x00 – 0x1000) the bootloader is capable of flashing and running the application.

4 Using the bootloader

The bootloader expects the image to load in SREC format, for instructions on how to generate an SREC file on KDS and S32DS please refer to [Appendix A](#) on page 11. Precompiled example SREC files are available within this AN package, these examples work on both the TRK-KEA128 and FRDM-KEAZ128 boards.

NOTE

Some IDE's place the name of the project in the first SREC phrase (S0), this could 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 two different communication channels:

- UART
- SPI

If the UART interface (default) is selected then the bootloader interface can be used directly with the board, otherwise a UART to SPI translator board is required (source code included in AN package).

4.1 SPI interface

To use the SPI interface a UART to SPI bridge must be used, this has been implemented on the UART2SPI project, simply load this project on the board that will work as a converter and connect the signals as follow:

Table 1. SPI connections

Bridge (Master)	Target (Slave)
PTG4 (SCL)	PTD0 (SCL)
PTG5 (MOSI)	PTD1 (MOSI)
PTG6 (MISO)	PTD2 (MISO)
PTG7 (PCS)	PTD3 (PCS)

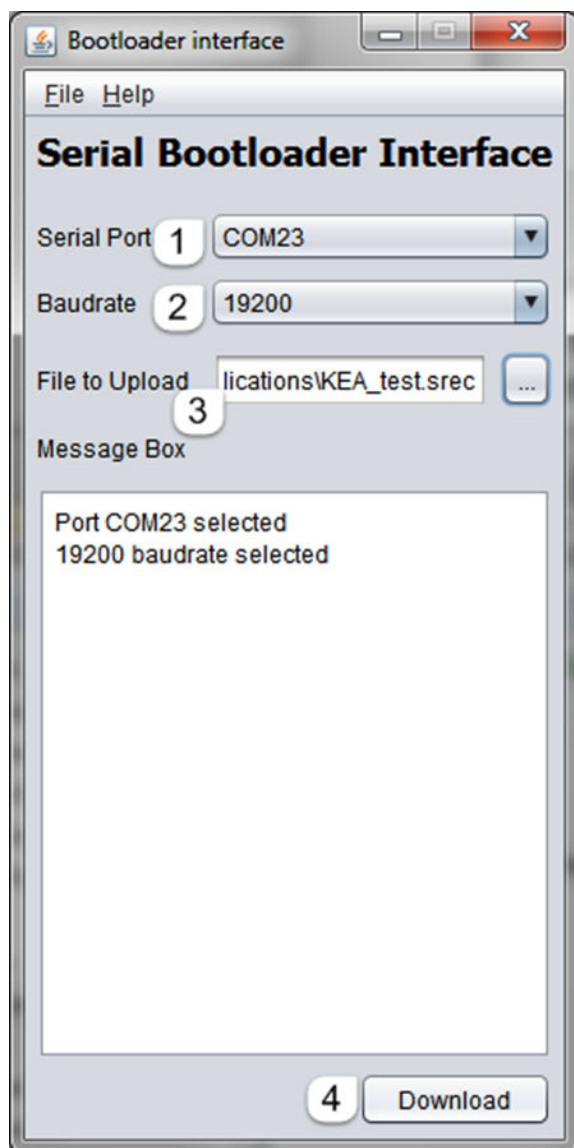
Once connected simply follow the steps described on the UART interface section (select the bridge's UART port instead of the target).

4.2 UART interface

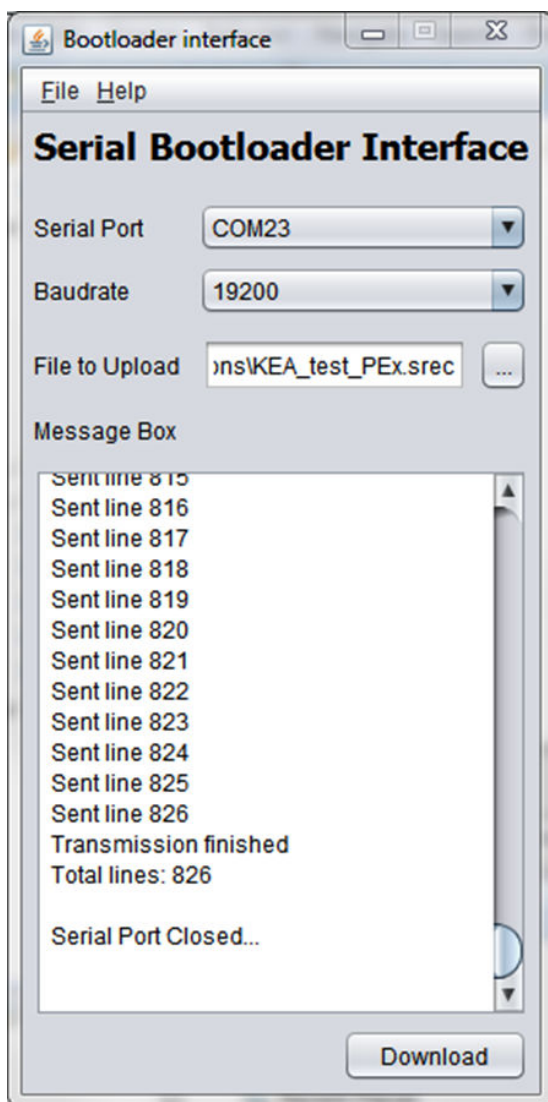
While using the UART interface simply open the java application located in 'Java interface/' and follow these steps:

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.

