TrueSTUDIO® Success

Working with bootloaders on Cortex-M devices

atollic

atollic

What is a bootloader?

General definition:

"A boot loader is a computer program that loads the main operating system or runtime environment for the computer after completion of the self-tests." - Wikipedia

In microcontroller land (ARM Cortex-M0/M3/M4/M7):

"A bootloader enriches the capabilities of the microcontroller and makes it a self-programmable device"

This is our definition!

atollic

Why use a bootloader?

Enables a device/product to upgrade itself in the field.

- Firmware is rarely bug free! – Need a method to upgrade a product's firmware when defects are found.
- New requirements – Need a method to upgrade a product's firmware due to new functionality.

A product recall might not be a feasible option!

Coverage

This document will cover the area of bootloaders from the perspective of Atollic TrueSTUDIO on ARM Cortex-M devices.

- Constructing and building the bootloader.
- Constructing and building the main application.
- Interaction between the bootloader and the main application.
- Use cases for debugging the above.

atollic

Coverage

It will not cover the actual self-update feature (downloading and flash reprogramming of the main application)

Many methods exists and are highly application specific!

- Update via USB, USART, CAN, SPI, ...
- Device capabilities.
- Device vendor support libraries.
- Etc.

Example hardware & code

STM32-F4-Discovery kit
from STMicroelectronics

Download the example projects with ready-made code from TrueSTORE (inside TrueSTUDIO).

- File → New → Download new example project from TrueSTORE → STMicroelectronics → STM32F4-Discovery →
  - STM32F4_Discovery_Bootloader_APP
  - STM32F4_Discovery_Bootloader_BL

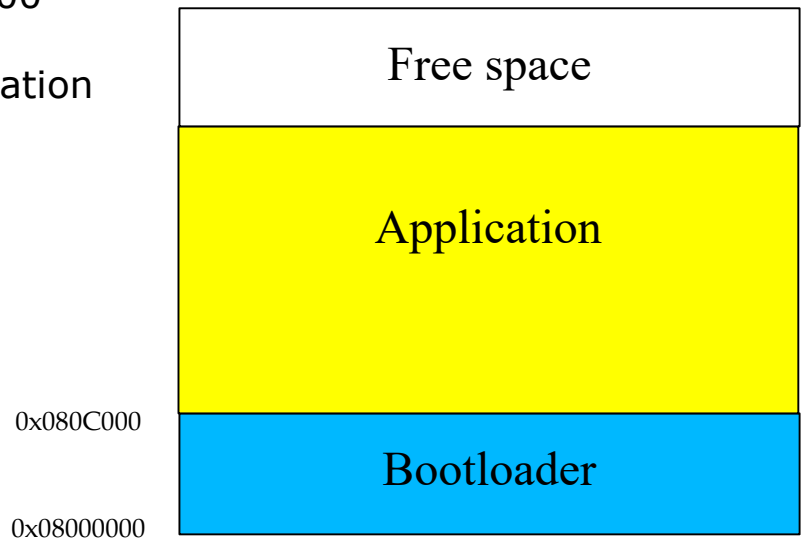
BL = Bootloader. APP = Application. Both must be downloaded.

Memory map

- Bootloader and application separated in flash
  (Important! Separated by flash pages!)

Our example:

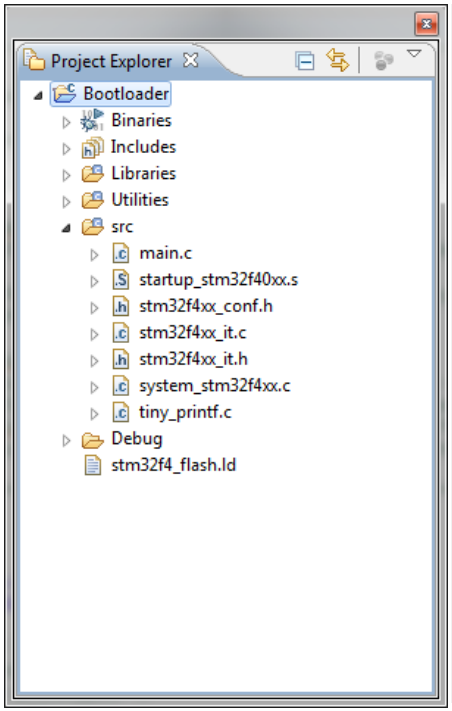
- First 3 16K flash pages allocated to the bootloader.
  - 0x08000000 - 0x0800C000
  - Reset vector located at 0x08000000
- Rest of the flash allocated to the application
  - 0x0800C000 -



atollic

Constructing the bootloader

- Start with a project template generated by the project wizard. (STM32F4-Discovery board, code in flash memory.)
- Linker configuration file (stm32f4_flash.ld) will, by default, place the code at the start of flash at 0x08000000. This is what we want!



stm32f4_flash.ld

```
ENTRY(Boot_Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x20020000;    /* end of 128K RAM */

/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0;      /* required amount of heap  */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 1024K
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 128K
  MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
  CCMRAM (rw)     : ORIGIN = 0x10000000, LENGTH = 64K
}

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

