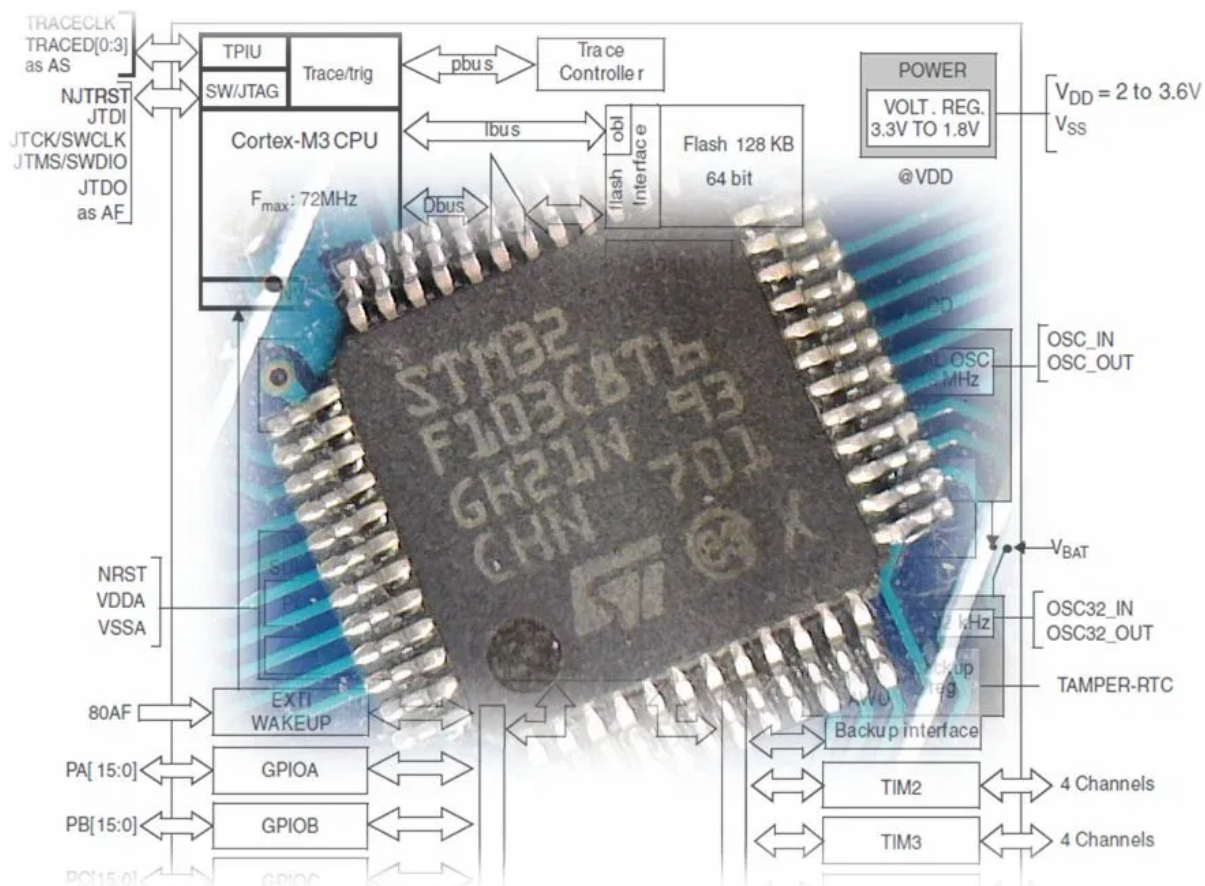


karooza.net (<https://karooza.net/>)

Covering electronics and coding for the hobbyist that wants to build everything



Proudly powered by WordPress (<http://wordpress.org/>) | Theme: FlyMag (<http://themeisle.com/themes/flymag/>) by Themeisle.