Using the bootloader
UART interface



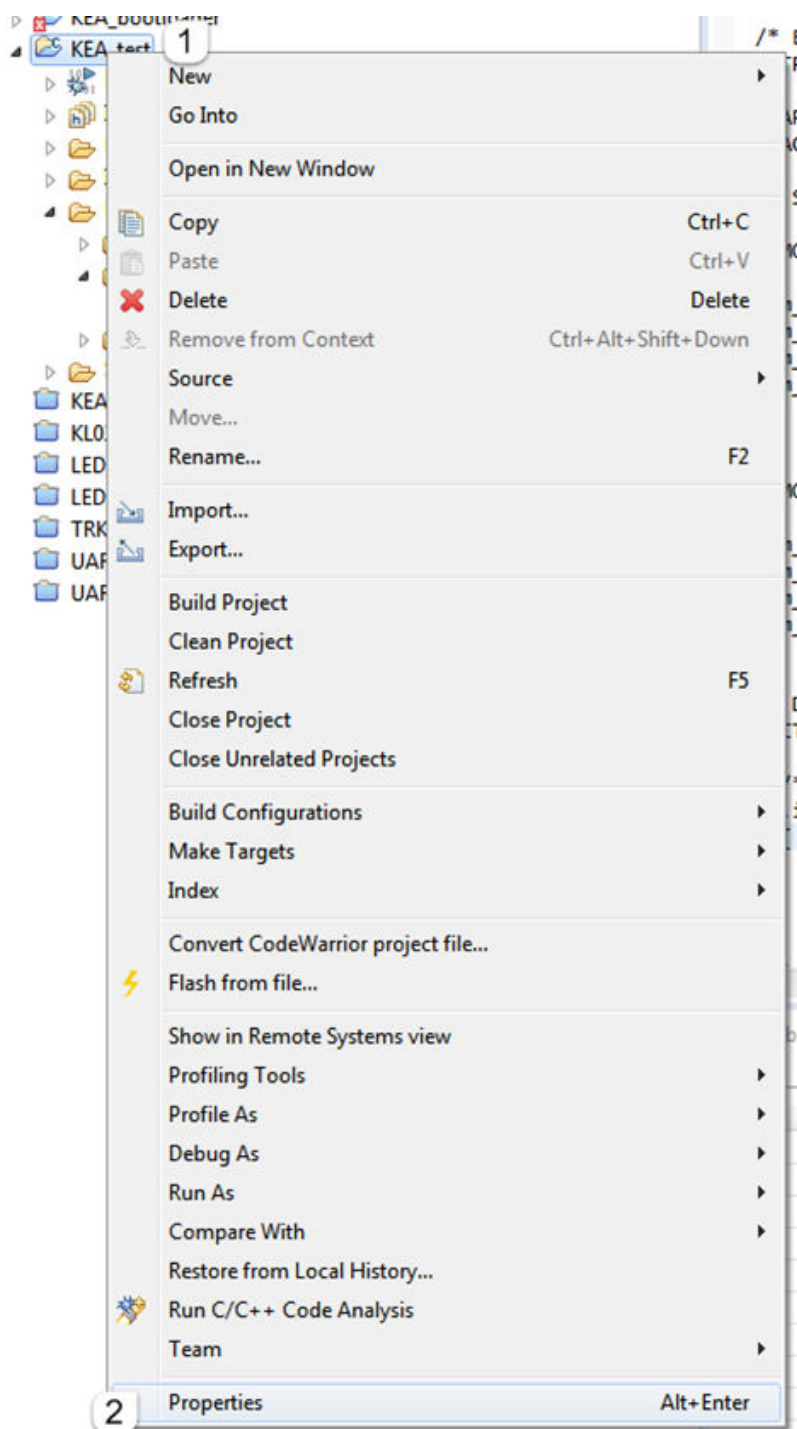
Once the whole SREC file has been sent the java interface will close the port and the application should start execution on the target board.