```

atollic

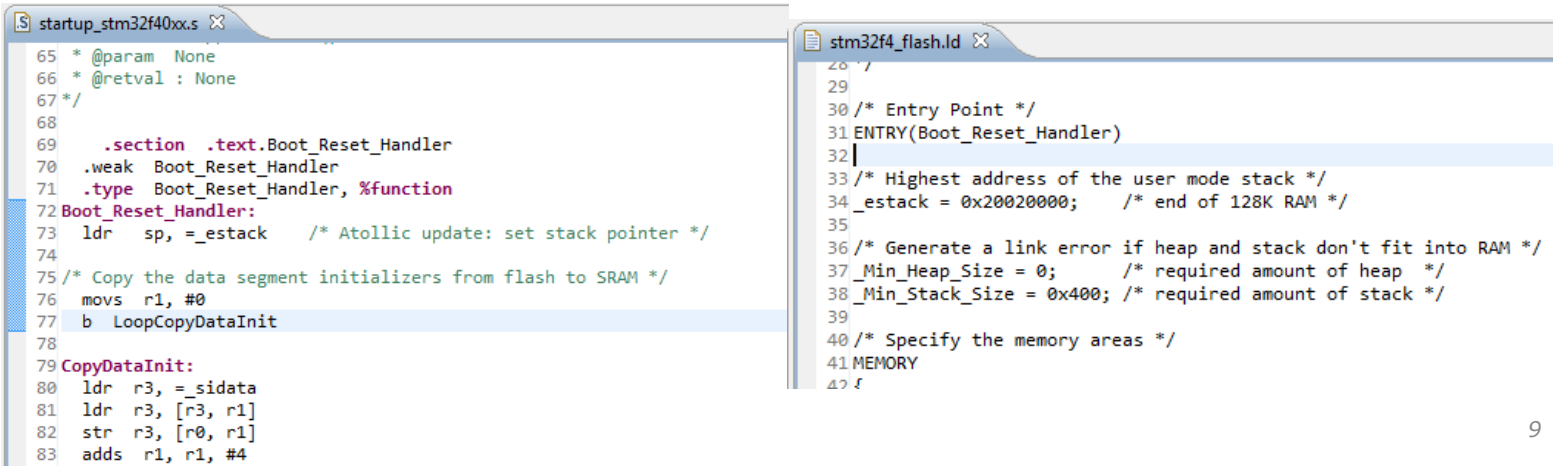
Constructing the bootloader

To avoid confusing the debugger when debugging the bootloader and the application in the same debug session:

- Use different symbolic names for critical functions
- Simplifies breakpoint handling etc.

For instance the entry point for the bootloader will be the **Reset_Handler** function by default. You probably have another version with the same name in the application also!

- Rename **Reset_Handle**r to **Boot_Reset_Handler** and update all references (7 in startup_stm32F40xx.s and 1 in stm32f4_flash.ld)!
- Rename **main()** to **boot_main()** and update all references!



A basic boot_main()

- Performs a firmware update if requested (not implemented in this example!).
- Sets up the environment for the application:
  - Locates and sets the application stack pointer address. (Stored at first entry in application vector table.)
  - Locates the application entry point. (Stored at second entry in application vector table.)
  - Configures the vector table offset register. (Exceptions/IRQ now finds its handlers here!)
  - Starts the application.

atollic

A basic boot_main()

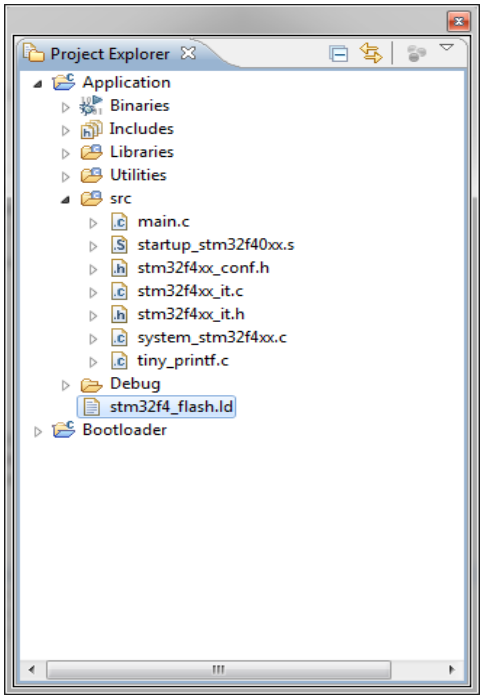
main.c

```
/* Application start address */
#define APPLICATION_ADDRESS        0x0800C000

typedef void (*pFunction)(void);

/**
**===========================================================================
**  Abstract: Bootloader
**===========================================================================
*/
int boot_main(void)
{

    pFunction appEntry;
    uint32_t appStack;

    /* Check if firmware update required */
    if(checkFirmwareUpdate()){

        /* Perform the update */
        performFirmwareUpdate();

    }

    /* Get the application stack pointer (First entry in the application vector table) */
    appStack = (uint32_t) *((__IO uint32_t*)APPLICATION_ADDRESS);

    /* Get the application entry point (Second entry in the application vector table) */
    appEntry = (pFunction) *(__IO uint32_t*) (APPLICATION_ADDRESS + 4);

    /* Reconfigure vector table offset register to match the application location */
    SCB->VTOR = APPLICATION_ADDRESS;

    /* Set the application stack pointer */
    __set_MSP(appStack);

    /* Start the application */
    appEntry();

    while(1);

}
```

atollic

Constructing the application

- Start with a project template generated by the project wizard. (STM32F4-Discovery board, code in flash memory.)
- Linker configuration file (stm32f4_flash.ld) will, by default, place the code at the start of flash at 0x08000000 → **0x0800C000**. We need to change this! You may also want to reduce LENGTH of FLASH 1024K- 3x16K = **976K**



