



TrueSTUDIO[®] Success

Working with bootloaders on Cortex-M devices

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!

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.

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.

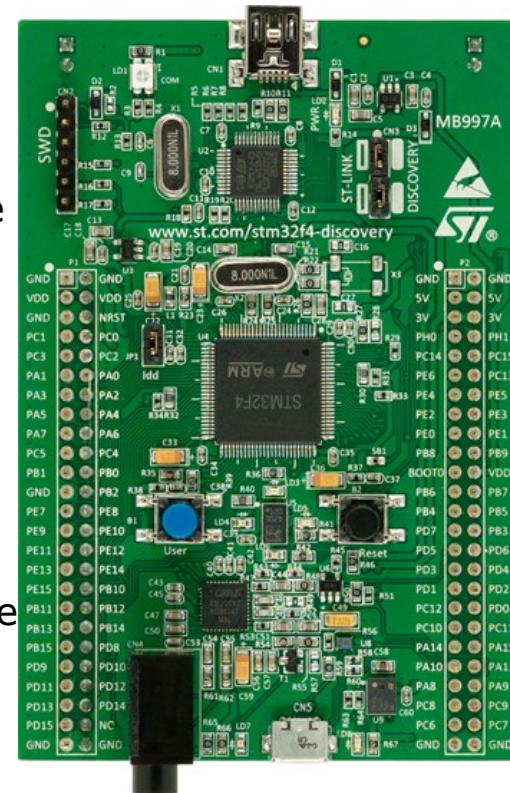
Example hardware & code

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.

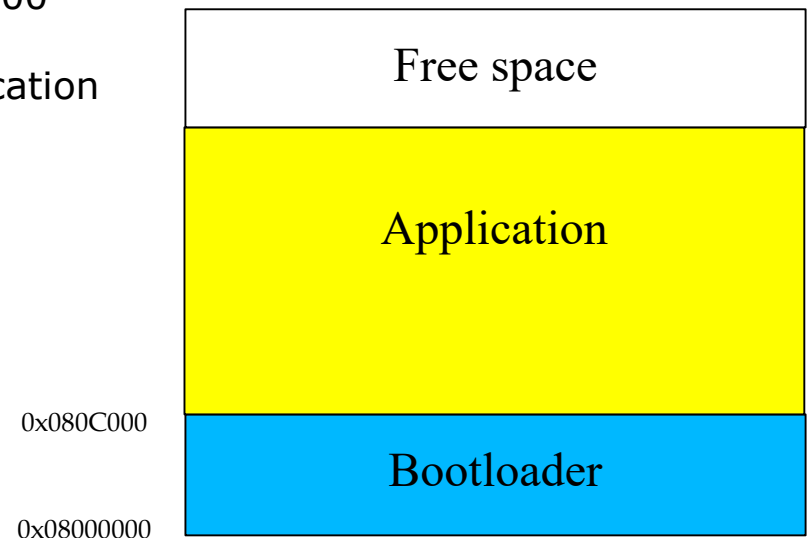


Memory map

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

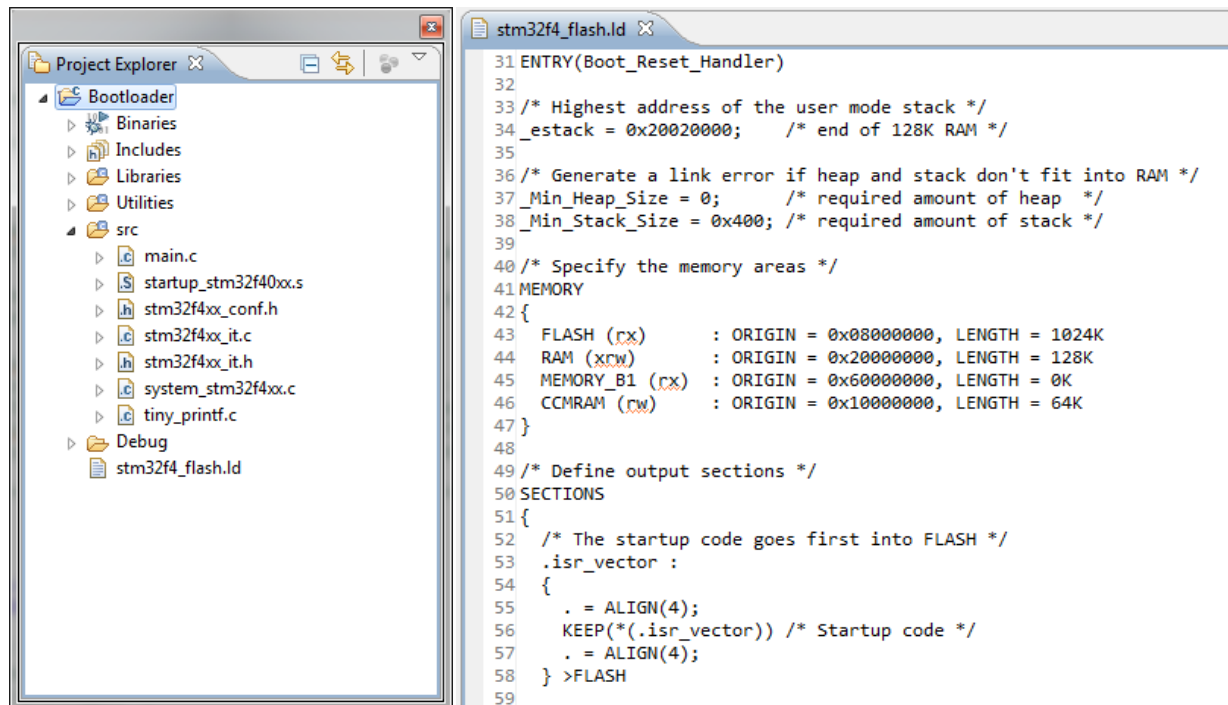
Our example:

- 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 -



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!