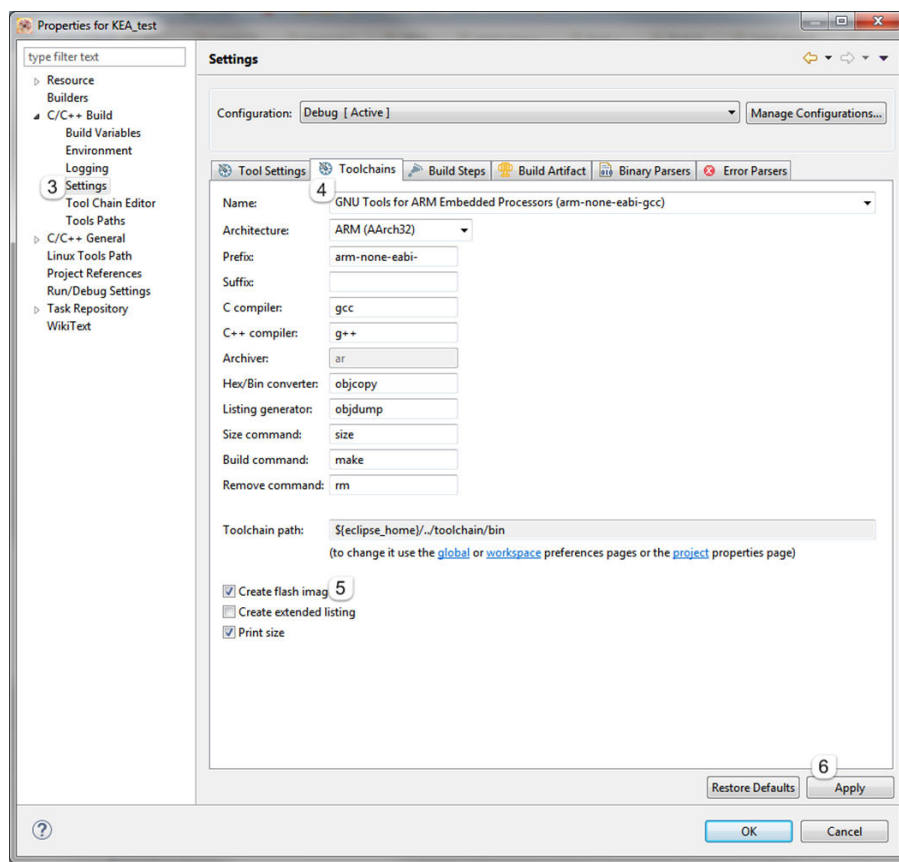


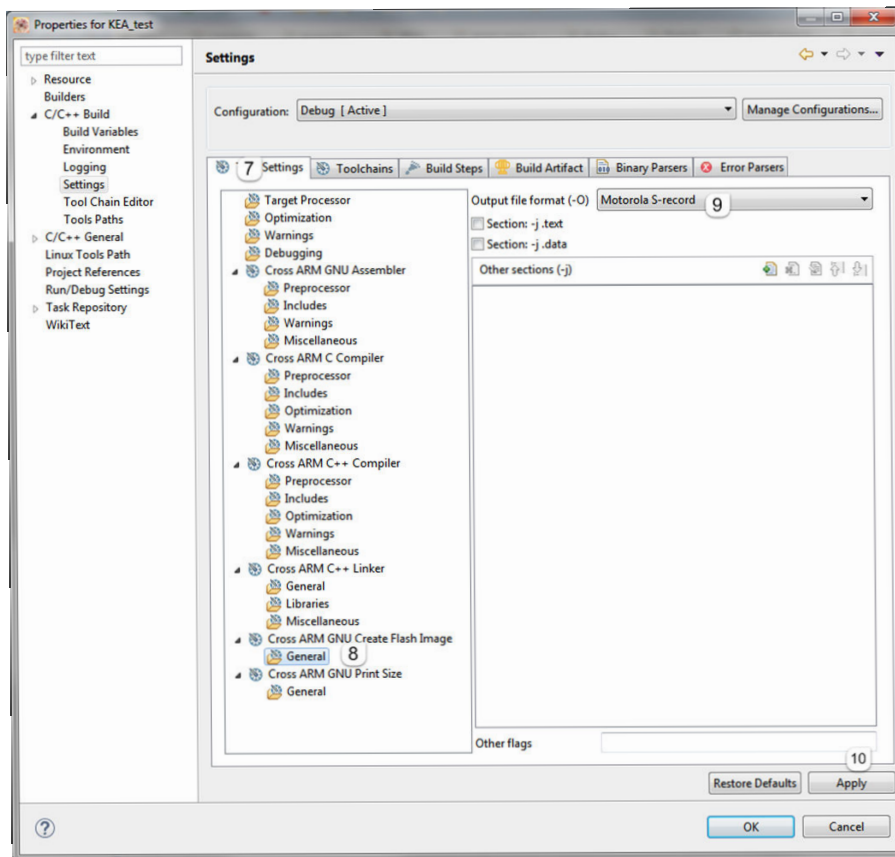
A Appendix A

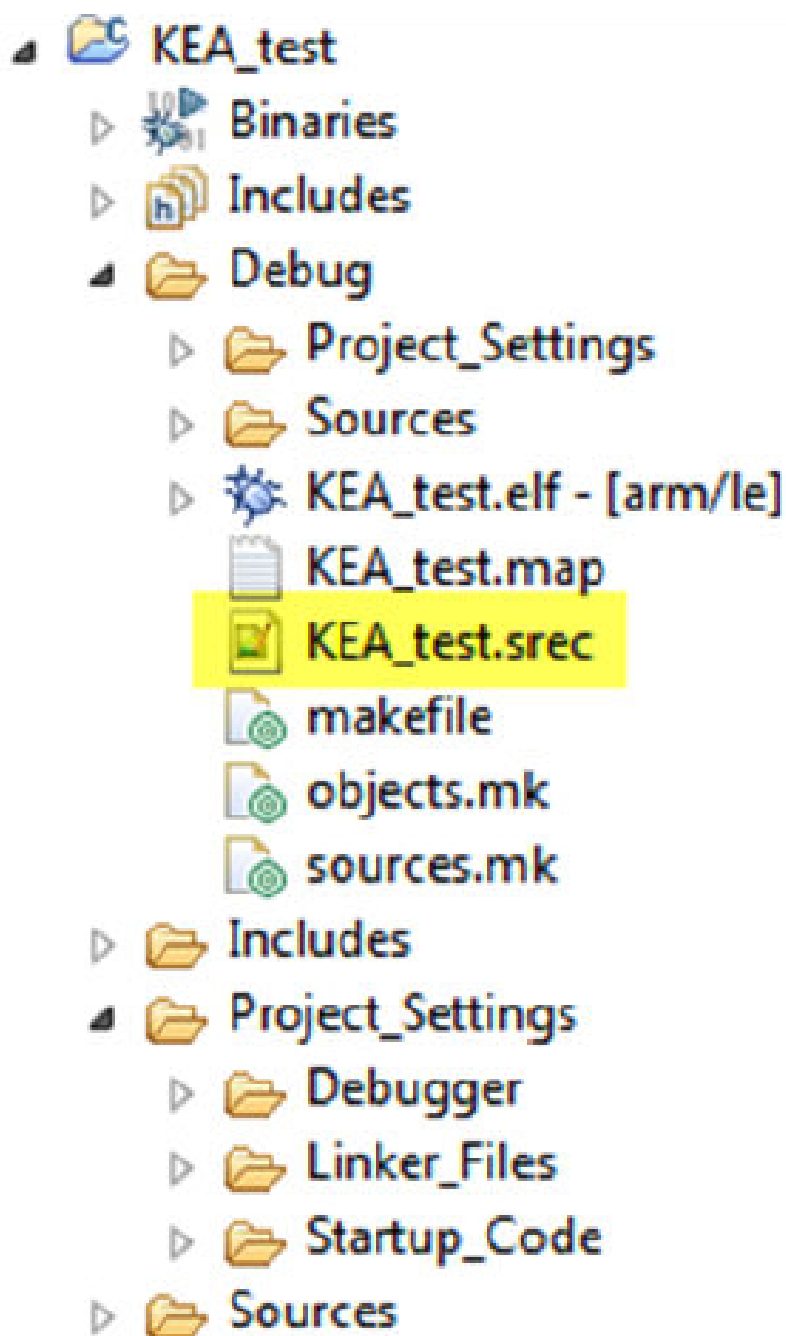
A.1 On KDS

On KDS:

