By Shawn Prestridge, IAR Systems Software Inc.
Co-author Colin Flournoy, IAR Systems Software Inc.

Constructing a bootloader with IAR Embedded Workbench and the STM32F207ZG-SK

- **Download application project (zip)**

As software becomes more and more complex, the potential for defects to exist in shipped code increases exponentially. It is therefore becoming quite common for design specifications to call for the ability to dynamically update a device's firmware in the field. This is done commonly via a bootloader. In this article, we will discuss some of the issues involved with developing a bootloader and show you how to set one up with the Embedded Workbench on the STM32F207ZG-SK board.

The pragmatics of designing a bootloader are somewhat difficult to discuss because there can be so many different requirements placed on it, e.g. the mechanism for getting the new application into the MCU: for some, it will be through the USB peripheral of the MCU, for others, it might be through the Ethernet peripheral and TCP/IP and still others might choose to do it through the venerable RS232 port. Because a bootloader is a very common thing to do, it is highly advisable that you check with your silicon provider's Field Applications Engineer to see if they have already created a sample bootloader project for your MCU which can greatly simplify your construction of a bootloader to fit your needs.

There also are subtle differences when creating a bootloader depending on the architecture you choose. For example the reset vector table may differ slightly whether you are using an ARM device versus a Cortex-M3. Also some chip manufacturers allow you to relocate the vector table while others may not. It is also not within the scope of this article to point out these subtleties, but to, for the sake of brevity, choose one as a demonstration and explain the setup.

Our test setup includes the IAR STM32F207ZG-SK development kit which contains a Cortex-M3 device. There are some prerequisites and assumptions taken on the part of the author. We are assuming that the reader has the correct hardware and is familiar with how to download a basic project to the internal flash of the device. The reader should also have a basic understanding of the macrofile system and how it can be used to manipulate the hardware to setup register values, etc.., prior to entering the debugger.

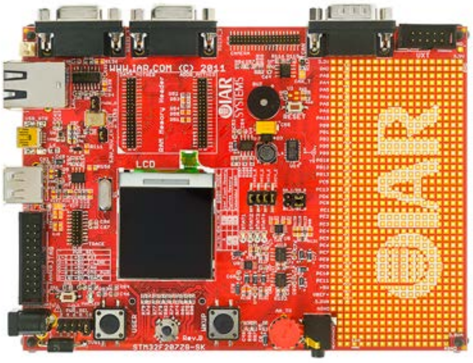
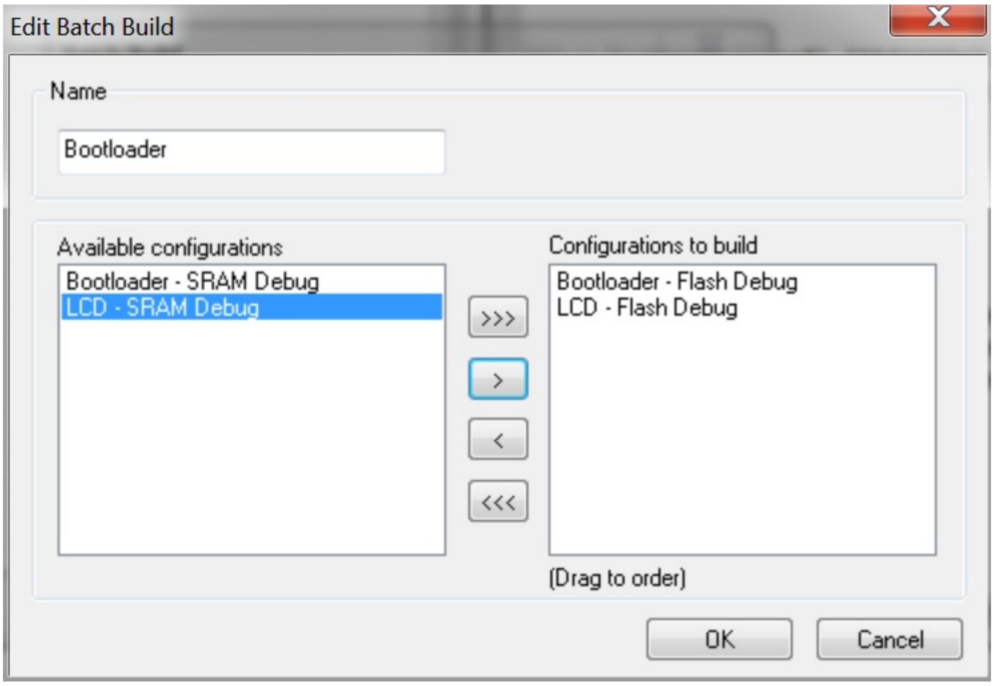