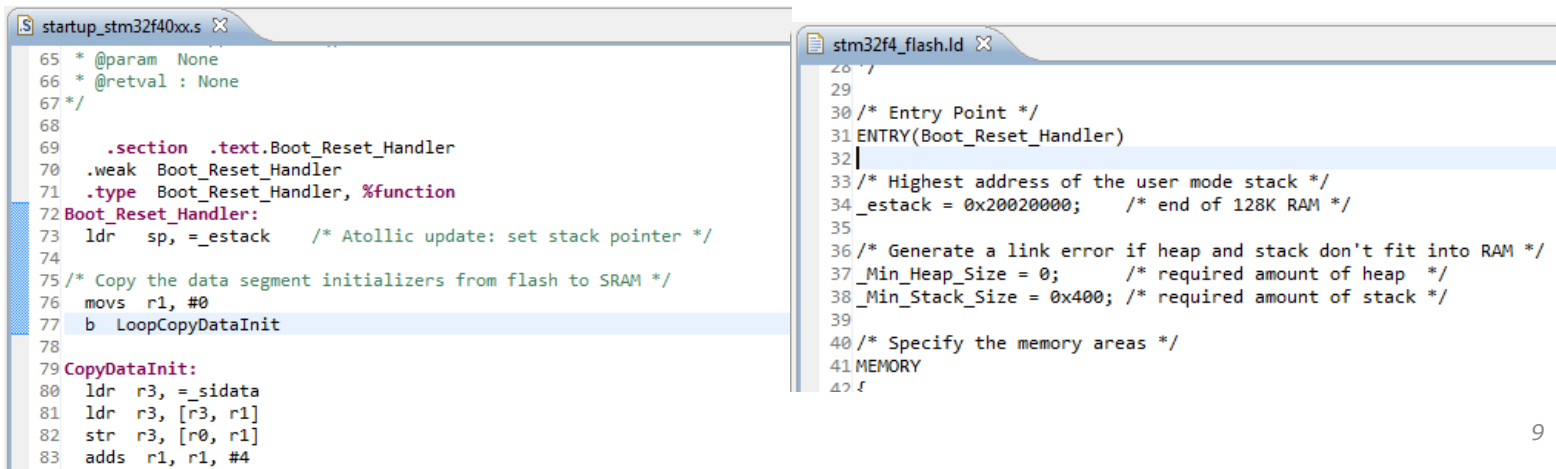
Constructing the bootloader

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_Handler** 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!



The screenshot shows two code editors side-by-side. The left editor, titled 'startup_stm32f40xx.s', displays assembly code. Lines 65-67 show comments for '@param' and '@retval'. Lines 69-71 define a section '.text.Boot_Reset_Handler' with a weak symbol 'Boot_Reset_Handler' of type 'function'. Lines 72-74 show the 'Boot_Reset_Handler' function, which sets the stack pointer 'sp' to '_estack'. Lines 75-77 show a comment and code for copying data segment initializers from flash to SRAM. Lines 79-83 show the 'CopyDataInit' function, which loads the start address of the data segment and increments the pointer. The right editor, titled 'stm32f4_flash.ld', displays a linker script. Lines 26-29 show comments for the entry point. Lines 30-32 show the 'ENTRY(Boot_Reset_Handler)' directive. Lines 33-35 show comments for the highest address of the user mode stack and the end of 128K RAM. Lines 36-39 show comments for generating a link error if heap and stack don't fit into RAM, and the required amount of heap and stack. Lines 40-41 show comments for specifying the memory areas and the 'MEMORY' directive.

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.

A basic boot_main()