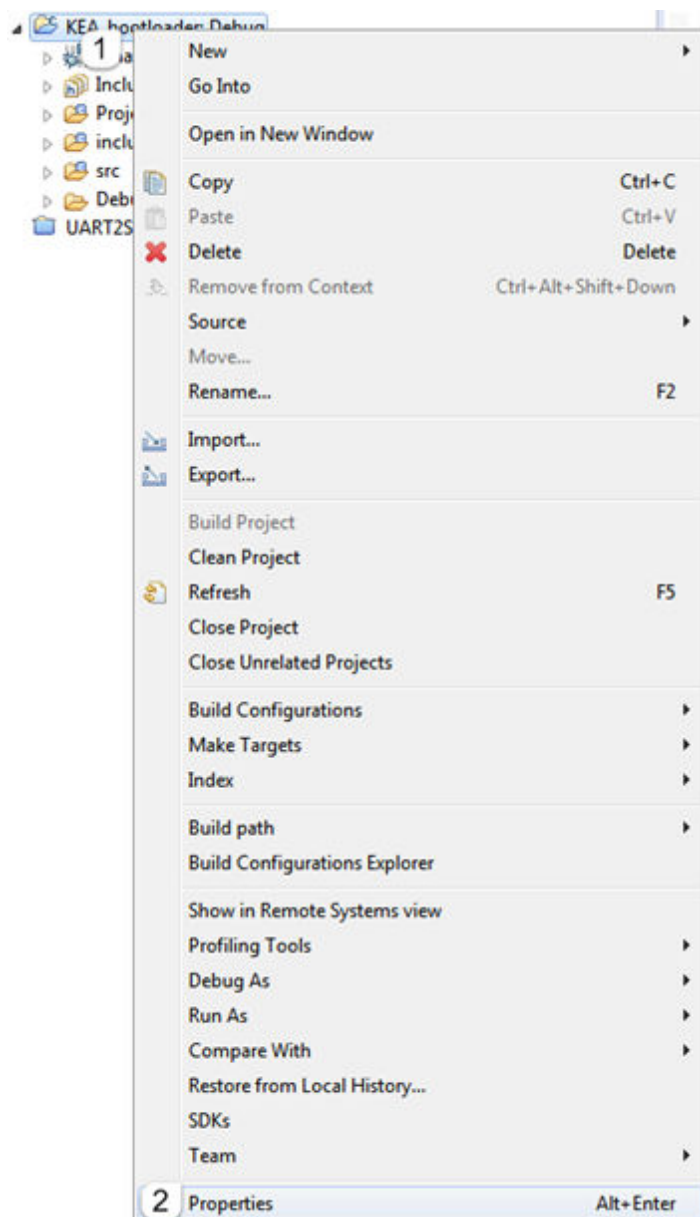


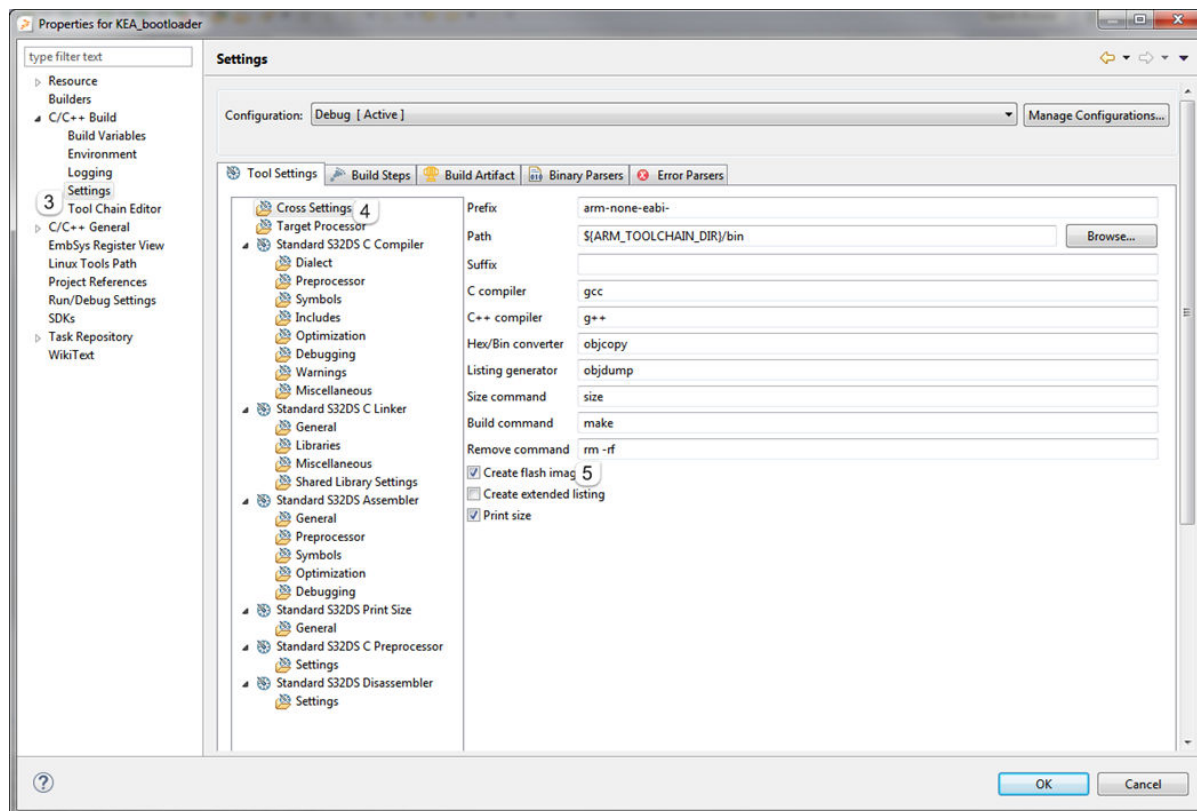


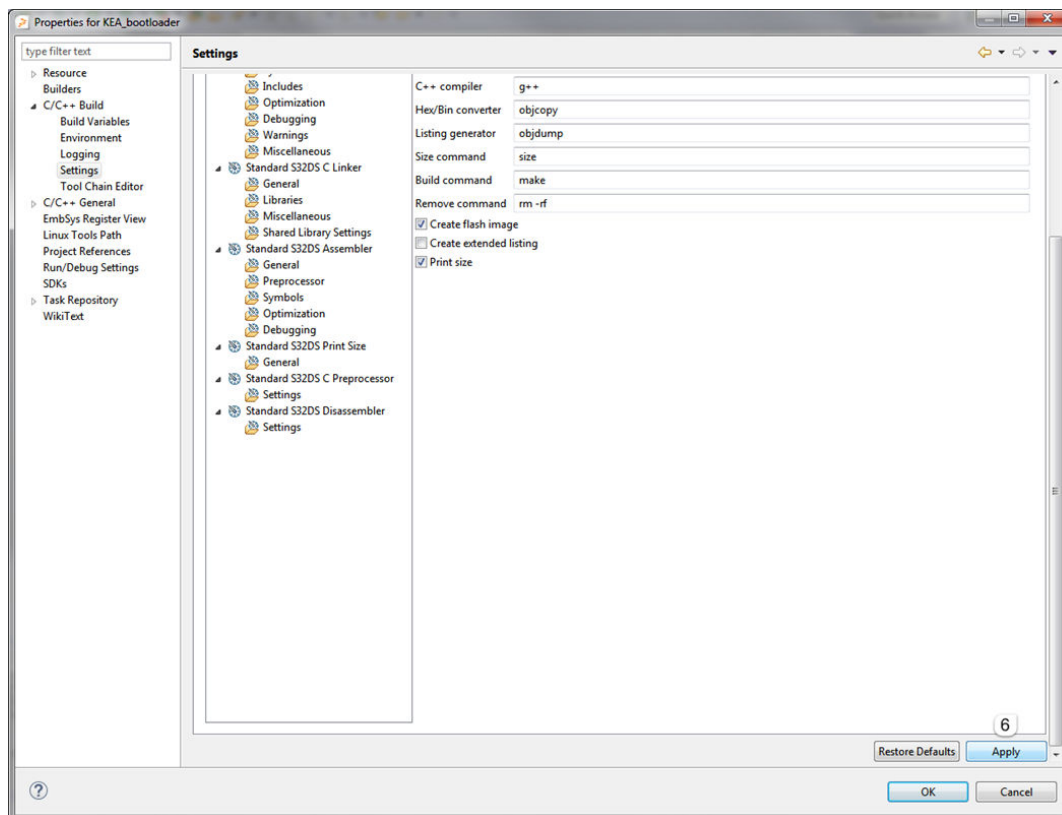


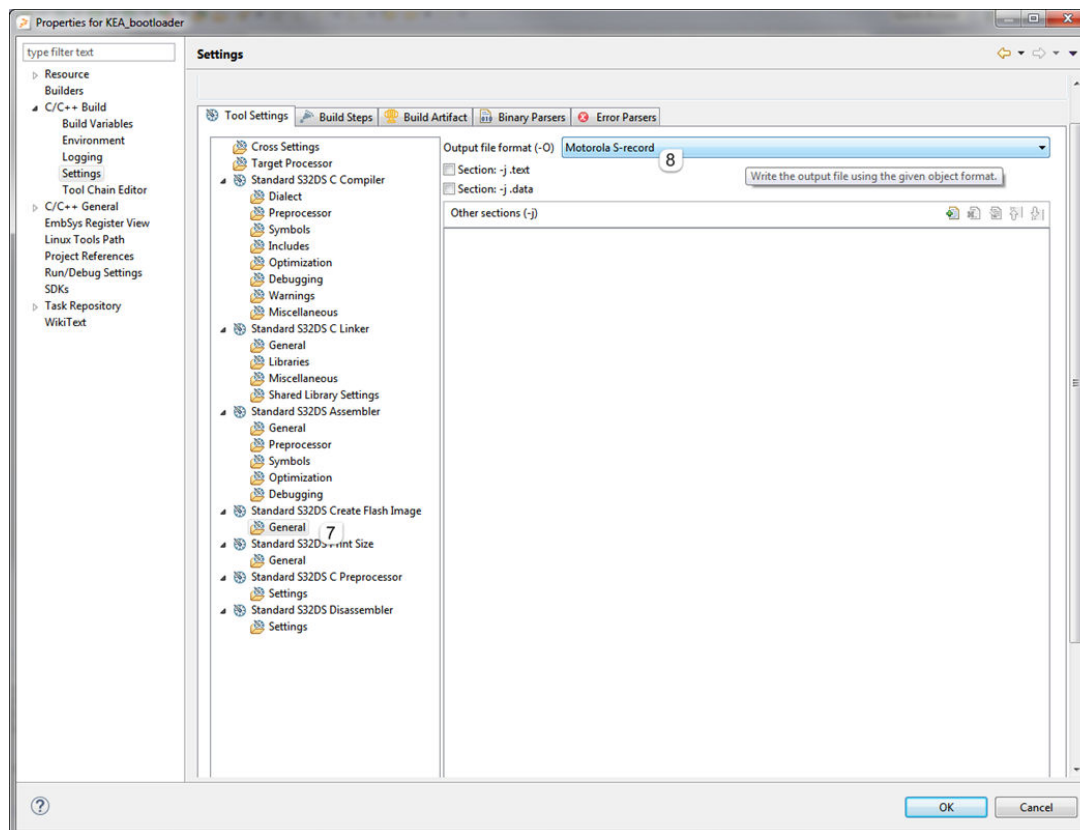


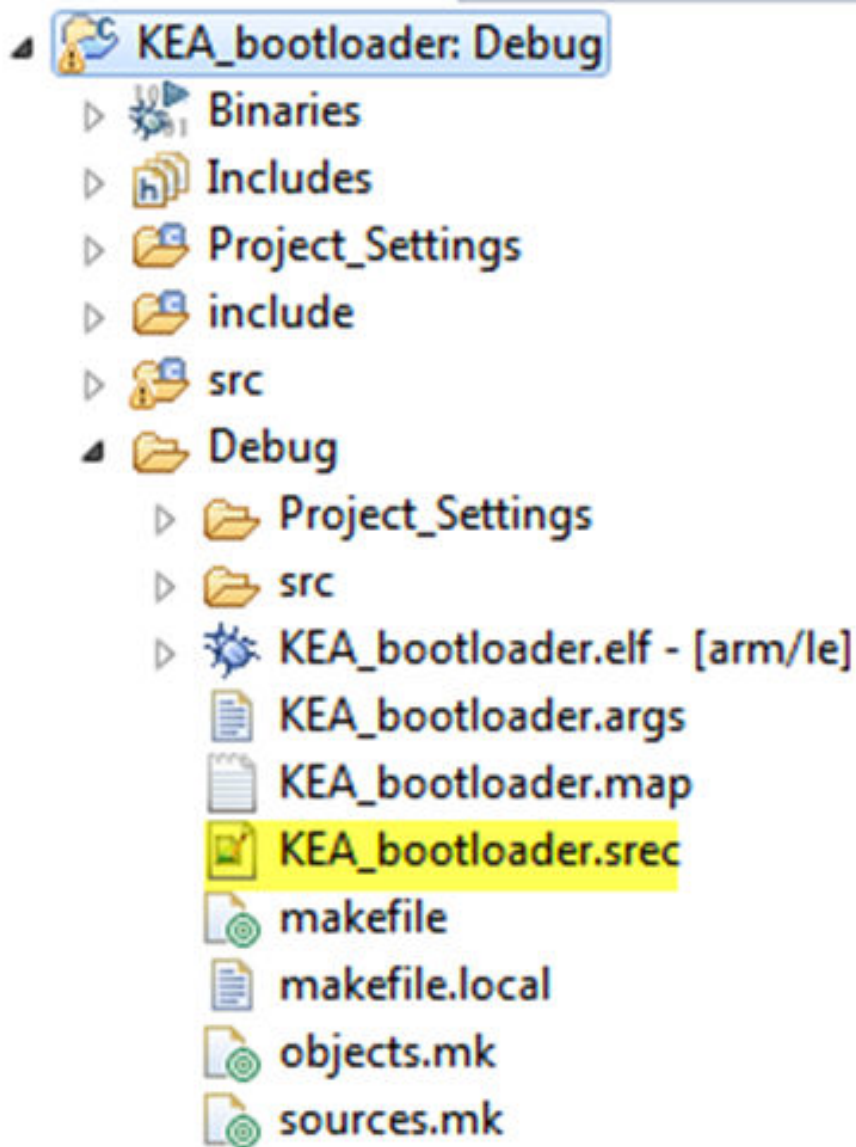
A.2 S32DS











B Appendix B

B.1 Linker file on KDS

```
/* Linker file for GNU C Compiler*/

/* Entry Point */
ENTRY(Reset_Handler)
HEAP_SIZE = DEFINED(__heap_size__) ? __heap_size__ : 0x00000400;
STACK_SIZE = DEFINED(__stack_size__) ? __stack_size__ : 0x00000400;
```

```

/* Specify the memory areas */
MEMORY
{
m_interrupts      (RX)      : ORIGIN = 0x00001000, LENGTH = 0x00000100
m_text            (RX)      : ORIGIN = 0x00001100, LENGTH = 0x0001EF00
m_data            (RW)      : ORIGIN = 0x1FFFF000, LENGTH = 0x00004000
}

/* Define output sections */
SECTIONS
{
/* The startup code goes first into internal flash */
.interrupts :
{
__VECTOR_TABLE = .;
. = ALIGN(4);
KEEP(*(.(isr_vector)) /* Startup code */
. = ALIGN(4);
} > m_interrupts

/* The program code and other data goes into internal flash */
.text :
{
. = ALIGN(4);
*(.text) /* .text sections (code) */
*(.text*) /* .text* sections (code) */