stm32f4_flash.ld

```
30 /* Entry Point */
31 ENTRY(Reset_Handler)
32
33 /* Highest address of the user mode stack */
34 _estack = 0x20020000;    /* end of 128K RAM */
35
36 /* Generate a link error if heap and stack don't fit into RAM */
37 _Min_Heap_Size = 0;      /* required amount of heap  */
38 _Min_Stack_Size = 0x400; /* required amount of stack */
39
40 /* Specify the memory areas */
41 MEMORY
42 {
43   FLASH (rx)      : ORIGIN = 0x0800C000, LENGTH = 1024K
44   RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 128K
45   MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
46   CCMRAM (rw)     : ORIGIN = 0x10000000, LENGTH = 64K
47 }
48
49 /* Define output sections */
50 SECTIONS
51 {
52   /* The startup code goes first into FLASH */
53   .isr_vector :
54   {
55     . = ALIGN(4);
56     KEEP(*(.isr_vector)) /* Startup code */
57     . = ALIGN(4);
58   } >FLASH
59
```

atollic

Constructing the application

NOTE!

In this example we have chosen to let the bootloader set up the basic environment (stack pointer, vector table, etc.) for the application.

Lookout for any application code that might circumvent this behavior!

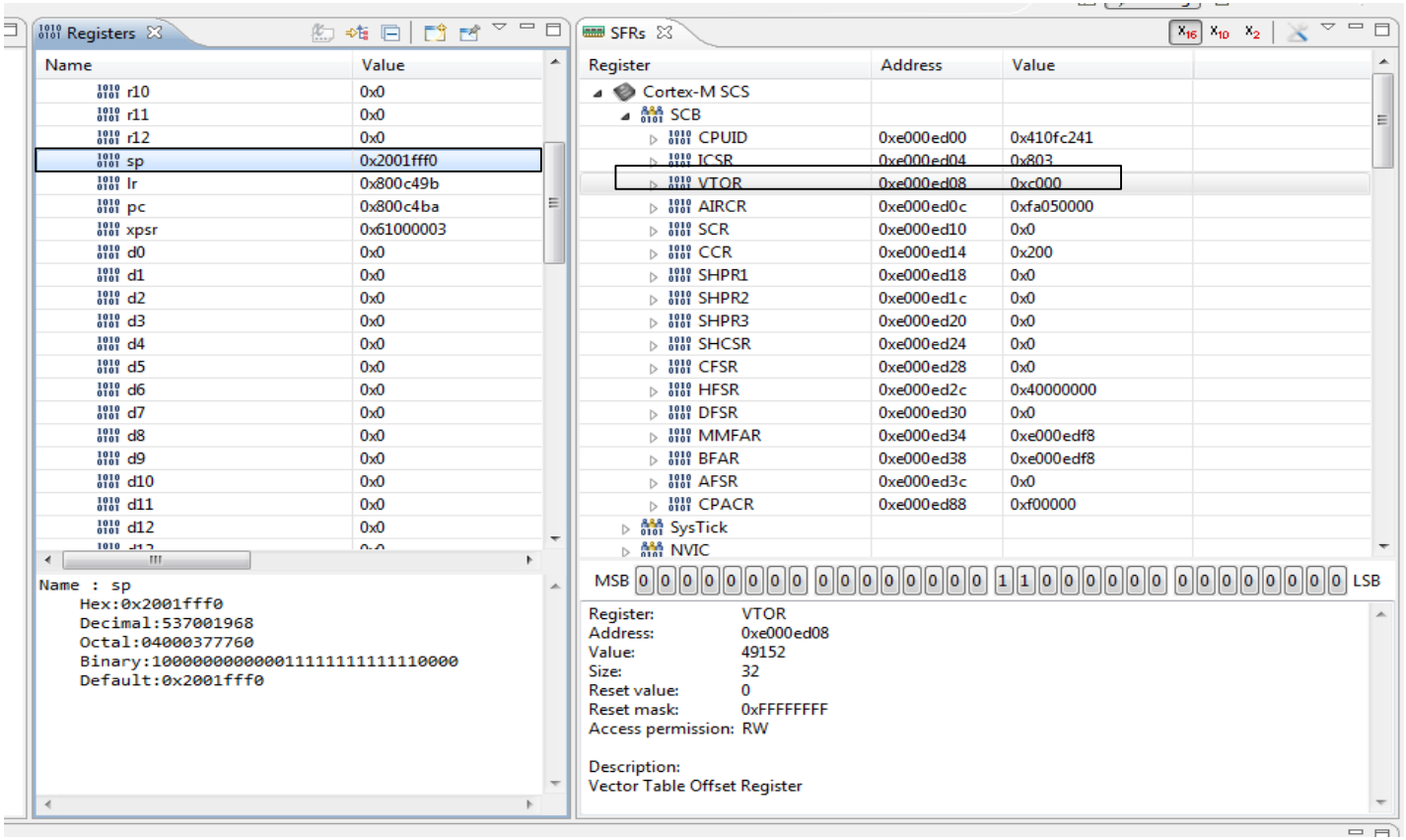
For instance the “SystemInit” in this example:

```
#ifdef VECT_TAB_SRAM

  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in
Internal SRAM */

#else

  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in
Internal FLASH */

#endif
```

atollic

Constructing the application

Always a good idea to verify!