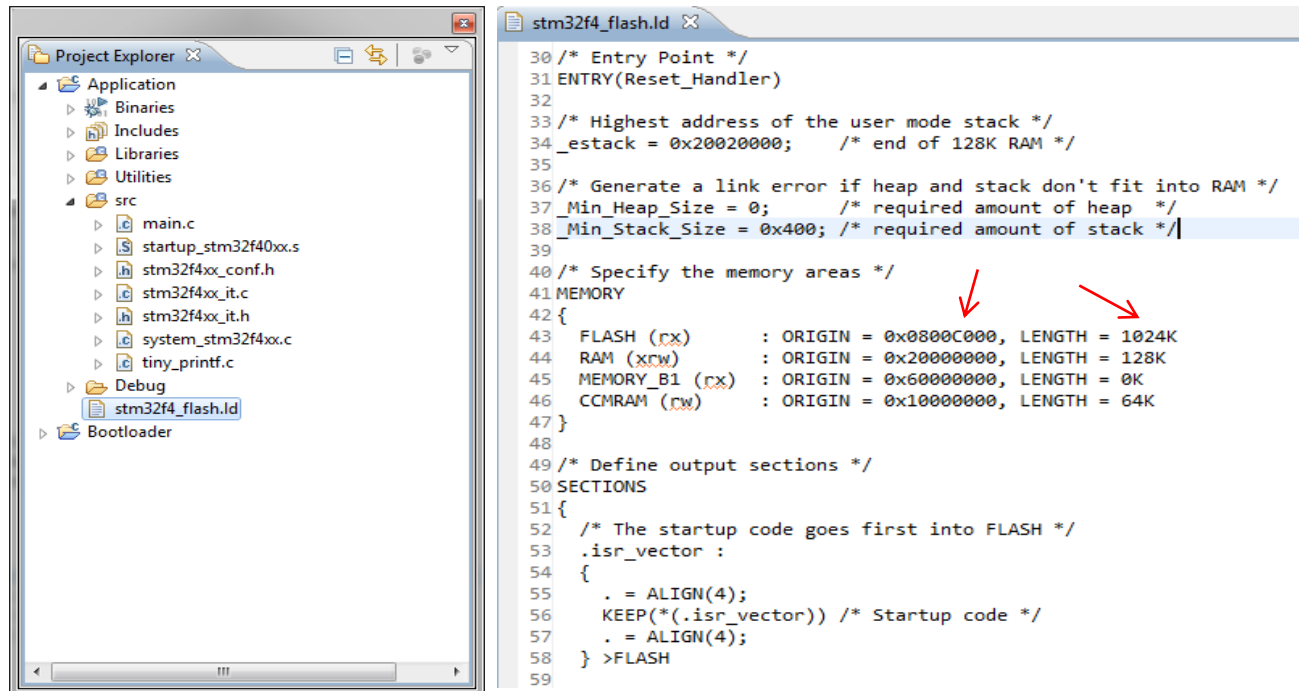
```

37
38
39 /* Application start address */
40 #define APPLICATION_ADDRESS    0x0800C000
41
42 typedef void (*pFunction)(void);
43
44 /**
45  **=====
46  ** Abstract: Bootloader
47  **=====
48  */
49 int boot_main(void)
50 {
51
52     pFunction appEntry;
53     uint32_t appStack;
54
55     /* Check if firmware update required */
56     if(checkFirmwareUpdate()){
57
58         /* Perform the update */
59         performFirmwareUpdate();
60
61     }
62
63     /* Get the application stack pointer (First entry in the application vector table) */
64     appStack = (uint32_t) *((__IO uint32_t*)APPLICATION_ADDRESS);
65
66     /* Get the application entry point (Second entry in the application vector table) */
67     appEntry = (pFunction) *(__IO uint32_t*) (APPLICATION_ADDRESS + 4);
68
69     /* Reconfigure vector table offset register to match the application location */
70     SCB->VTOR = APPLICATION_ADDRESS;
71
72     /* Set the application stack pointer */
73     __set_MSP(appStack);
74
75     /* Start the application */
76     appEntry();
77
78     while(1);
79
80 }

```

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**



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 */
```

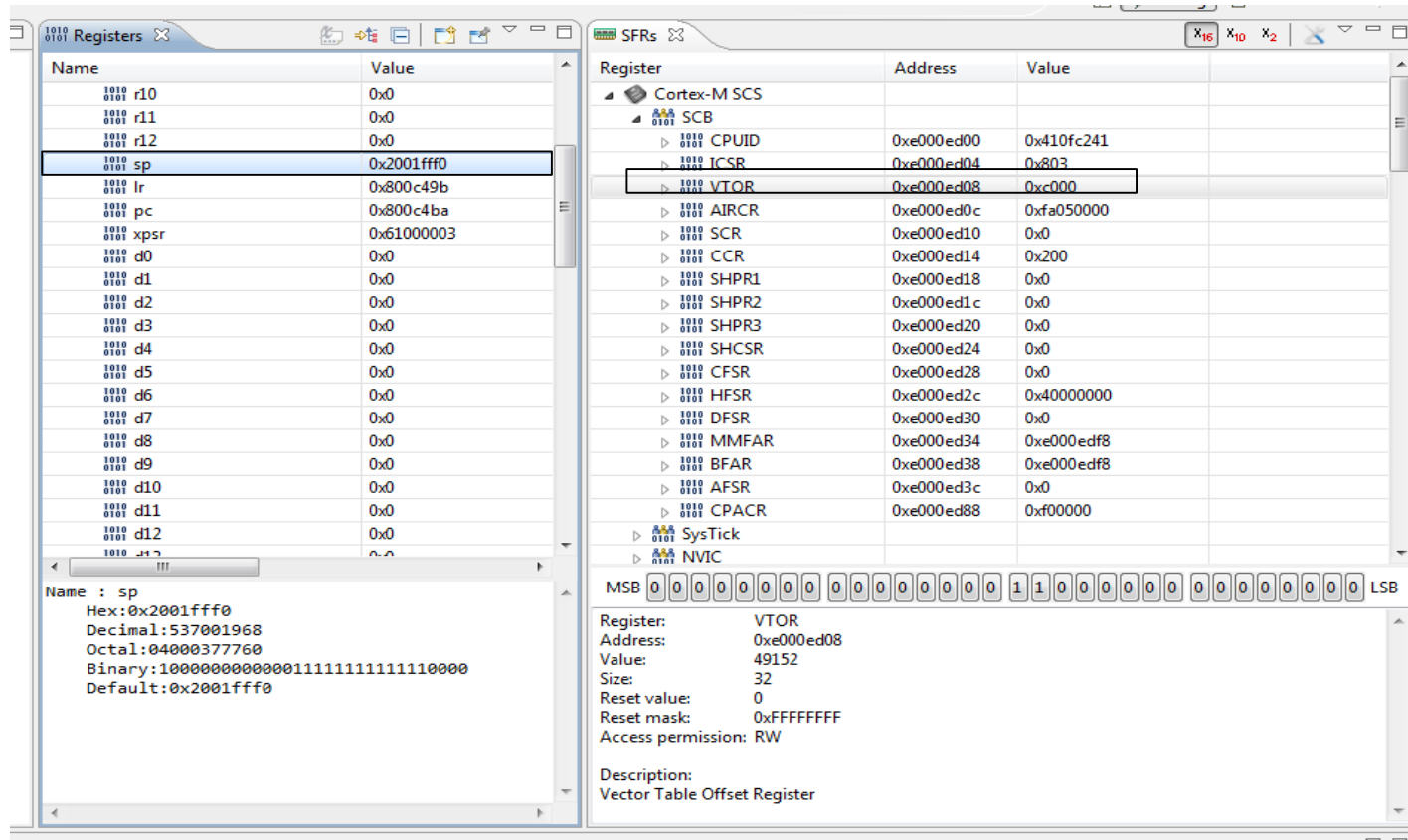
```
#else
```

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