Going Bare Metal on STM32

During the last couple of months I have started using my Blue Pill boards more often for even the smallest of projects and tests. When using them with STM32duino in the Arduino IDE they are just as easy to program as a normal Arduino UNO. In a previous post I gave more of an introduction if you are interested. But more recently I wanted to experiment with programming the STM32F103 without using any IDE but only a text editor and the GNU compiler for ARM processors. The main driver for this was to eventually start using other frameworks and in particular libopenm3. Going this route, the idea was also to learn a bit more about setting up the tool chain and not being bound to some specific IDE. This forced me to take a step back and take a closer look at the inner workings of how a C/C++ program is compiled, uploaded and executed on the microcontroller. After a decent amount of time spent reading up on the topic and writing test code I now have a more in depth understanding of the process. In this post I would like to share my newly gained knowledge. I will try to keep it as simple as possible but some basic understanding of programming and microcontrollers would be helpful to follow along. If you want to try out the examples you would also need a development board (would recommend the blue pill) and a STLink programmer.

I used the STM32F103 MCU and therefore most of the information in this discussion is based on it. But since it is an ARM Cortex M3 device, a lot of the details discussed, is also relevant to many other microcontrollers based on this architecture.

We will start the discussion with a bit of theory to get familiar with a few basic concepts, followed by some examples. For the theory we will look briefly at the following topics:

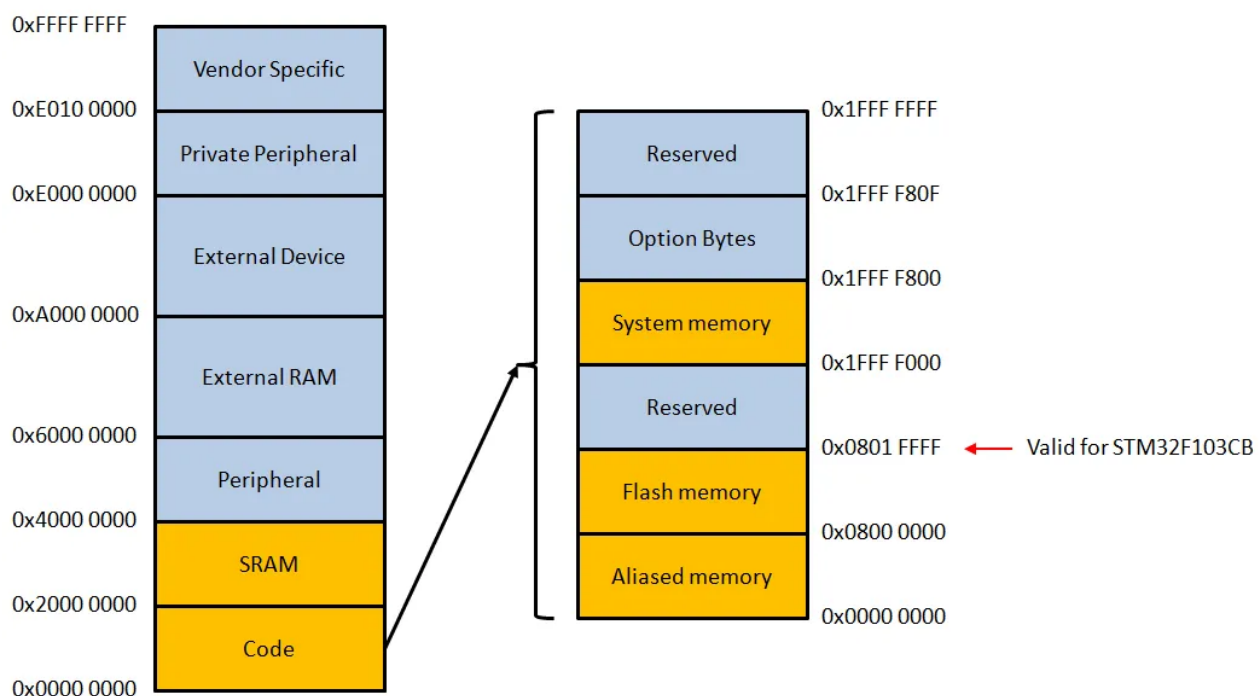
- The memory map
- Boot sequence
- Vector table
- Organisation of memory
- Memory alignment
- Instruction set
- Building the program

From there we will start to configure the tools we need and create a simple example.