Figure 1: STM32F207ZG-SK development board

In the IAR hierarchy for project management, the workspace is the top-level entity and is what you open when start the Embedded Workbench. Workspaces can have multiple projects contained within them– for this article, I have attached a workspace with two projects: a main application project (LCD), based off our LCD example for this board as well as a bootloader project (Bootloader). Within the workspace, we can control in what order the projects are built. For our purposes, we want the bootloader to be built first and then the application; this is because we will place the bootloader image in the application when the application is built. This can be done using the “Batch build” facility available from the Embedded Workbench IDE by clicking Project-BatchBuild (see figure 2). You can edit the batch to add projects to the build as well as rearrange them in the order that you would like for them to be built. This is not essential, however, as you can also manually build each project if you prefer.



Name: Bootloader

Available configurations:
Bootloader - SRAM Debug
LCD - SRAM Debug

Configurations to build:
Bootloader - Flash Debug
LCD - Flash Debug

Figure 2: Batch build

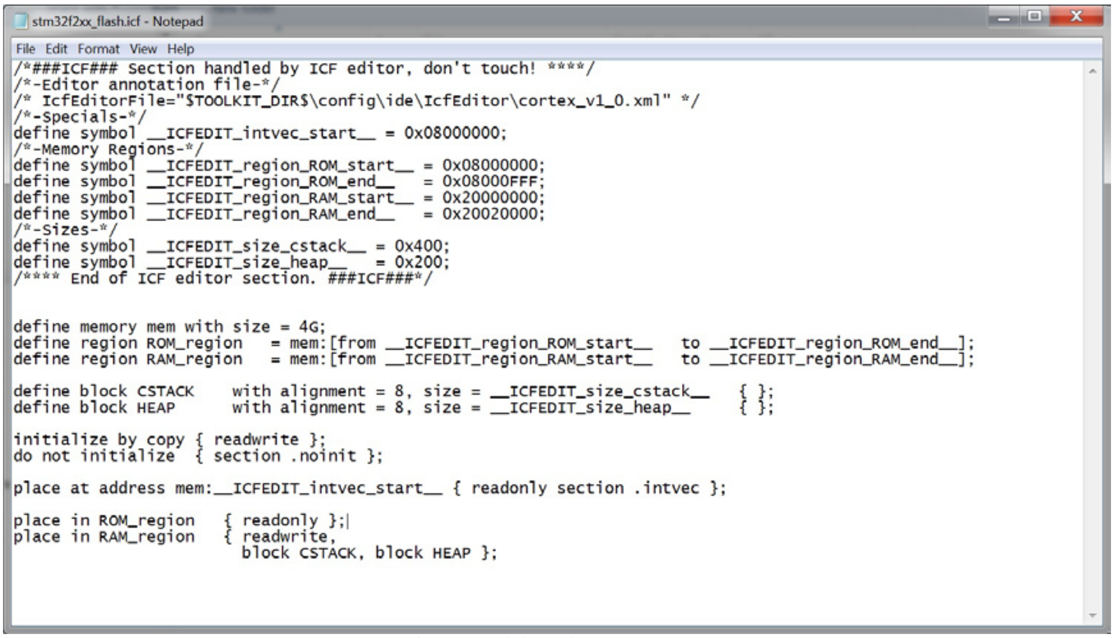
There are compelling reasons to NOT place your bootloader and application in the same project. First, the Embedded Workbench will generate a massive binary (especially if the bootloader and application start addresses are far apart). If there are peripheral addresses between the end of the bootcode and the start of the application code, you may experience flashloader failures when they try to write blank bytes to those addresses, thus necessitating a rewrite of the flashloader. Even if this isn’t the case, a bootloader will most likely have a statically allocated array for code reception that will waste space when the application is running. By having them as two separate applications, the linker can more efficiently allocate memory. Yet another compelling reason is that you cannot independently revise your bootloader or application—if they are in the same project, touching one automatically means touching the other and can therefore lengthen your test-and-fix cycle for your project.