```
#endif
```

Constructing the application

Always a good idea to verify!



The screenshot displays two windows from the Atollic IDE:

- Registers Window:** A table listing various registers and their values. The 'sp' register is highlighted with a value of 0x2001fff0.
- SFRs Window:** A table listing System Function Registers (SFRs) for the Cortex-M SCS. The 'VTOR' register is highlighted with an address of 0xe000ed08 and a value of 0xc000.

Register Details (from Registers window):

Name	Value
r10	0x0
r11	0x0
r12	0x0
sp	0x2001fff0
lr	0x800c49b
pc	0x800c4ba
xpsr	0x6100003
d0	0x0
d1	0x0
d2	0x0
d3	0x0
d4	0x0
d5	0x0
d6	0x0
d7	0x0
d8	0x0
d9	0x0
d10	0x0
d11	0x0
d12	0x0

SFR Details (from SFRs window):

Register	Address	Value
CPUID	0xe000ed00	0x410fc241
ICSR	0xe000ed04	0x803
VTOR	0xe000ed08	0xc000
AIRCR	0xe000ed0c	0xfa050000
SCR	0xe000ed10	0x0
CCR	0xe000ed14	0x200
SHPR1	0xe000ed18	0x0
SHPR2	0xe000ed1c	0x0
SHPR3	0xe000ed20	0x0
SHCSR	0xe000ed24	0x0
CFSR	0xe000ed28	0x0
HFSR	0xe000ed2c	0x40000000
DFSR	0xe000ed30	0x0
MMFAR	0xe000ed34	0xe000edf8
BFAR	0xe000ed38	0xe000edf8
AFSR	0xe000ed3c	0x0
CPACR	0xe000ed88	0xf00000

VTOR Register Details:

- Register: VTOR
- Address: 0xe000ed08
- Value: 49152
- Size: 32
- Reset value: 0
- Reset mask: 0xFFFFFFFF
- Access permission: RW
- Description: Vector Table Offset Register

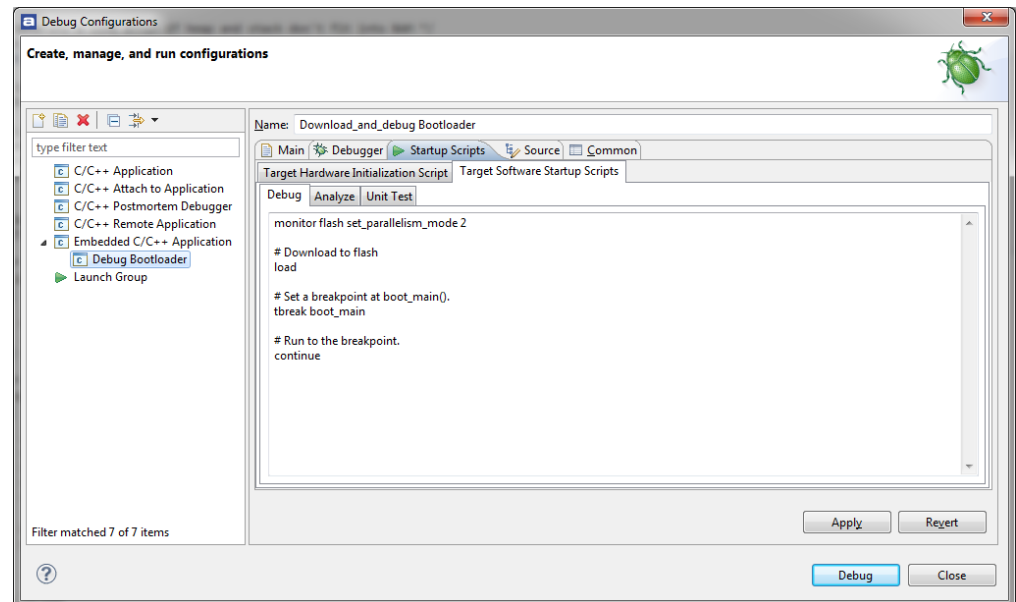
Putting it all together

Use cases during development

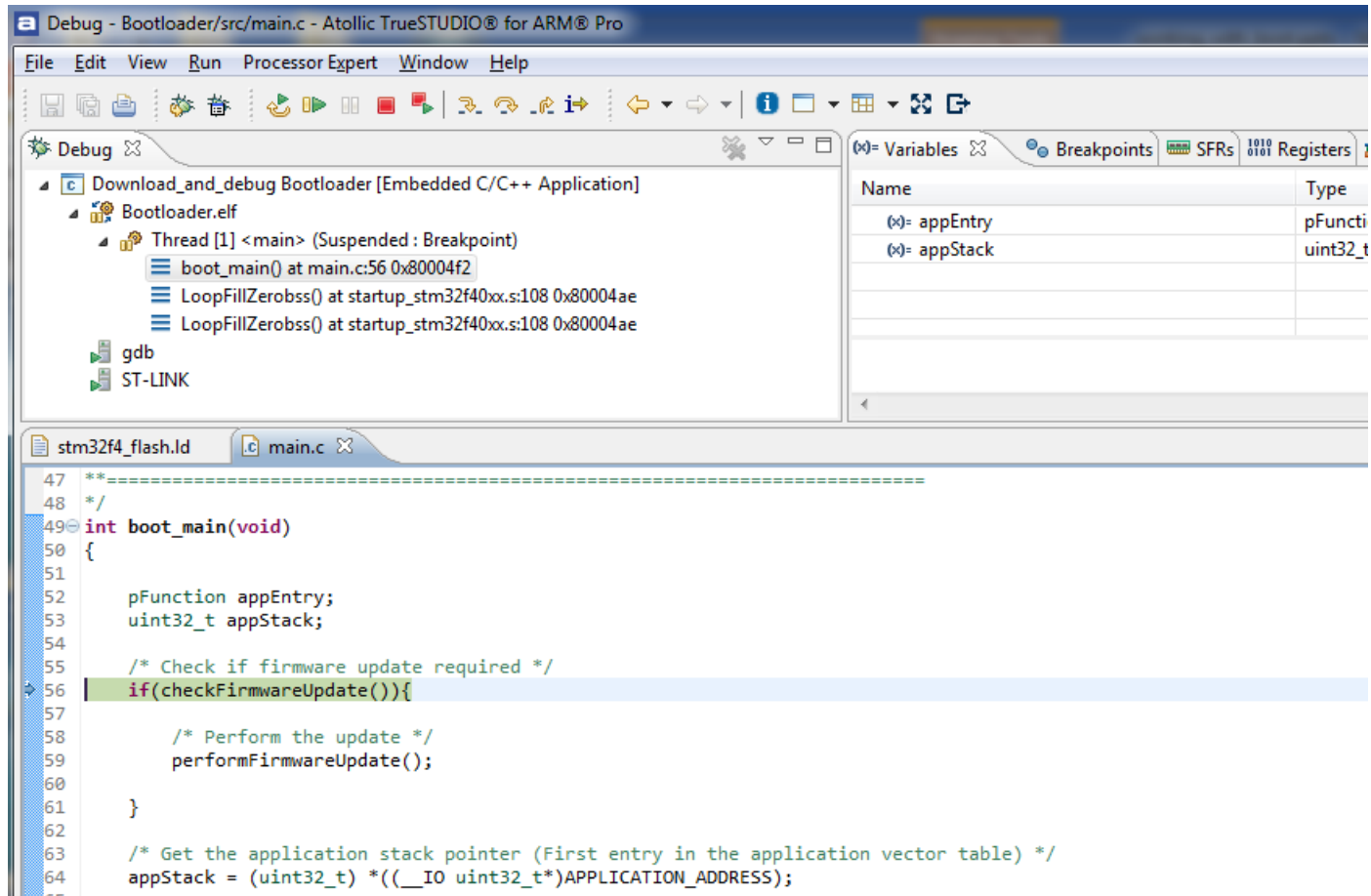
- Debugging the bootloader
- Debugging the application
- Debugging the bootloader & application