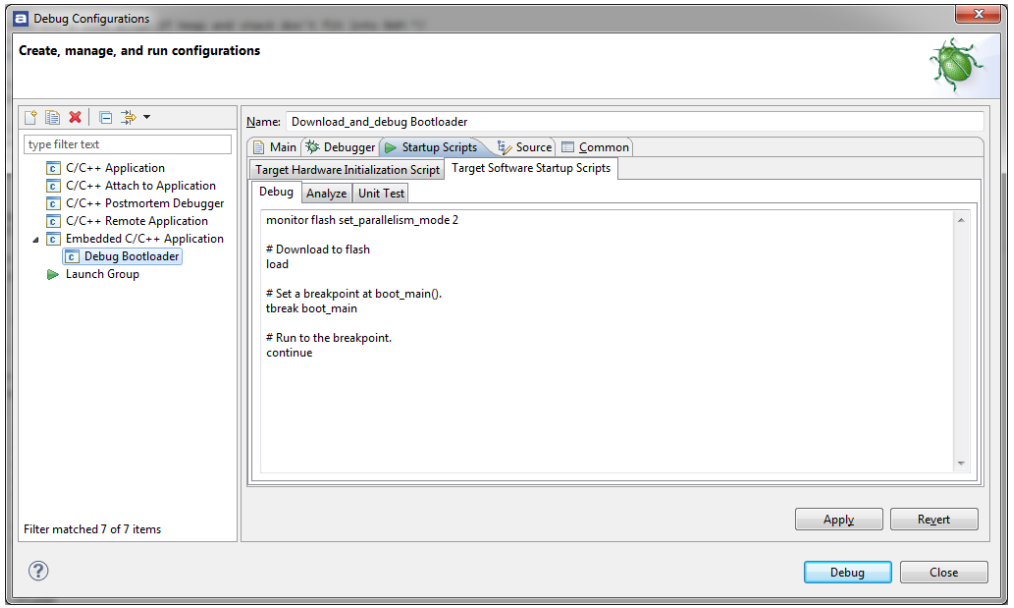
Putting it all together

Use cases during development

- Debugging the bootloader
- Debugging the application
- Debugging the bootloader & application
  - Application programmed by debugger
  - Application programmed by bootloader