Basic Setup

For this example, the idea is to setup the linker so that all of the code for the main application project is placed above a certain address (0x08010000 in this case) and all of the bootloader code is placed below that address (starting from 0x80000000). Once this is done, it is possible to produce a binary image of the bootloader project; the advantage of doing so is that you can import the binary image of the bootloader into the application project which will allow you to download the bootloader and main application all at once with the debugger. To summarize, we do the following:

Bootloader Project:

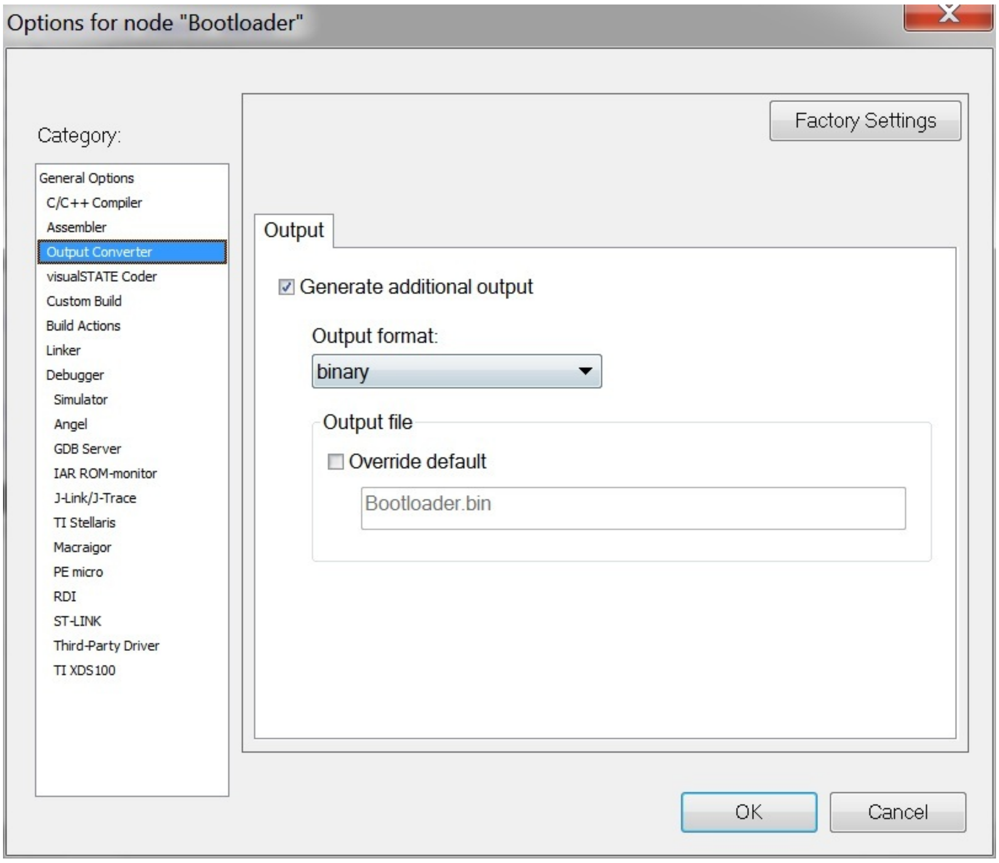
1. Create a project for the bootloader.
2. Setup the linker to place all of the code and constant data between 0x08000000 and the end of the space reserved for the bootloader (0x08000FFF in this case). See Figure. 3.



```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x08000FFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20020000;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0x200;
/**** End of ICF editor section. ###ICF###*/


define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
```

Figure 3: Linker placement setup

3. Set the output converter (Project-Options-OutputConverter-Output) in the bootloader project to create a binary file. The binary file is simply the content of the memory, i.e. there is no formatting or address information of any kind. See Figure 4.