 - Application programmed by debugger
 - Application programmed by bootloader

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!



Debugging the bootloader



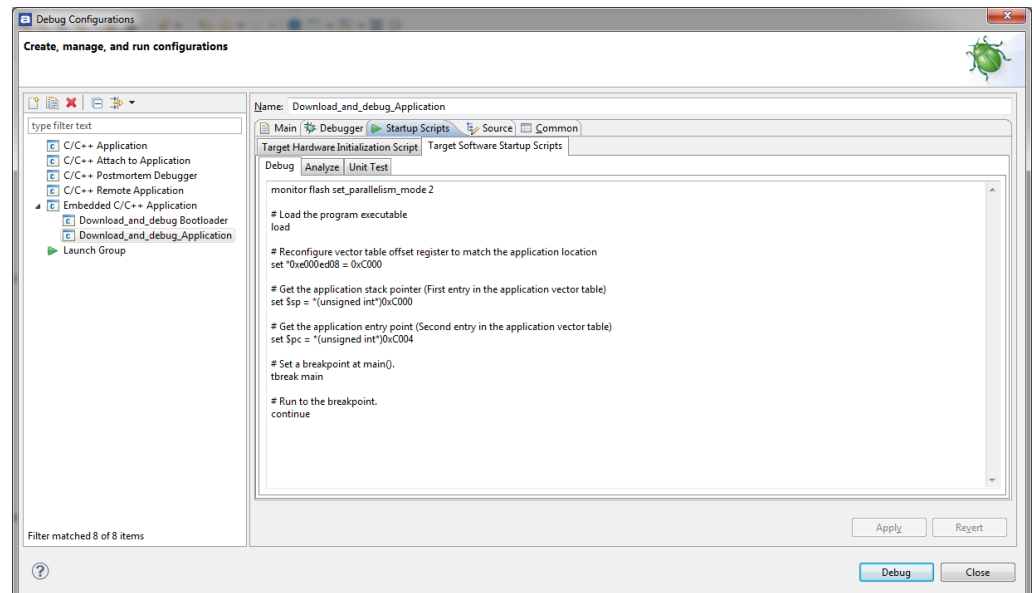
Debugging the bootloader

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

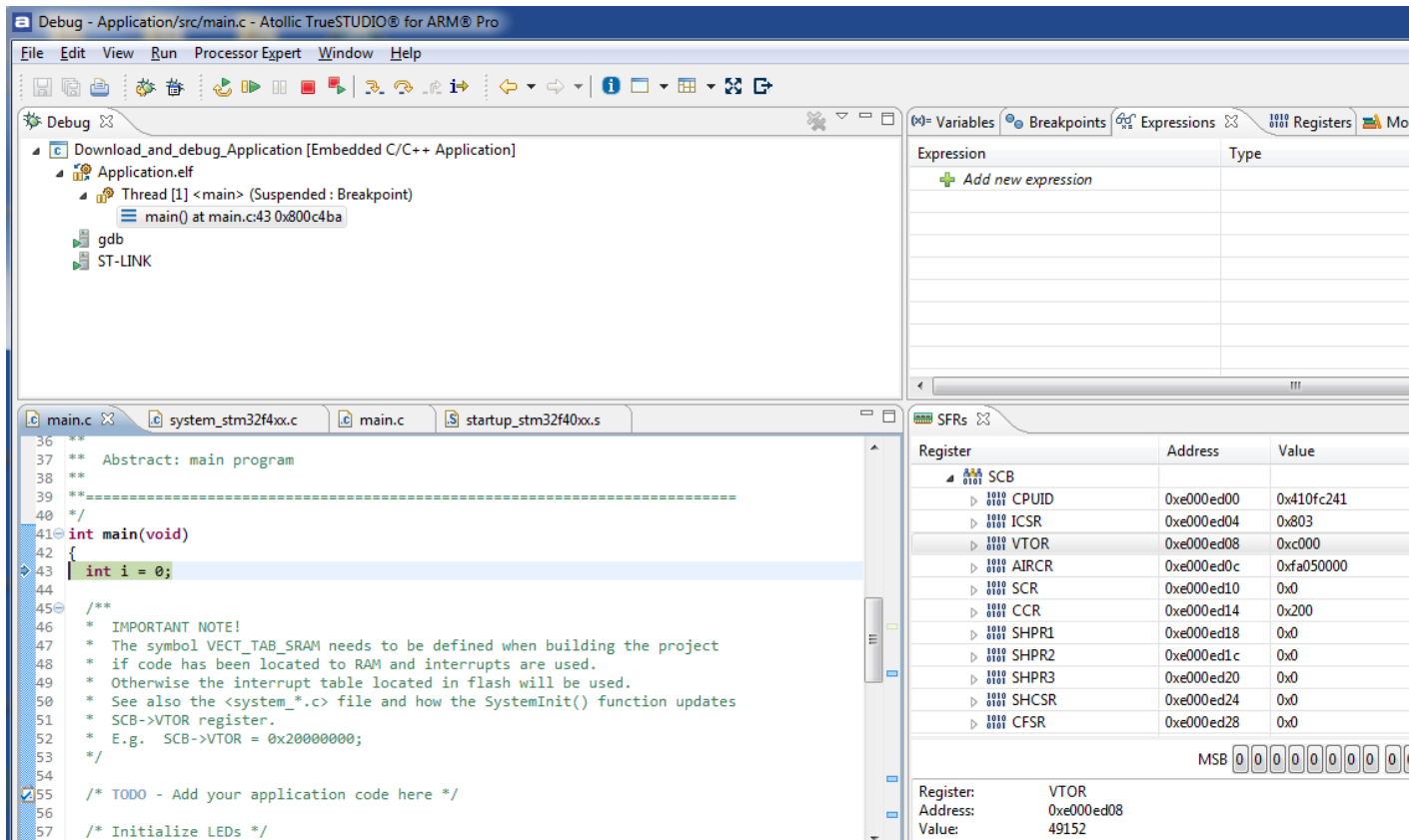
More on that later!

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



Debug - Application/src/main.c - Atollic TrueSTUDIO® for ARM® Pro

File Edit View Run Processor Expert Window Help

Debug

- Download_and_debug_Application [Embedded C/C++ Application]
 - Application.elf
 - Thread [1] <main> (Suspended : Breakpoint)
 - main() at main.c:43 0x800c4ba
 - gdb
 - ST-LINK

main.c system_stm32f4xx.c main.c startup_stm32f40xx.s