Memory Map

During our discussion we regularly refer to the memory areas of the Cortex M3 so it's a good idea to start by having a quick overview of how it is laid out. Cortex M3 devices, being 32bit, has a linear addressable memory space of 4GB. The diagram below shows how the main areas are mapped out:

ARM Cortex M3 Memory Map



By reading or writing to these addresses we have access to the different memories and peripherals of the MCU. For this discussion however the sections of interest are the two bottom blocks, the Code and SRAM sections. The Code section is where most of the flash memory we have access to is located and as its name suggests the SRAM section is where the RAM is located.

If we zoom in on the Code section we can see that it's divided into even smaller sections. Here the sections of interest to us are the System, Flash and Aliased memory.

The System memory is where things like the factory bootloader are stored and to which we typically do not have write access.

The Flash memory section is where our program instructions and some constant data are typically stored and to this memory we mostly have complete write and read access. The range of this memory depends on the available flash memory of the device being used. In the case of the STM32F103CB this memory can extend up to address 0x0801 FFFF which provides $0x0801\text{ FFFF} - 0x0800\text{ 0000} = 0x001\text{F FFFF} = 131.071\text{ kB}$ of memory (the datasheet guarantees at least 128kB).

The last section, the Aliased memory can be mapped to the Flash, System memory or the SRAM depending on the boot mode selected with the boot pins. Typically we can not read or write this memory directly, how this works and why it's done will become more clear when we speak about the boot sequence.

Boot Sequence

When power is applied to the MCU the Program Counter (PC) value will be 0 and will therefore start at address 0x0. Remember that the Program Counter holds the address of the next instruction to be performed. The value it expects at this address (0x0) would be the address that marks the top of the Stack memory (which is part of the SRAM). This address is then copied to the Stack Pointer (SP) register for later use. The Program Counter then steps to the next address which is 0x0000 0004 and expects the address of the reset handler at this location. This is typically the start address of where our program is stored. The MCU will then update the Program Counter with this address which means that the next instruction to be executed will be the first instruction in our program.

In our discussion on the memory we said that our program normally lives in the Flash memory which starts at 0x0800 0000 and we also said that we can't write to the Aliased memory (which starts at 0x0). So how do we set the values of address 0x0000 0004 (which is in Aliased memory space) to point to the start of our program? Well, this is why the Aliased memory can be mapped, by selecting (with BOOT0 and BOOT1 pins) to boot from the Flash memory the Flash memory is actually mapped to the Aliased memory. This means that when the Program Counter looks at address

0x0 it's actually looking at address 0x0800 0000. So since we can write to the Flash memory section we can provide the starting address of our program at address 0x0800 0004 which would be the same as 0x0000 0004. Typically the part of our program located at this address is called the startup code and will perform some initialization steps and then call the `main()` function in our C/C++ code.

Vector Table

The vector table is a section of our flash memory that mostly holds the addresses of various handlers. In the boot sequence we said that the reset handler is located at address 0x0000 0004 and holds the address of our startup code. Well this is part of the vector table and for the STM32F103 it starts at address 0x0 and extends up to address 0x0000 014C. As a reminder, this is aliased memory, so if we are booting from Flash memory it would actually mean that the vector table extends from 0x0800 0000 to 0x0800 014C. If you have used interrupts before then this is also where the handlers (address to the code that will be run) is set. For the moment we only care about the reset handler but the datasheet gives some details about the rest if you are interested.