☑ Generate additional output

Output format: binary

Output file

☐ Override default

Bootloader.bin

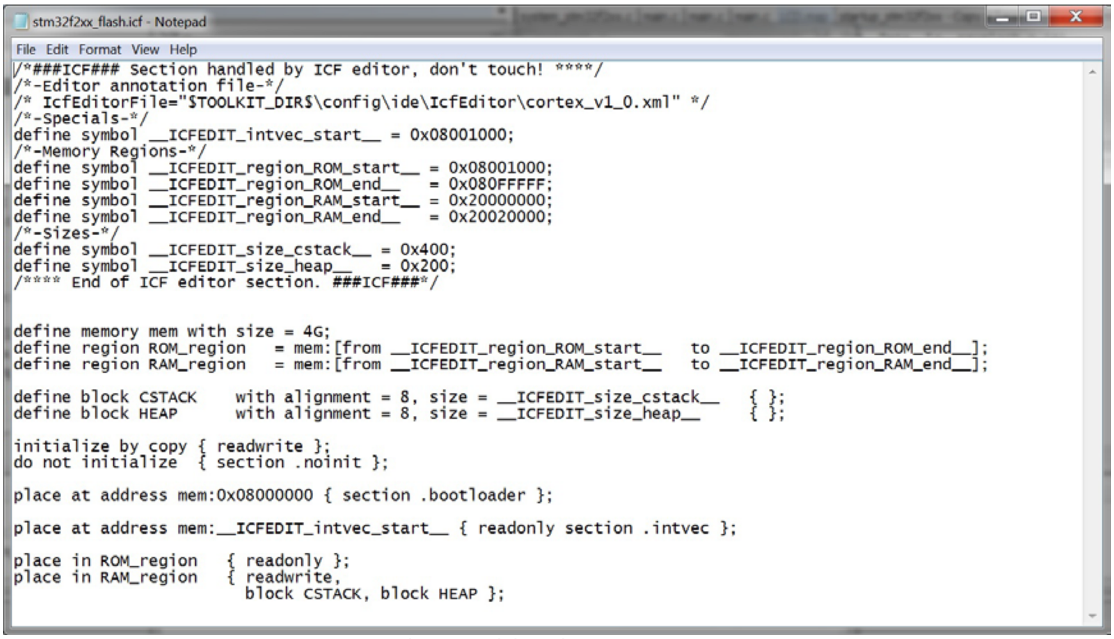
Figure 4: Output converter setup

Application Project:

1. Create a project for the main application (app). In this case we are using the default LCD example for the STM32F207ZG-SK board.
2. Set up the linker to place all the code (and constant data) between the start of the space reserved for the application and the end of memory. In this case starting from the end of the bootloader section 0x08001000 to the end of flash (0x080FFFFF in the case of the STM32F207ZG).Set up the linker file to place the section ".bootloader" at the desired location. To accomplish this, I have added the following line to my linker file (.icf).

   ```
   place at address mem:0x08000000 { section .bootloader };
   ```

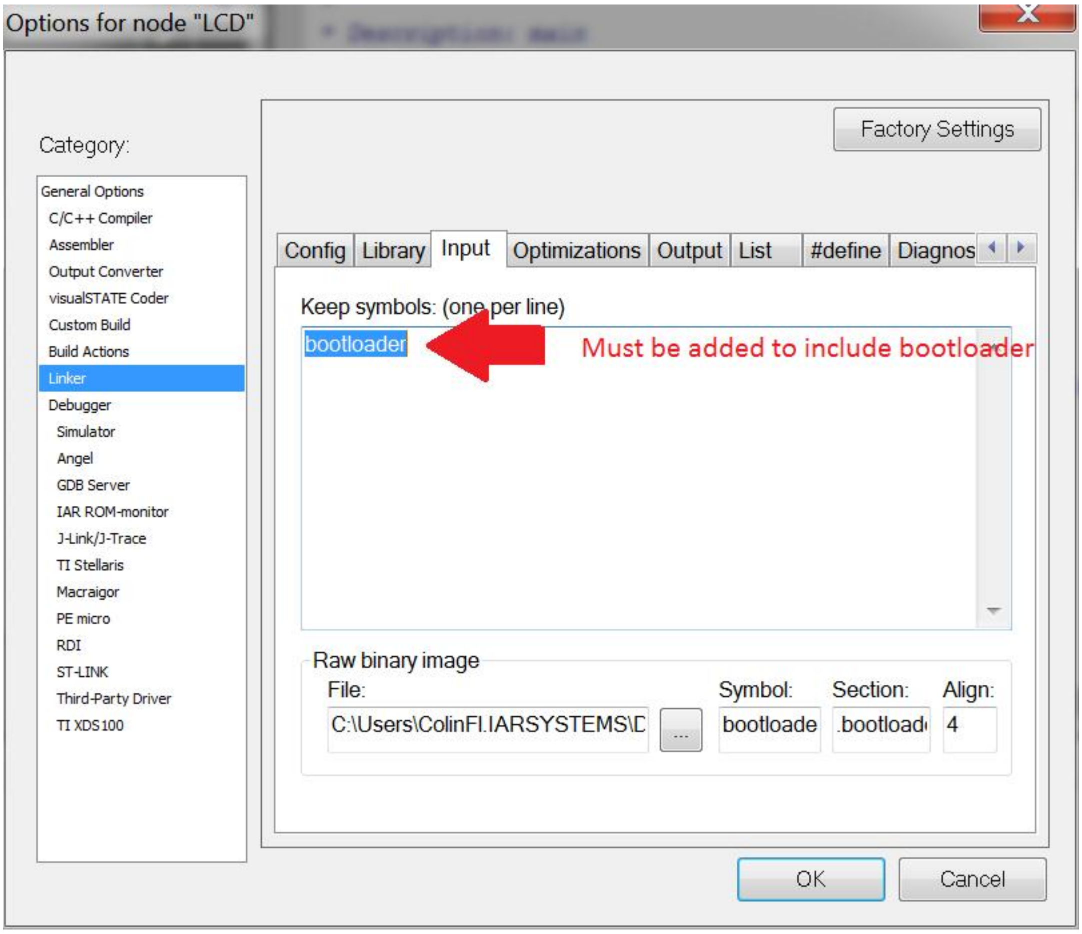
   This tells the linker to place the bootloader, or raw binary image, at the start of flash. See Figure 5.



```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08001000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08001000;
define symbol __ICFEDIT_region_ROM_end__   = 0x080FFFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20020000;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0x200;
/**** End of ICF editor section. ###ICF###*/


define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

place at address mem:0x08000000 { section .bootloader };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
```