```

36 **
37 ** Abstract: main program
38 **
39 ** =====
40 **
41 int main(void)
42 {
43     int i = 0;
44
45     /**
46      * IMPORTANT NOTE!
47      * The symbol VECT_TAB_SRAM needs to be defined when building the project
48      * if code has been located to RAM and interrupts are used.
49      * Otherwise the interrupt table located in flash will be used.
50      * See also the <system_*.c> file and how the SystemInit() function updates
51      * SCB->VTOR register.
52      * E.g. SCB->VTOR = 0x20000000;
53      */
54
55     /* TODO - Add your application code here */
56
57     /* Initialize LEDs */
  
```

SFRs

Register	Address	Value
SCB		
CPUID	0xe00ed00	0x410fc241
ICSR	0xe00ed04	0x803
VTOR	0xe00ed08	0xc000
AIRCR	0xe00ed0c	0xfa050000
SCR	0xe00ed10	0x0
CCR	0xe00ed14	0x200
SHPR1	0xe00ed18	0x0
SHPR2	0xe00ed1c	0x0
SHPR3	0xe00ed20	0x0
SHCSR	0xe00ed24	0x0
CFSR	0xe00ed28	0x0

MSB 0 0 0 0 0 0 0 0 0 0

Register: VTOR
Address: 0xe00ed08
Value: 49152