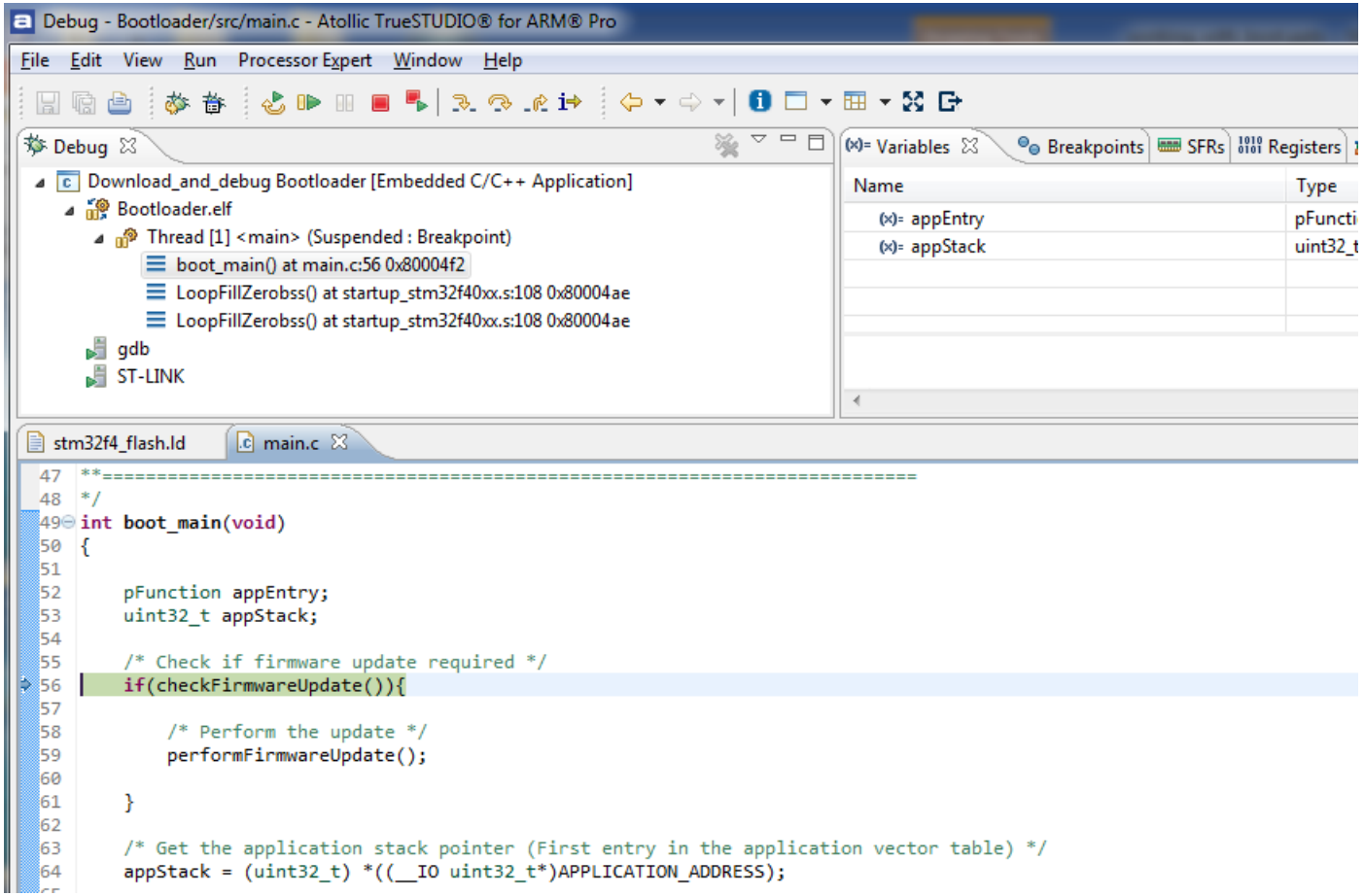
Debugging the bootloader

- Create a debug configuration, **Download_and_debug_Bootloader**, for the Bootloader project.
- Edit the debug startup script and instruct the debugger to set a breakpoint at **boot_main**. (or **Boot_Reset_Handler** if debugging prior boot_main)
- Start the debugger!



atollic

Debugging the bootloader



atollic

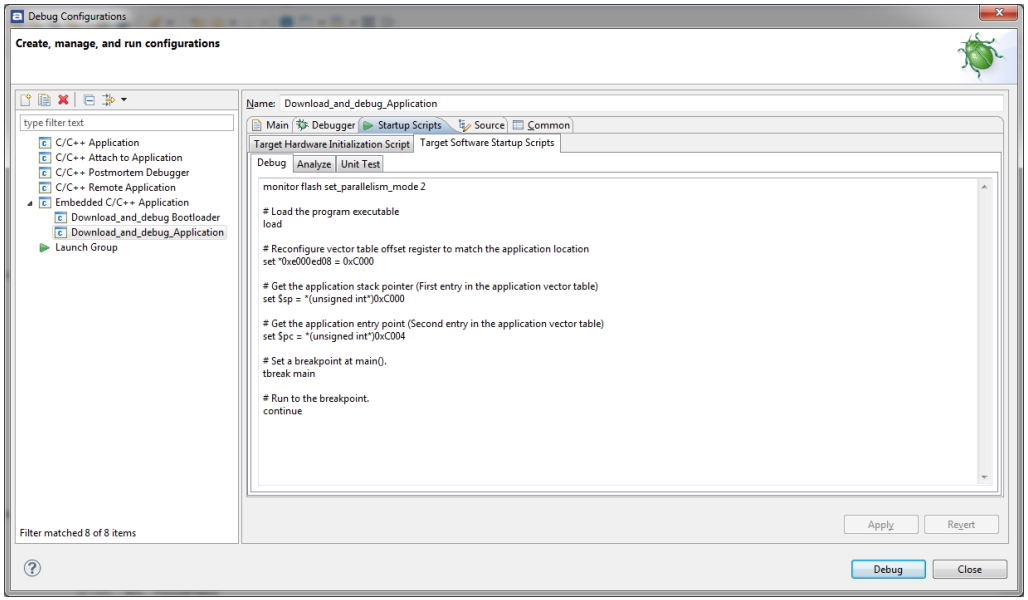
Debugging the bootloader

No source level debugging after branching off to the application (**appEntry()**)

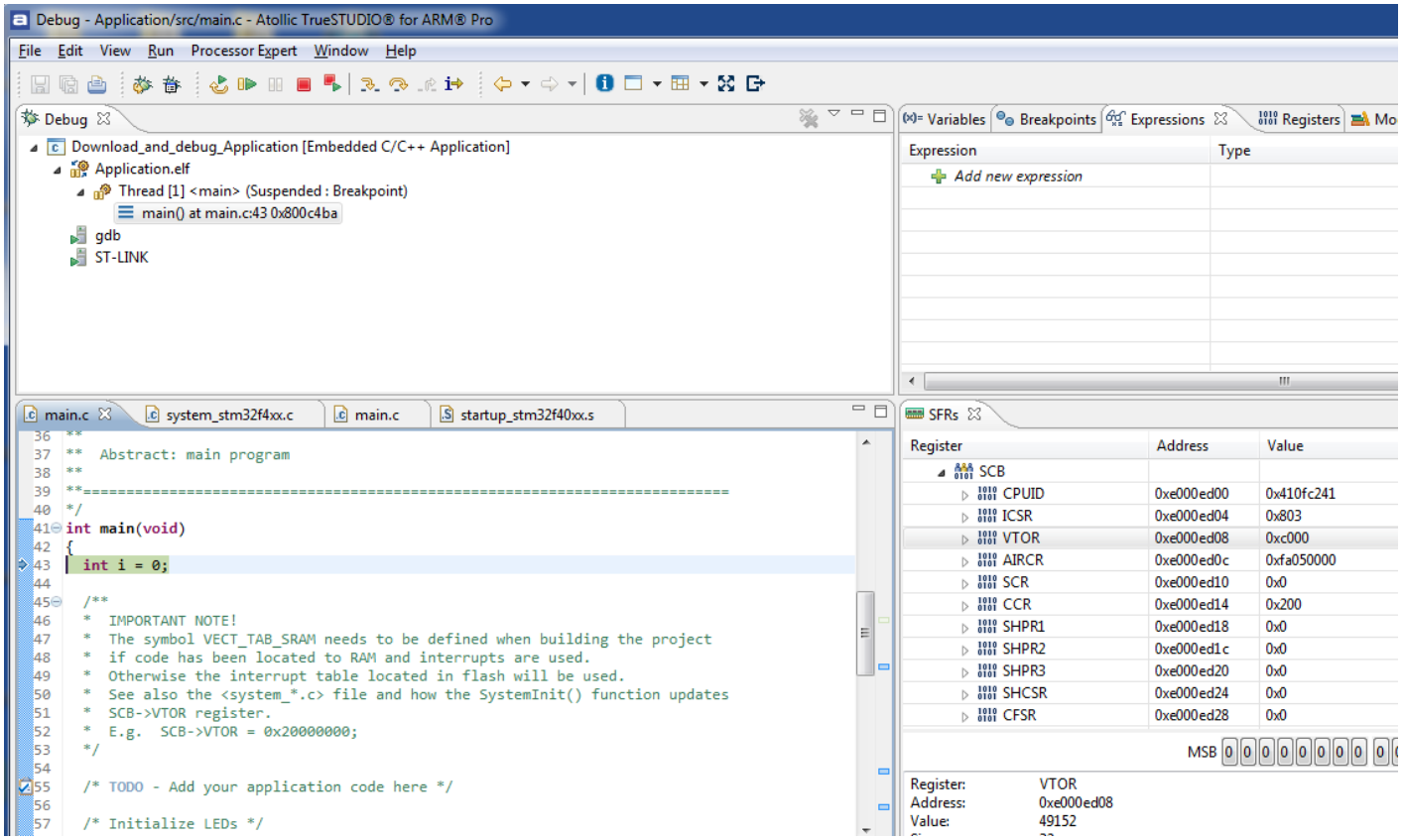
More on that later!