Organising the Memory

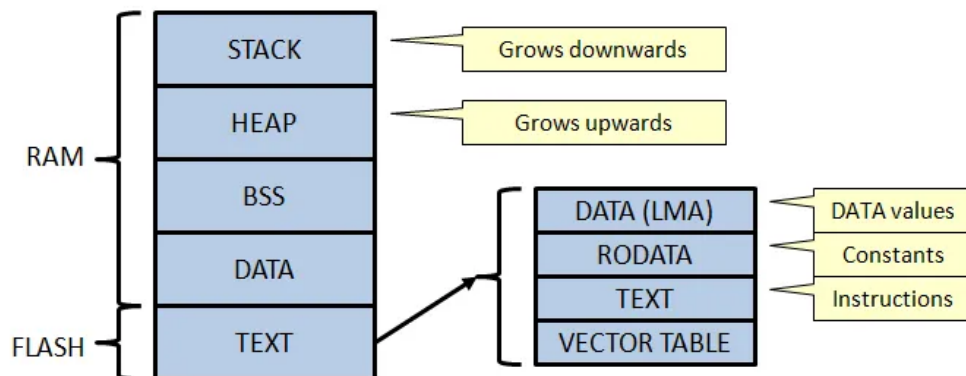
Earlier we talked about how the memory in the MCU is mapped out to the different regions of the FLASH and SRAM memory. When our program is compiled the compiler also organises the program into different sections which we would need to fit to these regions. The most popular of these sections are the TEXT, DATA and BSS sections. This is done to separate for example, things like the instructions which is typically in read only memory from variables which should be in read/write memory. The read only memory is typically FLASH memory but does not have to be, it's also possible to load instructions to the SRAM and execute them from there, but in this post we will use the FLASH memory as our read only memory.

The region that holds the instructions of our program goes into the TEXT section. The TEXT section is located in the FLASH memory and is typically only written to when we upload our program to the MCU. If our program contains constants the compiler knows that these values will always stay the same and there is no need for them to be in RAM. Therefore they can also be placed in FLASH memory and forms part of the RODATA (Read Only Data) section.

If our program had global or static variables (which lives for the whole lifetime of our program) then the compiler can already assign space (addresses) in RAM for it. The addresses it assigns to these variables will be in the BSS section which is in RAM memory.

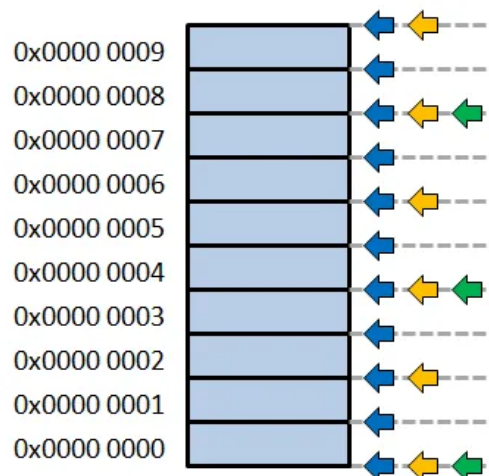
Should we have global or static variables which have also been initialized (given a

value) in our program then the compiler will assign the address in the DATA section and the value itself in the LMA (Load Memory Address) DATA section. When the MCU boots-up these values should then be copied from the LMA DATA to VMA (Virtual Memory Address) DATA section.



Memory Alignment

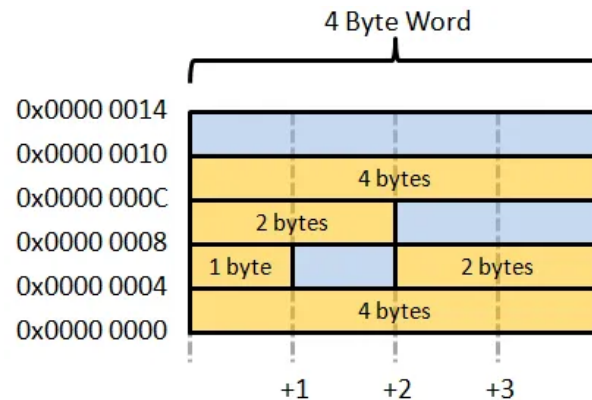
For the processor to efficiently fetch data the address from where it fetches data should be a multiple of the data type size. For example, if we retrieve a 32bit (word) value then the address must be a multiple of 4. So valid addresses would be 0x0800 0000, 0x0800 0004, etc. When using a 16 bit (half word) value the address must be a multiple of 2 and for 8bit addresses any address can be used. This is known as aligned memory access.