*(.rodata) /* .rodata sections (constants, strings, etc.) */
*(.rodata*) /* .rodata* sections (constants, strings, etc.) */
*(.glue_7) /* glue arm to thumb code */
*(.glue_7t) /* glue thumb to arm code */
*(.eh_frame)
KEEP (*(.(init))
KEEP (*(.(fini))
. = ALIGN(4);
} > m_text

.ARM.extab :
{
*(.ARM.extab* .gnu.linkonce.armextab.*)
} > m_text

.ARM :
{
__exidx_start = .;
*(.ARM.exidx*)
__exidx_end = .;
} > m_text

.ctors :
{
__CTOR_LIST__ = .;
/* gcc uses crtbegin.o to find the start of
the constructors, so we make sure it is
first. Because this is a wildcard, it
doesn't matter if the user does not
actually link against crtbegin.o; the
linker won't look for a file to match a
wildcard. The wildcard also means that it
doesn't matter which directory crtbegin.o

```

Appendix B

Linker file on KDS

```
is in. */
KEEP (*crtbegin.o(.ctors))
KEEP (*crtbegin?.o(.ctors))
/* We don't want to include the .ctor section from
from the crtend.o file until after the sorted ctors.
The .ctor section from the crtend file contains the
end of ctors marker and it must be last */
KEEP (*(EXCLUDE_FILE(*crtend?.o *crtend.o) .ctors))