Figure 5: Linker placement setup 2

3. Set up the linker to import the binary file created in part 1 of the bootloader project; this is done from Project-Options-Linker-Input. The linker will treat the binary image as though it were a constant array and simply copy its contents to the address specified in the linker configuration file. The linker must be given a symbol name for this image as well as in which section to place this image and the number of bytes on which to align the image. The linker must also be told to keep this symbol in case the symbol is not referred to in the application—any symbol not specifically referred to by an application will normally be discarded. In the attached example, I have chosen to make the symbol name "bootloader", the section name ".bootloader" and the alignment 4 bytes.



Keep symbols: (one per line)
bootloader

Must be added to include bootloader

Raw binary image
File: C:\Users\ColinFI.IARSYSTEMS\D
Symbol: bootloade
Section: .bootload
Align: 4

Figure 6: Linker input

Debugging

In general the process will be the same for all intents and purposes when it comes to debugging your application. The slight modifications come due to the architecture/device you may be using. Let's summarize once more before we begin setting up the debugger.

1. We created a bootloader project which builds an additional raw binary file as input to the application
2. The actual application was then created and the raw binary file(bootloader) was added via the project options
3. The linker files were both modified to not have the code/data sections overlap one another

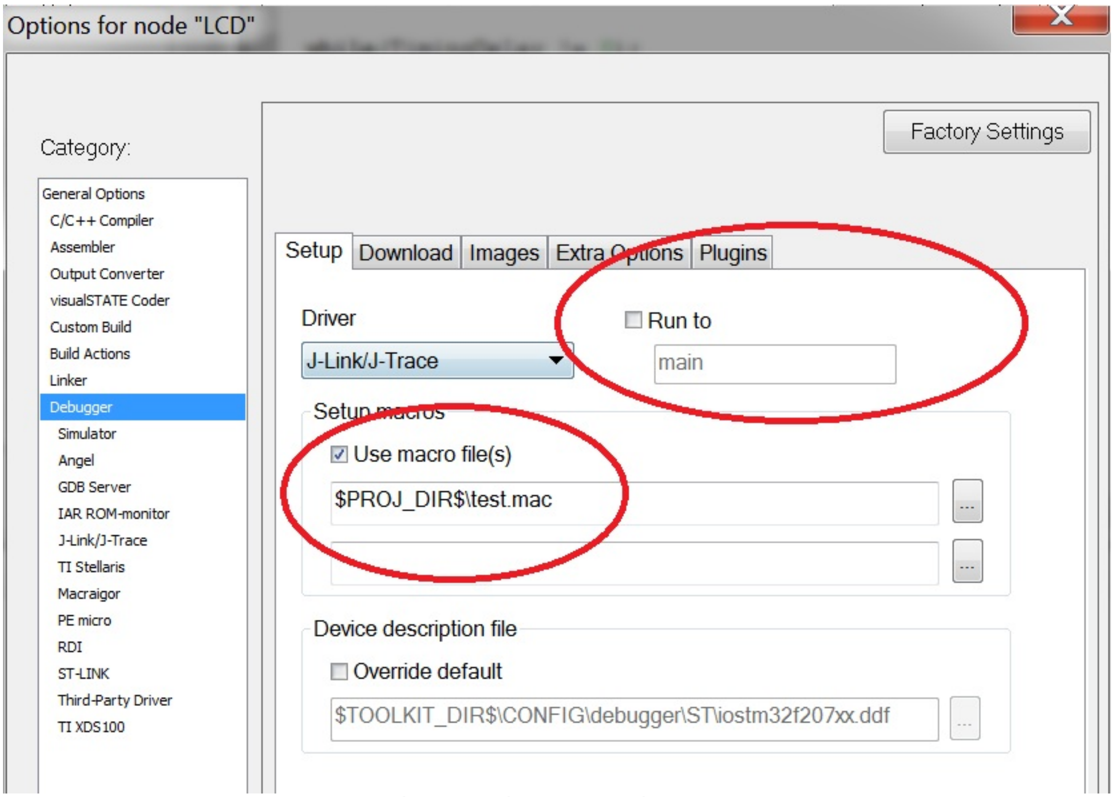
Now we need to somehow let the debugger know that we aren't just debugging the application code, but both the bootloader and the application code simultaneously. Not only that, we need to take into account that there is a vector table for both your bootloader code and your application. The STM32F2xx devices contain a register which contains the base address of the vector table. This will need to be adjusted for the fact that the vector table location changes when our bootloader application jumps to the main application.