Looking at the image above, imagine each block represents one byte of memory and the address of the byte is to the left of it. Green arrows would indicate valid addresses for word (32 bit) values. Yellow arrows would be valid for half word (16 bit) values and blue arrows for byte values.

If we try to read data from unaligned memory the processor would need to fetch the data twice and we would get a performance penalty. One solution to this problem is to

use alignment instructions when data is assigned an address. Such an instruction would typically round the address up to the closest multiple of the data size. This works but at the cost of creating unused memory locations. In the image below we can see the effect of memory alignment.



Take as example address 0x0000 0008 which holds a 2 byte value, the next value to be stored is a 4 byte value. But since we need to align the memory we can't store it at address 0x0000 000A, the first available address would be 0x0000 000C which creates 2 bytes of unused space.

Normally the alignment is handled by the compiler but knowing about it helps to understand some other concepts. We can also use this to improve the memory use of our programs by packing our data in such a way that we don't fragment the memory too much.

Thumb Instructions

As we know by now the processor in the STM32F103 belongs to the Cortex M3 series which uses the ARMv7-M architecture. This means that these processors do not use the full ARM instruction set but rather the optimized Thumb instruction set.

Interesting play on words there, from Arm to smaller Thumb... Anyway the Thumb instruction set supports both 16bit and 32bit instructions, the advantage of 16bit instructions are that they use less program memory. While in many cases using 32bit instructions can be faster, so by being able to use both we get the best of both worlds. Some processors supports both ARM and Thumb instruction sets and the set being used can be selected during branch (jumping to another address) instructions. If the LSB (least significant bit) of the address to which the program branches is '0' then the ARM instruction set is used. If the LSB is '1' then the Thumb instruction set is used. Since the instructions are always aligned to 16bit or 32bit memory boundaries, changing the LSB has does not affect the destination address.

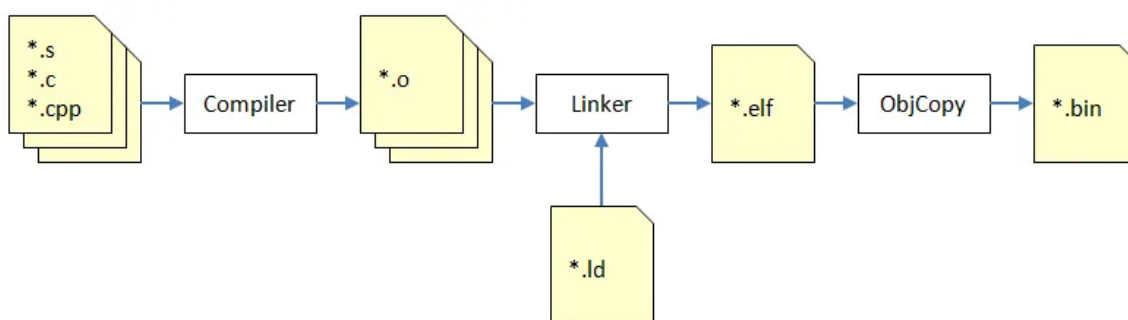
Knowing this is important because even though our Cortex M3 processor only supports Thumb instructions we still need to follow this convention to prevent the

processor from generating an error.

Building our program code

If you are familiar with how to build your program from source code to binary then you can skip this section, this is only to refresh our memory on the process. Creating a binary file from our program code (C/C++ or assembly) is a multi-step process. As a first step the compiler runs a pre-processor through the code, the job of the pre-processor is to replace all the macros (like `#include<...>`, `#define ...`, etc.) with the actual code. Then the source code files gets compiled into object files. Technically C/C++ code files are first compiled to assembly and from assembly to machine code. If our program also contain assembly files then they are obviously compiled directly to machine code. The compiler creates a machine code object file for each input source file. These object files contains the instructions and data for the various functions of our program. Typically these object files also reference functions and data from other object files and therefore they need to be linked together, this is the task of the linker. In addition to object files, the linker also takes as input a linker script file (if none is provided it uses a default script) which is used to set some rules for the linker. Later in this post we will also take a closer look at linker scripts.