KEEP (*(SORT(.ctors.*)))
KEEP (*(.ctors))
__CTOR_END__ = .;
} > m_text

.dtors :
{
__DTOR_LIST__ = .;
KEEP (*crtbegin.o(.dtors))
KEEP (*crtbegin?.o(.dtors))
KEEP (*(EXCLUDE_FILE(*crtend?.o *crtend.o) .dtors))
KEEP (*(SORT(.dtors.*)))
KEEP (*(.dtors))
__DTOR_END__ = .;
} > m_text

.preinit_array :
{
PROVIDE_HIDDEN (__preinit_array_start = .);
KEEP (*(.preinit_array*))
PROVIDE_HIDDEN (__preinit_array_end = .);
} > m_text

.init_array :
{
PROVIDE_HIDDEN (__init_array_start = .);
KEEP (*(SORT(.init_array.*)))
KEEP (*(.init_array*))
PROVIDE_HIDDEN (__init_array_end = .);
} > m_text

.fini_array :
{
PROVIDE_HIDDEN (__fini_array_start = .);
KEEP (*(SORT(.fini_array.*)))
KEEP (*(.fini_array*))
PROVIDE_HIDDEN (__fini_array_end = .);
} > m_text

__etext = .; /* define a global symbol at end of code */
__DATA_ROM = .; /* Symbol is used by startup for data initialization */

/* reserve MTB memory at the beginning of m_data */
.mtb : /* MTB buffer address as defined by the hardware */

. = ALIGN(8);
__mtb_start = .;
KEEP(*(.mtb_buf)) /* need to KEEP Micro Trace Buffer as not referenced by application
*/
= ALIGN(8);
mtb_end = .;