So first we need to "tell" the debugger to start at the reset vector associated with the bootloader and not the application code, initially. This is done via, what we call a macro file. If you go to your application project options-debugger-setup tab, you will see an option to add a macro file (.mac). Here is where you are going to tell the debugger to run your bootloader code first as we want to debug the bootloader and then move on to the application.

In the example you will see a macro file called test.mac which is already added to the application options. You can name the macrofile whatever you want, as long as it has a .mac extension. In this file we need to tell the debugger to start at the Reset_Handler for the vector table associated with the bootloader. We also have to initialize the stack pointer to that of the bootloader and not the application. The below macro code shows how this is done.

```
//in test.mac
execUserReset()
{
   MSP = *(int*)0x08000000;       //set the stack pointer to the beginning of
                                  the bootloader vector table
   PC = *(int*)0x08000004;        //set the program counter to the beginning
                                  of the bootloader reset handler
}
```

Once you have finished editing the macro file in a text editor, you need to add it and enable the file under your project options-debugger-setup tab. The first step in testing your program is to make sure that the bootloader is executing first within the debugger. For this purpose we uncheck the “run to main” option in the below figure.



Driver: J-Link/J-Trace

☐ Run to: main

Setup macros

☑ Use macro file(s)

$PROJ_DIR$\test.mac

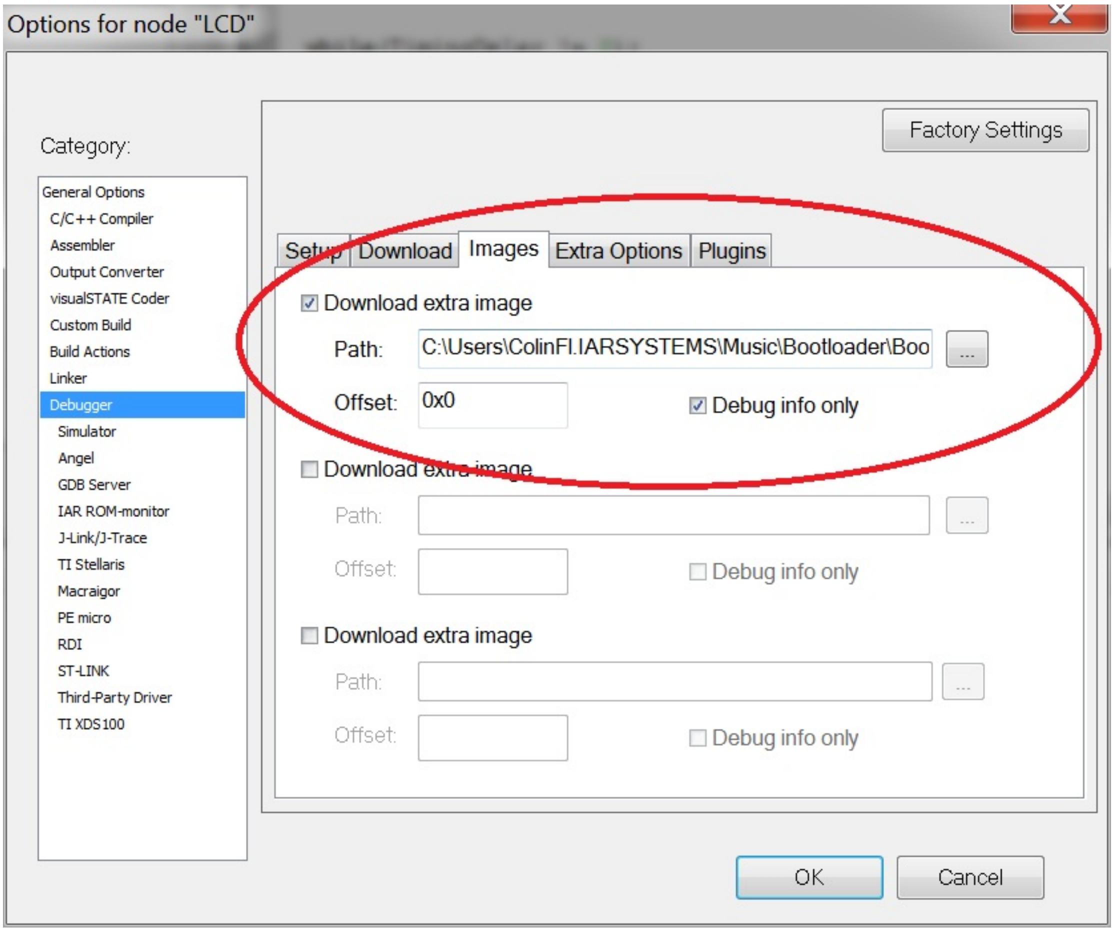
Device description file