The output from the linker is typically an executable file and in our case will be a `.elf` (Executable and Linkable Format). Since this is an executable file it contains more than just the instructions and data of our program. When flashing our program to the MCU we want only the instructions and data used by our program. To do this we use a special tool in the GNU toolset called "objcopy" to create a binary (`.bin`) file which we can then flash to the MCU.



Installing GNU Tools for ARM Embedded processors

In order to start building some code examples we need a compiler and linker that can compile our code for ARM processors. One of the most popular options for doing this is the GNU Tools for ARM Embedded processors which can be downloaded from <https://developer.arm.com/tools-and-software/open-source-software/developer->

tools/gnu-toolchain/gnu-rm/downloads (<https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads>). After you downloaded and installed it you can find all the tools in the bin folder (for me it is C:\Program Files (x86)\GNU Tools ARM Embedded\6 2017-q2-update\bin). Most C/C++ compilers provide separate compiler, assembler and linker tools. GNU C/C++ compiler also provides these tools as separate packages but they also include a combined tool, GCC. Using GCC we can provide all our source files, header files and linker script and build the binary in one go. However for the purpose of this post we will do it step by step to make sure we understand each step of the process. If you take a look at the tools in the installed bin folder you will notice they all have the "arm-none-eabi-" prefix. This prefix helps to make sure that we cross compile for our MCU and not for our native or another system. For example, it could be that you also have GNU GCC installed for x86 and the path environment variable set, running gcc would then run the x86 compiler and not the ARM compiler. By adding the prefix we can distinguish between them. The first part of the prefix ("arm") indicates the compiler is for ARM processor, the second part ("none") means it does not target any specific system and the last part ("eabi") tells us that it uses the Embedded-Application Binary Interface.

Example code

As an example we will create a basic startup program that will perform some initialization and then call the main() function. Basic startup programs are often written in assembly but to keep things simple we will be using C.

startup.c

```

/*
 * Basic start file example
 * Jan Swanepoel, 2019
 *
 */

// Create references to symbols defined in the linker script
extern unsigned int _data_start;
extern unsigned int _data_end;
extern unsigned int _data_load;
extern unsigned int _bss_start;
extern unsigned int _bss_end;

void startup();           // Function prototype (forward declaration) for
startup function
int main();               // Function prototype for main function

// Below we create an array of pointers which would form our vector table
// We use __attribute__((section(".vectors"))) to tell the compiler that we want the
// array to be placed in a memory section that we call ".vectors"
unsigned int * vectors[2] __attribute__((section(".vectors"))) =
{
    (unsigned int *) 0x20005000,    // Address of top of stack. 20kB = 1024
    x 20 = 20480 bytes = 0x5000
    (unsigned int *) startup      // Address of the reset handler which is
also our startup function
};

// The startup function, address was provided in the vector table
void startup()
{
    volatile unsigned int *src, *dest;

    // Copy data section values from load time memory address (LMA) to their
address in SRAM
    for (src = &_data_load, dest = &_data_start; dest < &_data_end; src++, dest++)
        *dest = *src;

    // Initialize all uninitialized variables (bss section) to 0
    for (dest = &_bss_start; dest < &_bss_end; dest++)
        *dest = 0;

    // Calling the main function
    main();

    while(1);           // Normally main() should never return, but just in case
we loop infinitely
}

```

Then we also create a very basic main() function in main.c:

```
const int a = 7;           // Will be placed in rodata
int b = 3;                 // Will be placed in data (LMA)
int c = 0;                 // Will be placed in bss
int d;                     // Will be placed in bss (but first in COMMON)

int main()
{
    c = a + b;

    while (1);

    return 0;
}
```

Lets first take a look at the startup.c file.

It starts with adding some external references to symbols that are defined in the linker script. You will see them when we look at the linker script file, all that they do is point to some address as their names suggests. The “_data_start” symbol points to the address of where the DATA section in SRAM starts, “_data_end” to where it ends, “_data_load” to where the LMA for the DATA section starts and so forth.