} > m_data
```

```
.data : AT(__DATA_ROM)
{
    . = ALIGN(4);
    __DATA_RAM = .;
    __data_start__ = .;      /* create a global symbol at data start */
    *(.data)                 /* .data sections */
    *(.data*)                /* .data* sections */
    KEEP(*(.jcr*))
    . = ALIGN(4);
    __data_end__ = .;        /* define a global symbol at data end */
} > m_data

/* Symbol is used by startup for data initialization */
__DATA_END = __DATA_ROM + (__data_end__ - __data_start__);

/* Uninitialized data section */
.bss :
{
    /* This is used by the startup in order to initialize the .bss section */
    . = ALIGN(4);
    __START_BSS = .;
    __bss_start__ = .;
    *(.bss)
    *(.bss*)
    *(COMMON)
    . = ALIGN(4);
    __bss_end__ = .;
    __END_BSS = .;
} > m_data

.heap :
{
    . = ALIGN(8);
    __end__ = .;
    PROVIDE(end = .);
    __HeapBase = .;
    += HEAP_SIZE;
    __HeapLimit = .;
} > m_data

.stack :
{
    . = ALIGN(8);
    . += STACK_SIZE;
} > m_data

__StackTop    = .;
__StackLimit = __StackTop - STACK_SIZE;
PROVIDE(__stack = __StackTop);

.ARM.attributes 0 : { *(.ARM.attributes) }
}
```