☐ Override default

$TOOLKIT_DIR$\CONFIG\debugger\ST\iostm32f207xx.ddf

Figure 7: Debugger execution setup

We have already setup the app to include the raw binary image, but it is just that, a raw binary image with no debug info. We need to add the debug info so that you can also debug the bootloader at the source file level. We go to your project options-debugger-images tab. Here we want to enable “download extra image”. You need to point the path to the .out file of the bootloader project. The .out file not only contains the executable code for the bootloader, but also the debug info. This file is usually in the "exe" folder of your bootloader project. You only want to add the debug info, so we check "debug info only” and set and offset of image to 0x0.



☑ Download extra image

Path: C:\Users\ColinFI.IARSYSTEMS\Music\Bootloader\Boo

Offset: 0x0

☑ Debug info only

☐ Download extra image

Path:

Offset:

☐ Debug info only

☐ Download extra image

Path:

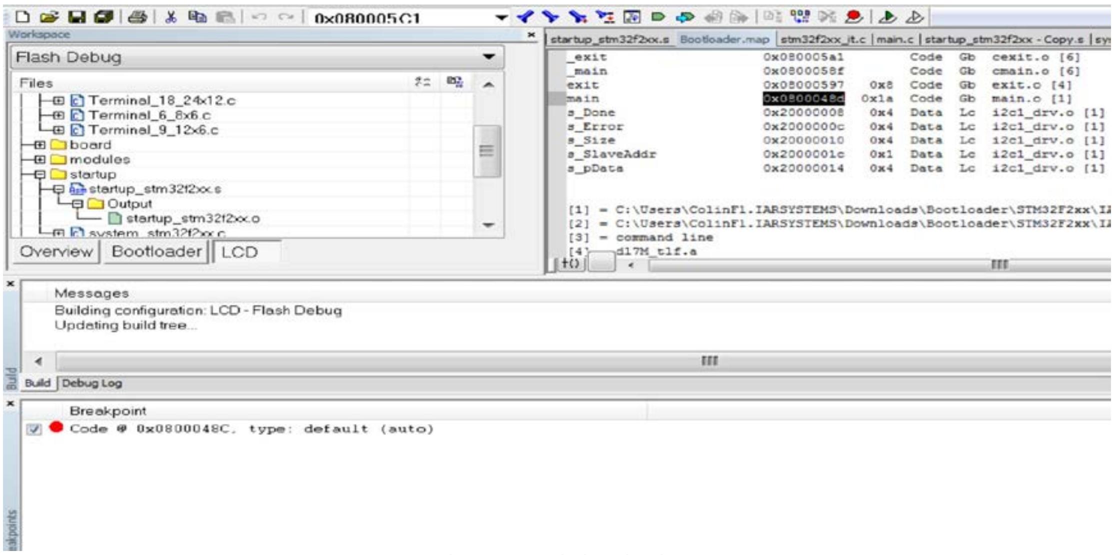
Offset:

☐ Debug info only

Figure 8 : Debugger images setup

Now if you have done everything correctly, then when you hit "download and debug", in your application, and you should see the "flash programming" dialog box appear twice. This is normal operation. It indicates that not only is your application being programmed, but also your bootloader program has been burned. The program counter will now reside on the bootloader reset handler/vector table, because the macrofile set your program counter to this location. Now you want to test the program out for flow control, in other words, you want to make sure that as you single step through the program you can go from your bootloader to your application with no issues.