Debugging the application

- Create a debug configuration, **Download_and_debug_Application**, for the Application project.
- The vector table offset register, stack pointer and program counter are setup just to be on the safe side! Might already be done by debugger or application startup code!
- Start the debugger!



Debugging the application



Debugging the application and bootloader

In order to do source-level debugging through both the bootloader and the application project “at the same time” some configuration is needed.

The debugger needs to have information regarding both of the ELF files!

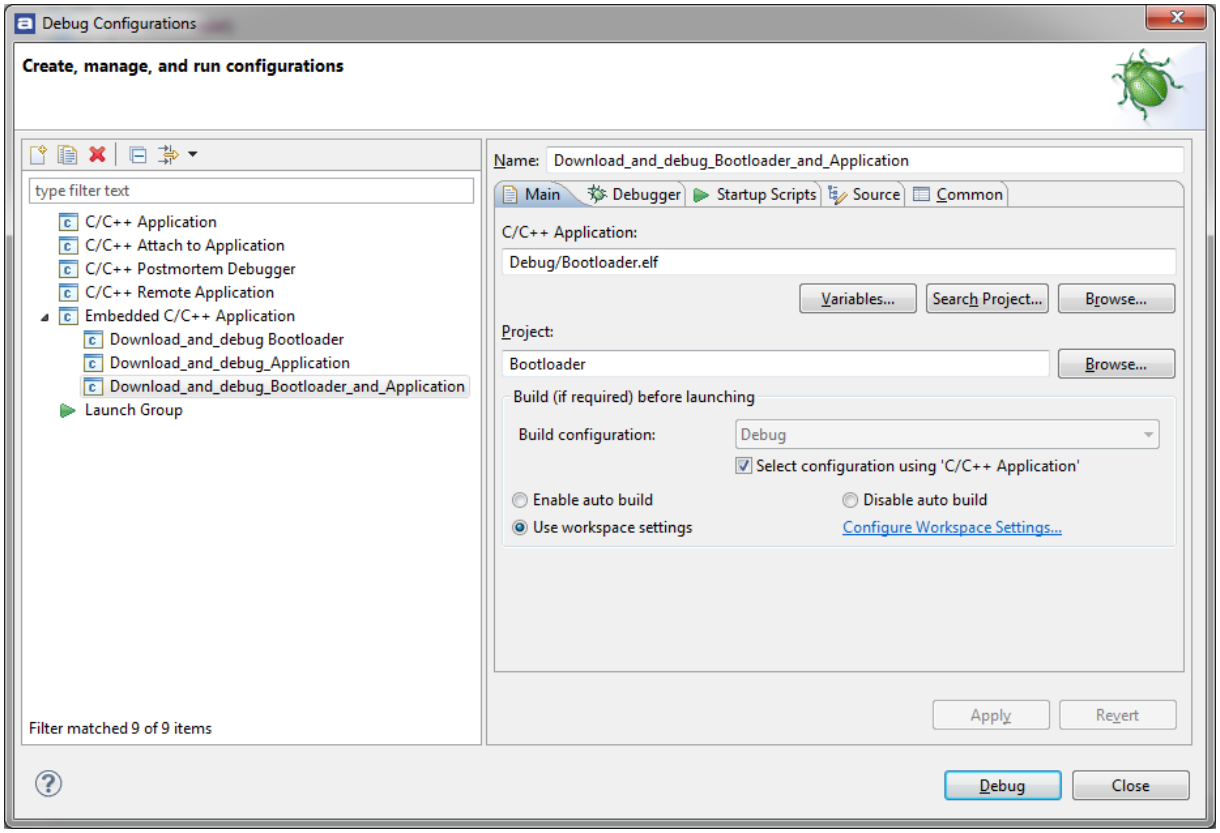
atollic

Debugging the application and bootloader - use case 1

Both the bootloader and application binary are to be programmed during a debug launch. Source-level debugging should work after the “jump”.