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.

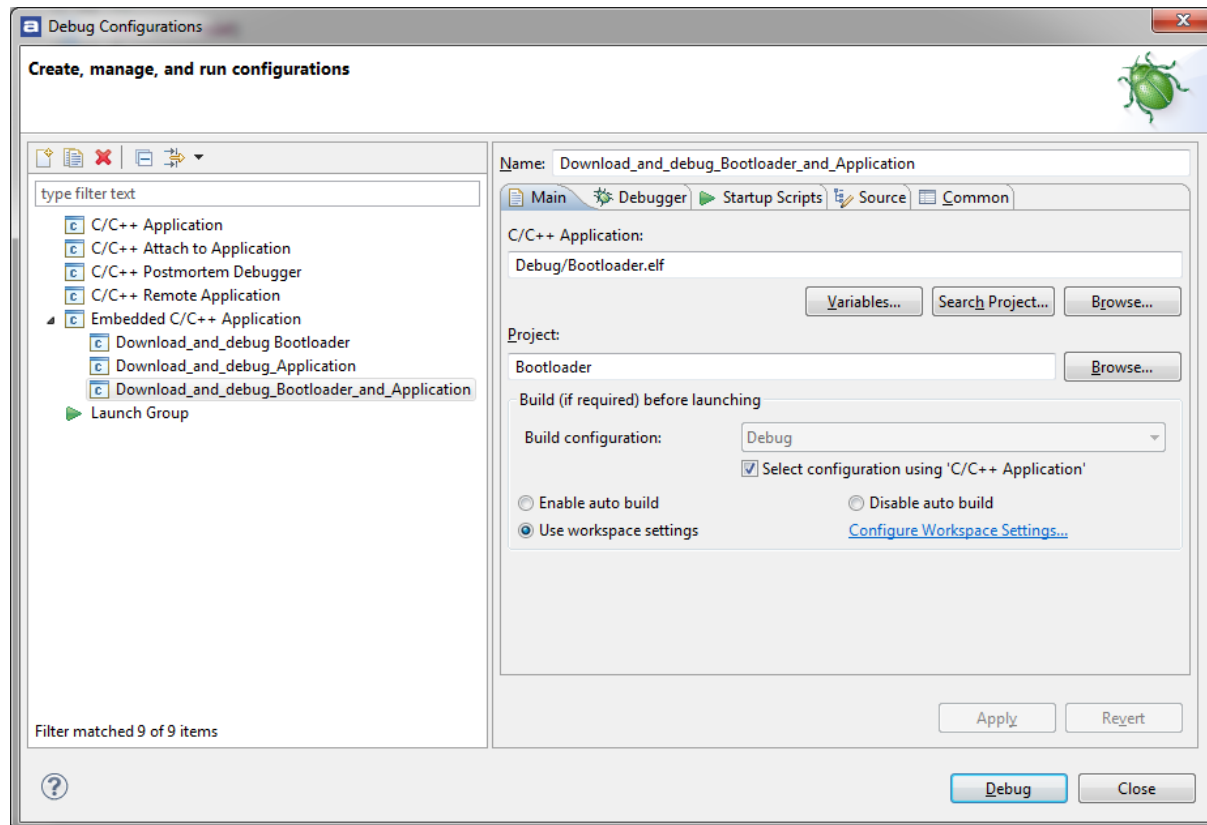
The debugger needs to have information regarding both of the ELF files!

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.



Debugging the application and bootloader – use case 1

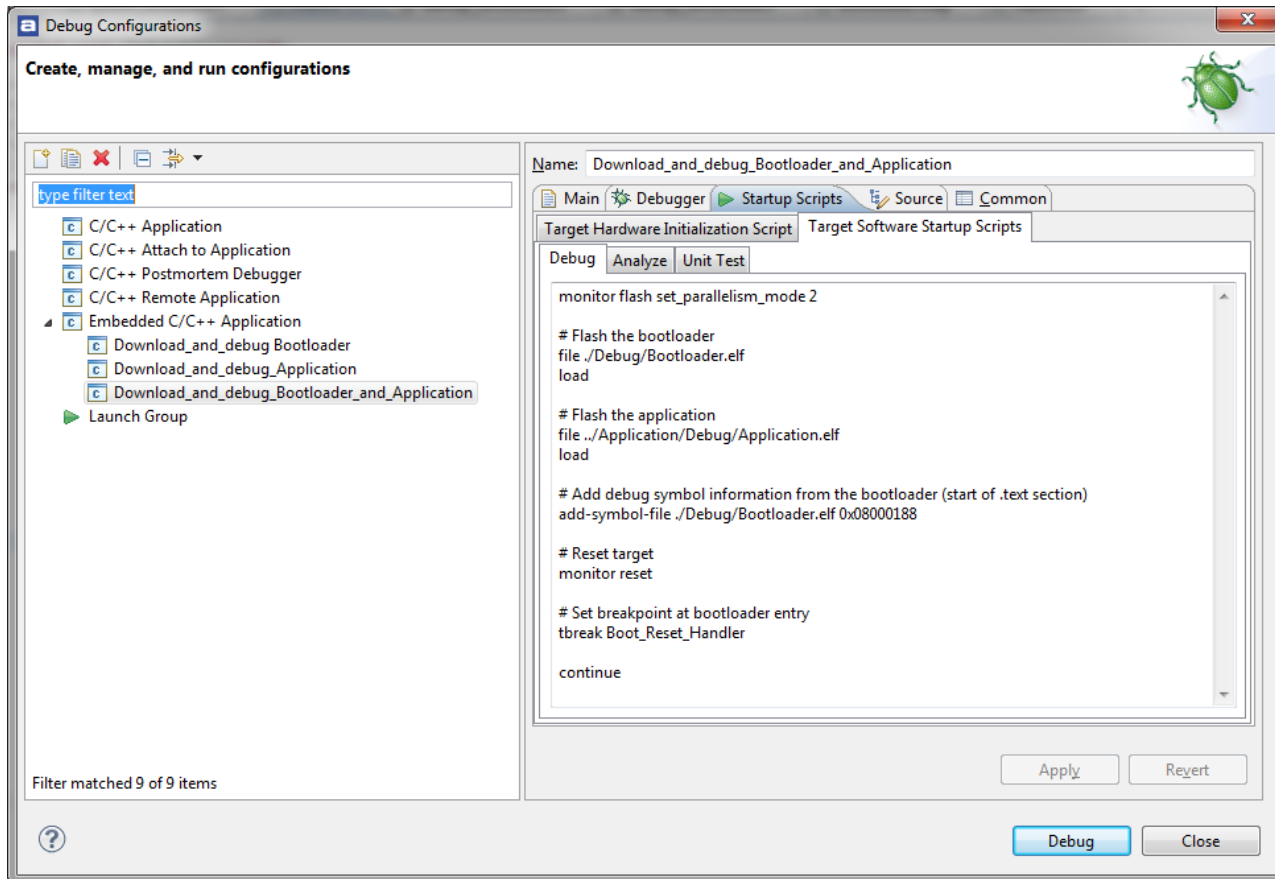
Edit the debugger startup script and instruct the debugger to perform the following:

- 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

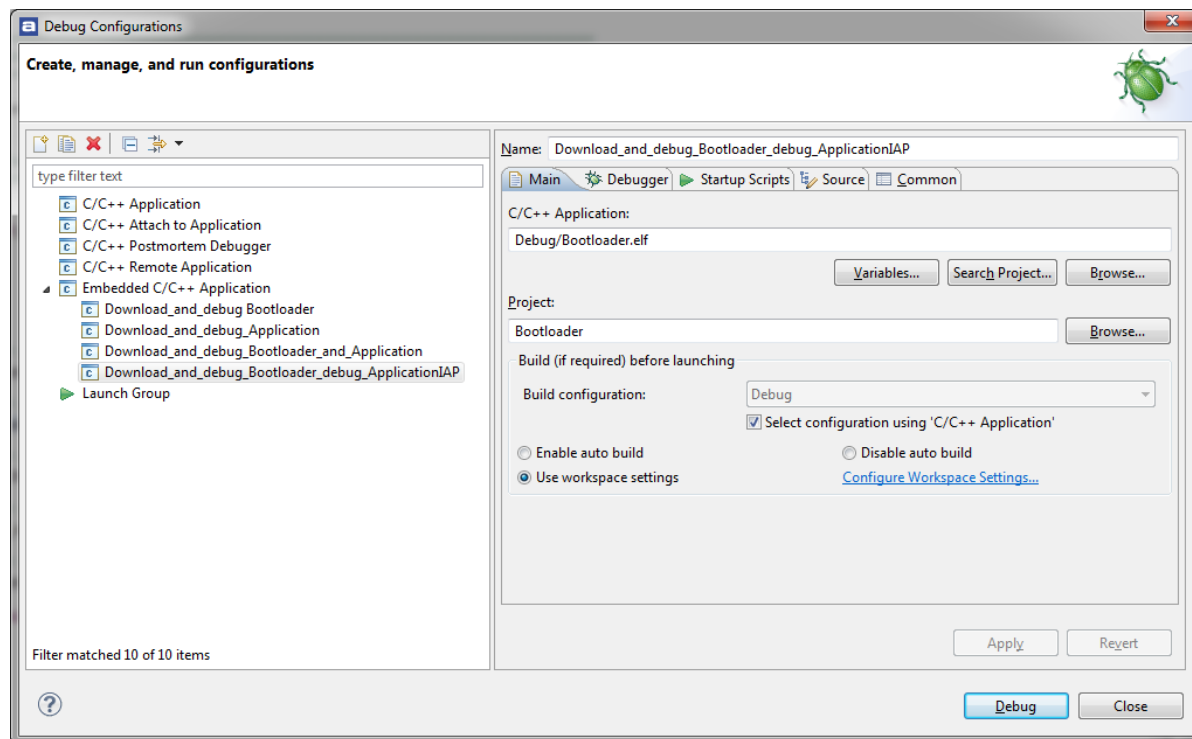


Debugging the application and bootloader - use case 2

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

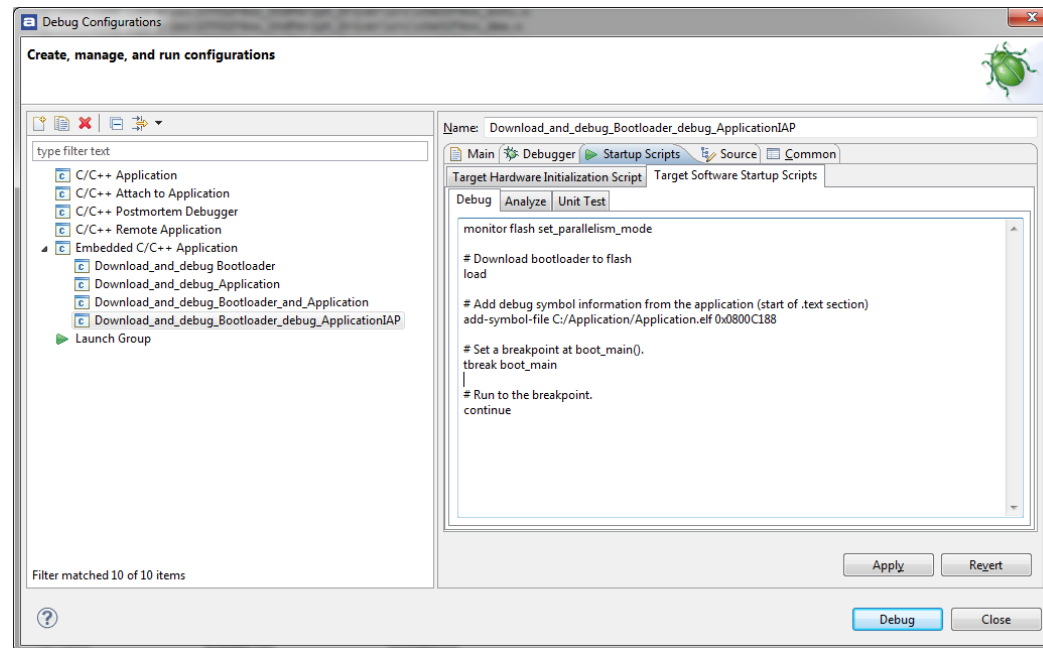
Create a fourth debug configuration connected to the Bootloader project. Use the "Download_and_debug Bootloader" configuration as template.



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

USA & AMERICAS

Atollic Inc.
241 Boston Post Road West
Marlborough, Massachusetts 01752
+1 (617) 674-2655

sales.usa@atollic.com

info@atollic.com