B.2 Linker file on S32DS

```
/* Linker file for GNU C Compiler */
```

```

/* Entry Point */
ENTRY(Reset_Handler)

/*
To use "new" operator with EWL in C++ project the following symbol shall be defined
*/
/*EXTERN(_ZN10__cxxabiv119__terminate_handlerE)*/
/* Highest address of the user mode stack */
_estack = 0x20003000;      /* end of SRAM */
__SP_INIT = _estack;
HEAP_SIZE = DEFINED(__heap_size__) ? __heap_size__ : 0x00000500;
STACK_SIZE = DEFINED(__stack_size__) ? __stack_size__ : 0x00000400;
/* Specify the memory areas*/
MEMORY
{
FLASH_1          (RX)      : ORIGIN = 0x00001000, LENGTH = 0x0001F000
SRAM              (RW)      : ORIGIN = 0x1FFFF000, LENGTH = 0x00004000
/* Define output sections*/
SECTIONS
{
/* The startup code goes first into internal flash */
.interrupts :
{
__VECTOR_TABLE = .;
. = ALIGN(4);
KEEP(*(.isr_vector))      /* Startup code */
. = ALIGN(4);
} > FLASH_1

/* The program code and other data goes into internal flash */
.text :
{
. = ALIGN(4);
*(.text)                  /* .text sections (code) */
*(.text*)                 /* .text* sections (code) */
*(.rodata)                /* .rodata sections (constants, strings, etc.) */
*(.rodata*)              /* .rodata* sections (constants, strings, etc.) */
*(.glue_7)                /* glue arm to thumb code */
*(.glue_7t)               /* glue thumb to arm code */
*(.eh_frame)
KEEP (*(.init))
KEEP (*(.fini))
. = ALIGN(4);
} > FLASH_1

.ARM.extab :
{
*(.ARM.extab* .gnu.linkonce.armextab.*)
} > FLASH_1

.ARM :
{
__exidx_start = .;
*(.ARM.exidx*)
__exidx_end = .;
} > FLASH_1 .ctors :
{
__CTOR_LIST__ = .;
/* gcc uses crtbegin.o to find the start of
the constructors, so we make sure it is
first. Because this is a wildcard, it

```

```

doesn't matter if the user does not
actually link against crtbegin.o; the
linker won't look for a file to match a
wildcard. The wildcard also means that it
doesn't matter which directory crtbegin.o
is in.  */
KEEP (*crtbegin.o(.ctors))
KEEP (*crtbegin?.o(.ctors))
/* We don't want to include the .ctor section from
the crtend.o file until after the sorted ctors.
The .ctor section from the crtend file contains the
end of ctors marker and it must be last */
KEEP (*(EXCLUDE_FILE(*crtend?.o*crtend.o) .ctors))
KEEP (*(SORT(.ctors.*)))
KEEP (*(.ctors))
__CTOR_END__ = .;
} > FLASH_1

.dtors :
{
    __DTOR_LIST__ = .;
    KEEP (*crtbegin.o(.dtors))
    KEEP (*crtbegin?.o(.dtors))
    KEEP (*(EXCLUDE_FILE(*crtend?.o *crtend.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
    __DTOR_END__ = .;
} > FLASH_1

.preinit_array :
{
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array*))
    PROVIDE_HIDDEN (__preinit_array_end = .);
} > FLASH_1

.init_array :
{
    PROVIDE_HIDDEN
    (__init_array_start = .);
    KEEP (*(SORT(.init_array.*)))
    KEEP (*(.init_array*))
    PROVIDE_HIDDEN (__init_array_end = .);
} > FLASH_1

.fini_array :
{
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP(*(SORT(.fini_array.*)))
    KEEP (*(.fini_array*))
    PROVIDE_HIDDEN (__fini_array_end = .);
} > FLASH_1

__etext = .;      /* define a global symbol at end of code */
__DATA_ROM = .; /* Symbol is used by startup for data initialization */

/* reserve MTB memory at the beginning of SRAM */
.mtb : /* MTB buffer address as defined by the hardware */
{
    . = ALIGN(8);

```

```

_mtb_start = .;
KEEP(*(.mtb_buf)) /* need to KEEP Micro Trace Buffer as not referenced by application */
. = ALIGN(8);
_mtb_end = .;
} > SRAM

.data : AT(__DATA_ROM)
{
. = ALIGN(4);
__DATA_RAM = .;
__data_start__ = .; /* create a global symbol at data start */
_sdata = .; /* create a global symbol at data start */
*(.data) /* .data sections */
*(.data*) /* .data* sections */
KEEP(*(.jcr*))
. = ALIGN(4);
__data_end__ = .; /* define a global symbol at data end */
_edata = .; /* define a global symbol at data end */
} > SRAM

/* Symbol is used by startup for data initialization */
__DATA_END = __DATA_ROM + (__data_end__ - __data_start__);
__data_size = _edata - _sdata;

/*Uninitialized data section */
.bss :
{
/* This is used by the startup in order to initialize the .bss section */
. = ALIGN(4);
__START_BSS = .;
__bss_start__ = .;
*(.bss)
*(.bss*)
*(COMMON)
. = ALIGN(4);
__bss_end__ = .;
__END_BSS = .;
} > SRAM

_romp_at = __DATA_ROM + SIZEOF(.data);
.romp : AT(_romp_at)
{
__S_romp = _romp_at;
LONG(__DATA_ROM);
LONG(_sdata);
LONG(__data_size);
LONG(0);
LONG(0);
LONG(0);
} > SRAM

.heap :
{
. = ALIGN(8);
__end__ = .;
_end = .;
PROVIDE(end = .);
__HeapBase = .;
. += HEAP_SIZE;
__HeapLimit = .;

```

```
} > SRAM

.stack :
{
. = ALIGN(8);
. += STACK_SIZE;
} > SRAM

__StackTop    = .;
__StackLimit = __StackTop - STACK_SIZE;
PROVIDE(__stack = __StackTop);
.ARM.attributes 0 : { *(.ARM.attributes) }
}
```

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.

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 μ Vision 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.

© 2017 NXP B.V.