Debugging the application and bootloader – use case 1

Create a third debug configuration connected to the Bootloader project.



atollic

Debugging the application and bootloader – use case 1

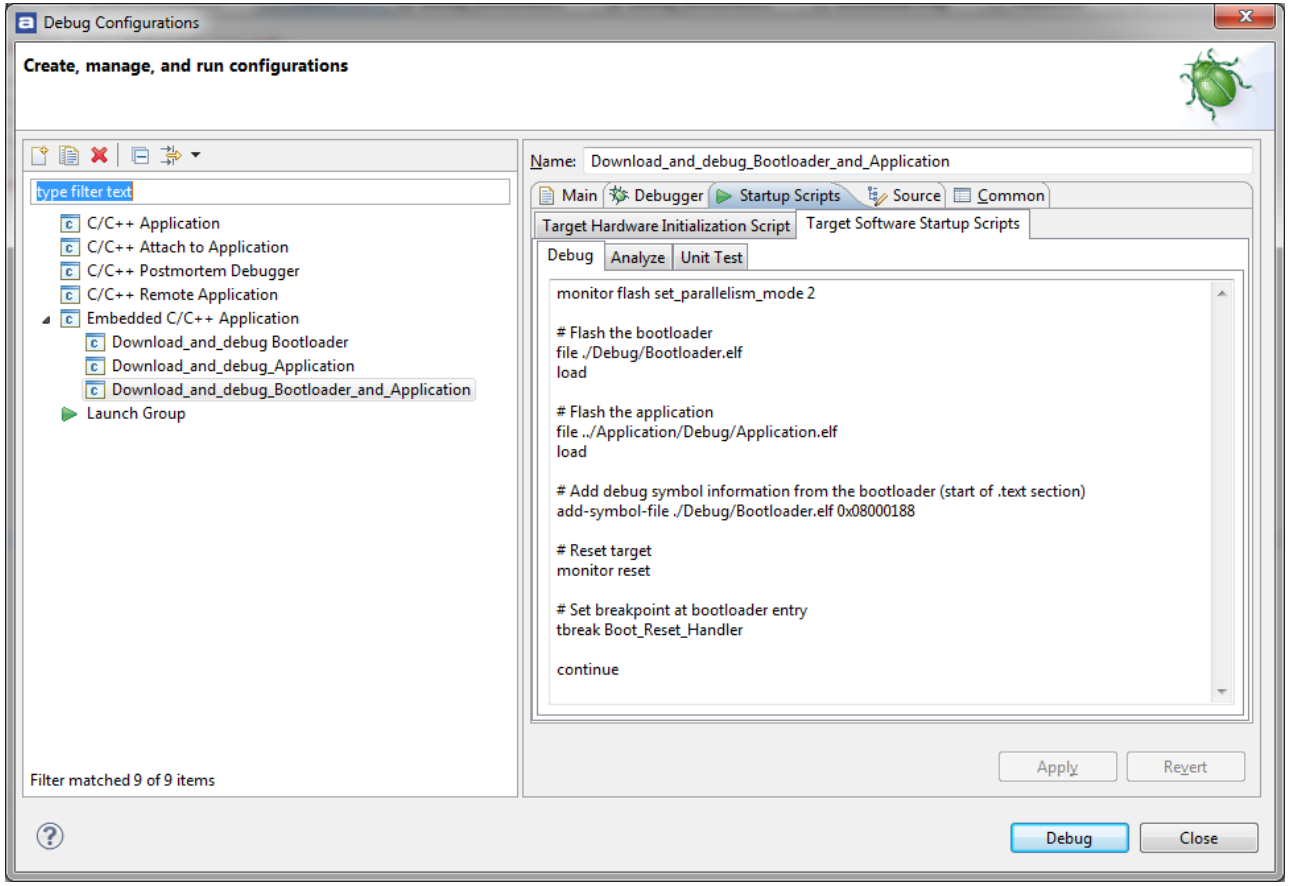
Edit the debugger startup script and instruct the debugger to perform the following:

- Program the bootloader binary.
- Program the application binary.
- Since the application binary in this case was programmed last we need to re-add the symbolic information for the bootloader binary.

The debugger is now aware of both the binaries!

- Reset target.
- Set breakpoint at bootloader entry and start execution.