Next up, we forward declare a function called startup() which will be our main entry point and then our main() function which will be called after initialization.

Next we create an array of pointers that will be our vector table. By adding the “**attribute** ((section(“**vectors**”)))” we tell the compiler that we want this array to be in a section called “**vectors**” in our object file. In this array we specify the top of the stack address at index 0 and the address of our startup function at index 1. Since the STM32F103C8 (used in the Blue Pill) has 20kB of SRAM which starts at address 0x2000 0000, the top of the SRAM which is where the top of the stack would be is at address 0x2000 5000.

Then we get to the startup function. The first task of this function is to copy the values of initialized variables (DATA section) from their load memory address in FLASH to their virtual memory address in RAM. This is done with a for loop and using the addresses provided by the external references. The second task of the startup function is to initialize all the un-initialized variables (BSS section), which is as simple as just setting their values to 0. Once this is done, our basic initialization is completed and we pass control over to the main function.

So as an overview, the main tasks of our startup program is to:

- create the vector table to provide the address to the top of the stack and to the reset handler
- copy initialization values to the initialized global variables (DATA section variables)
- zero the uninitialized global variables (BSS section variables)

- call the `main()` function

The `main.c` file is really simple and just declares 1 global constant and 3 global variables followed by the `main` function which adds `a` and `b` and places the result in `c`. The constant `a`, is global and can never have its value changed, therefore it will be stored in FLASH memory in the `RODATA` section. Variable `b` is initialized to a value 3, the value 3 would thus be stored in the `LMA DATA` section and the address of where this variable will exist in SRAM will be in the `VMA DATA` section. Variables `c` and `d` will both end up in the `BSS` section, even though `c` is initialized to 0 it will still go to `BSS` which will anyway have all its values initialized to 0.

Linker Script

The linker takes many object files as input and produces a single output executable file. We use the linker script in combination with the linker to have more control over how the output file should be compiled.

Now lets look at our linker script file:

```

MEMORY
{
    rom (rx) : ORIGIN = 0x08000000, LENGTH = 64K
    ram (rwx) : ORIGIN = 0x20000000, LENGTH = 20K
}

SECTIONS
{
    .text :                               /* Define output file TEXT section */
    {
        *(.vectors)                       /* Vector table */
        *(.text)                          /* Program code */
        . = ALIGN(4);                     /* Make sure data that follows are align
ed to 4 byte boundary */
        *(.rodata)                        /* Read only, section set aside for con
stants */
    } >rom

    .data :                               /* Define output file DATA section */
    {
        _data_start = .;                 /* Get the memory address (VMA) for star
t of section .data */
        *(.data)                         /* Initialized static and global variabl
e values */
        . = ALIGN(4);
        _data_end = .;                   /* Get the memory address (VMA) for end
of section .data */
    } >ram AT >rom                       /* After AT we specify the load-time loc
ation */

    _data_load = LOADADDR(.data);        /* Get the load memory address (LMA) for
section .data */

    .bss :                               /* Define output file BSS section */
    {
        _bss_start = .;                  /* Get memory address of start of bss se
ction */
        *(.bss)                          /* Uninitialized static and global varia
bles */
        *(COMMON)                        /* Uninitialized variables are placed in
COMMON section for object files */
        . = ALIGN(4);
        _bss_end = .;                    /* Get memory address of end of bss sect
ion */
    } >ram
}

```

Our basic linker script consists of two commands, a MEMORY and a SECTIONS

command.

In the MEMORY command we define the memory blocks of our MCU which consists of a ROM block starting at 0x0800 0000 with a length of 64KB and a RAM block starting at 0x2000 0000 with a length of 20KB. The syntax for defining these blocks are:

```
name [(attr)] : ORIGIN = origin, LENGTH = len
```

In our file we picked a name "rom" and provided the "rx" attributes which means this block is readable and executable to create the FLASH memory region. For the SRAM region we used "ram" as name followed by "rwx" attributes to indicate that this block is readable, writable and executable.