Now remember you have downloaded the two projects via the application project. That means that you have access directly to the source files of the application and not the bootloader. While in the debugger, if you try and click on a source file associated with the bootloader app, you will be prompted to switch configurations, or projects. This may not be desirable, because you want to continue debugging both applications. There is a way around this, however. You can either single-step at the disassembly level until you reach your bootloader code (main function), or you can just set a breakpoint via providing a code instruction address to break at. Open the view-breakpoints window and right click within the window. Select “new breakpoint” and add a new “code breakpoint”.



```
_exit                  0x080005a1          Code  Gb  cexit.o [6]
_main                  0x0800058f          Code  Gb  cmain.o [6]
exit                   0x08000597   0x8    Code  Gb  exit.o [4]
main                   0x0800048d   0x1a   Code  Gb  main.o [1]
s_Done                 0x20000008   0x4    Data  Lc  i2c1_drv.o [1]
s_Error                0x2000000c   0x4    Data  Lc  i2c1_drv.o [1]
s_Size                 0x20000010   0x4    Data  Lc  i2c1_drv.o [1]
s_SlaveAddr            0x2000001c   0x1    Data  Lc  i2c1_drv.o [1]
s_pData                0x20000014   0x4    Data  Lc  i2c1_drv.o [1]

[1] = C:\Users\ColinF1.IARSYSTEMS\Downloads\Bootloader\STM32F2xx\I
[2] = C:\Users\ColinF1.IARSYSTEMS\Downloads\Bootloader\STM32F2xx\I
[3] = command line
[4] = dl7M_tlf.a
```

```
Messages
Building configuration: LCD - Flash Debug
Updating build tree...
```

```
Breakpoint
☑ ● Code @ 0x0800048C, type: default (auto)
```

Figure 9: Code breakpoint

The beginning of your “main” in your bootloader is displayed in the map file of the bootloader project. You can use this to pick an appropriate address to break at. If all goes well and you hit “run”, you will break somewhere in you bootloader’s main function.

As discussed previously, the mechanism of how a bootloader can be implemented can vary widely. For the purposes of demonstration we have created one of the simplest possible bootloaders. One in which it always jumps to the application regardless of the input. This is done via a function pointer, which points to the address of the reset handler of the actual application. The code is included below for your reference.

```
//in bootloader main.c
int main(void)
{
  //jump to application via function pointer
  int * address = (int *)0x08001004;    //pointer to reset handler of
                                        application
  __set_MSP(*(int*)0x08001000);         //set the stack pointer to that of
                                        the application
  *(int*)0xE000ED08 = 0x08001000;       // VTOR: set the vector table to
                                        that of the app
  ((void (*)())(*address))();           // pointer recast as function
  pointer and the dereferenced/called
  while(1){ };                          //you application was never executed
}
```

This is where it starts to get specific to the STM32F2xx devices. We need to "move" the vector table location before the bootloader jumps to our application code. The bootloader and device currently "believe" our bootloader vector table is located at 0x08000000 and will reside there indefinitely. The bootloader code resets the vector table to address 0x08001000, where the application was originally linked. This will allow proper execution of the application code and setup the vector table appropriately.

So you are halfway there; we need to verify that we can reach and run the main application. If you set a breakpoint within the main function of your application, you should be able to break there and keep running the LCD application, eventually seeing output on the LCD screen. Hit "run" and watch the bootloader pass control to your application which should now execute.

To summarize this is the most basic kind of bootloader possible. It always jumps to your main application. This project can be expanded into a much more elaborate bootloader with conditionals and other user input. However, it demonstrates 3 important aspects.

1. It demonstrates how to create the skeleton of a bootloader, even if it is a simple example
2. It shows how to integrate both your bootloader and your application code to download and debug all at the same time
3. It is an example working on real hardware and can be used as a basis for your own setup

Bootloaders can be an invaluable tool when field upgrades are a necessity for your application. Hopefully with the previous steps, you will be well on your way to constructing a bootloader for your own setup. A link is provided to the example used in this application note below.

- **Download application project (zip)**