Debugging the application and bootloader – use case 1



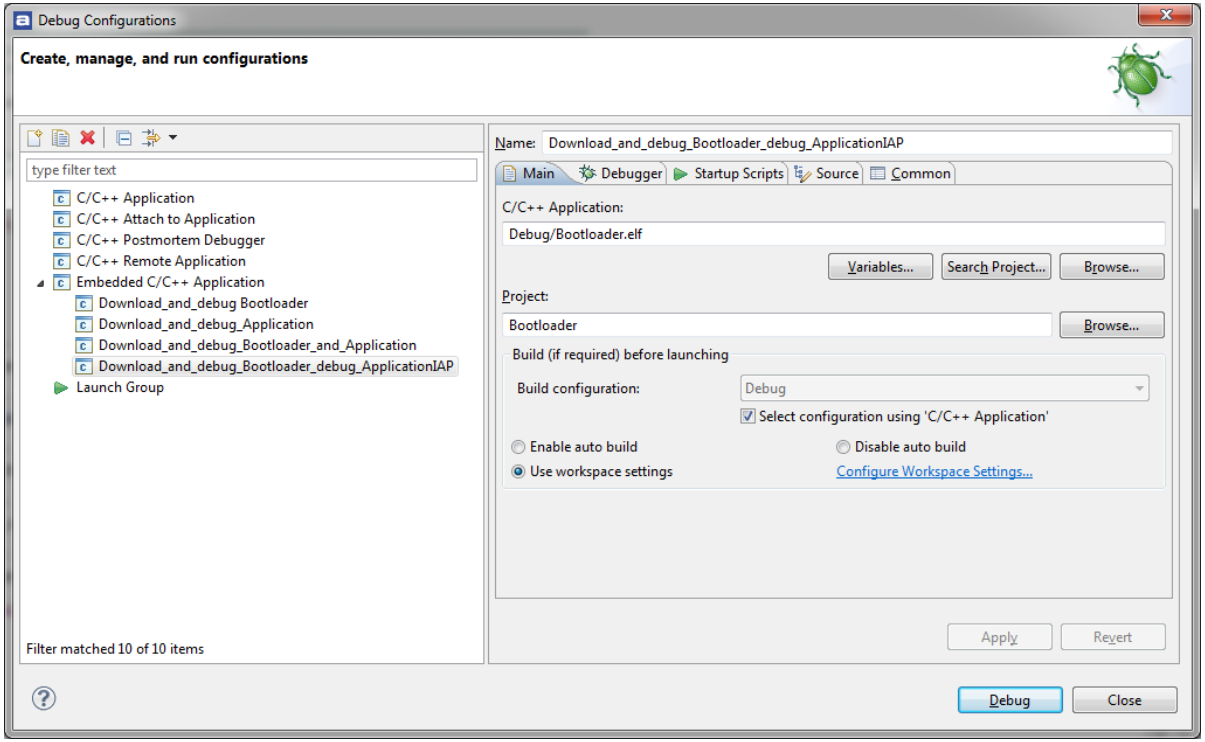
atollic

Debugging the application and bootloader - use case 2

Only the bootloader is to be programmed during a debug launch. The application is programmed by the bootloader (IAP). Source-level debugging should work after the “jump”.

Debugging the application and bootloader – use case 2

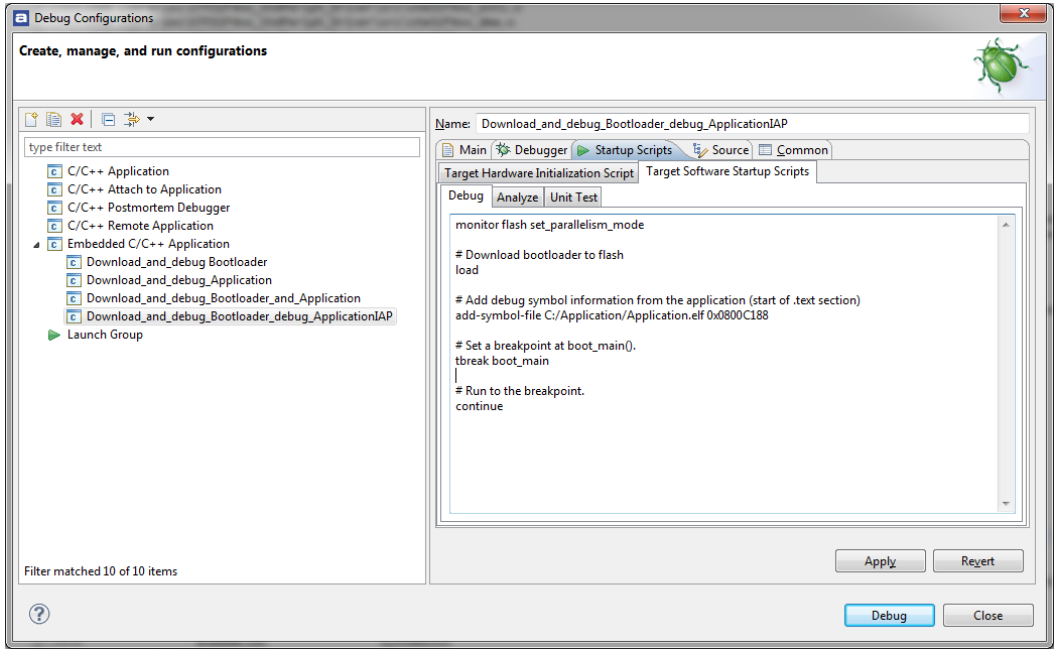
Create a fourth debug configuration connected to the Bootloader project. Use the "Download_and_debug Bootloader" configuration as template.



Debugging the application and bootloader – use case 2

Edit the debugger startup script with the following change:

- Add symbolic/debug information from the corresponding application's ELF file.



More information:

www.atollic.com

EUROPE & WORLDWIDE

Atollic AB

Science Park

Gjuterigatan 7

SE-553 18 Jönköping

Sweden

+46 36 19 60 50

info@atollic.com

USA & AMERICAS

Atollic Inc.

241 Boston Post Road West

Marlborough, Massachusetts 01752

+1 (617) 674-2655

sales.usa@atollic.com