The SECTIONS command is used by the linker to map input sections to output sections, and describes how to place the output sections in memory. In our file we have 3 output sections named, ".text", ".data" and ".bss". This means that our output file will have only these 3 sections. As the linker processes through each of the input object files it will look for the sections defined within these 3 output sections and if a match is found it will be added to the relevant output file section. Take the ".vectors" section for example, this is a section we defined in our startup.c code, this means that the array of pointers we defined to be our vector table will be placed here. Another example is the ".text" section, this one we did not explicitly define but the compiler did it in the background when we compiled to an object file. So the linker will place all the ".text" sections from the different input files at this location in the output file. It is also important to note that the order in which we defined the sections matter. At the end of each section we also use the memory blocks we created with the MEMORY command to define where these output sections should live. If you look through these 3 output sections you would also recognize the sections we discussed earlier in the post. One section that might be new is the COMMON section, it refers to global variables which are not initialized to 0 and not yet allocated (like variable 'd'). When objects are linked together, COMMON is merged with BSS which means these variables will then be allocated. The startup code will then also initialize them to 0 like the rest in the BSS section.

In all 3 sections you will find the line, ". = ALIGN(4)" which aligns the location counter (donated with ".") to the next word. This is done to make sure that the values which are placed following this function is word aligned (remember we discussed this earlier in the post). Take the ".text" section for example, just before the start of the ".rodata" section (which will contain the constant values) we make sure that we align the location counter to the next word. This will ensure that the values to follow will be word aligned.

In the ".data" and ".bss" sections you will find some of the external references we used in the startup function in the startup.c file. Here we used the location counter

(".") again to get the start and end addresses of the sections. To get the LMA address for the ".data" section we used the LOADADDR function.

Linker scripts can be difficult to grasp at first but as you spend time with them they slowly starts to get clearer. Luckily it's also not something you need to write every single time, once you have it for your target MCU then you can just reuse it for every new project. The one we wrote here is pretty basic but could be okay for most simple C projects. For C++ projects it might require some tweaking to support some of the more advanced features.

Building our example

So now finally we can get to building our code. Typically we would use a make file to perform the build, but since this is a very small program and to keep it simple we will just use a batch file. This is how our file looks:

```
ECHO OFF
CLS
SET PATH=%PATH%;c:\Program Files (x86)\GNU Tools ARM Embedded\6 2017-q2-update\bin\
ECHO ON

REM Compiling
arm-none-eabi-gcc -O0 -Wall -c -g -mcpu=cortex-m3 -mthumb main.c -o bin\main.o
arm-none-eabi-gcc -O0 -Wall -c -g -mcpu=cortex-m3 -mthumb startup.c -o bin\startup.o

REM Linking
arm-none-eabi-ld -o bin\prog.elf -T stm32f103.ld bin\startup.o bin\main.o
arm-none-eabi-objcopy bin\prog.elf bin\prog.bin -O binary

REM Disassembling
arm-none-eabi-objdump -D -h bin\startup.o > bin\startup.list
arm-none-eabi-objdump -D -h bin\main.o > bin\main.list
arm-none-eabi-objdump -D -h bin\prog.elf > bin\prog.list
arm-none-eabi-nm --numeric-sort bin\prog.elf

pause
```

At the top we add the GNU ARM Tools path to the PATH environment variable (will only be valid for this session) so that we can easily run the different tools.

Next the actual build process starts with compiling our source files to object files.

This is done with "gcc" and the following options:

- `Ox`, which sets the optimization level to `x`, we used `0` here which means we want no optimization. Normally it would be good to use some optimization but in this case we want to disassemble the objects later and therefore want as little as

possible optimization.

- `Wall`, turns all compile warnings on.
- `c`, tells GCC to only compile and not to perform the linking (remember GCC normally automatically performs linking).
- `g`, adds extra debugging information to allow us to debug our program later.
- `mcpu=cortex-m3`, indicates to GCC that we are compiling for a Cortex-M3 CPU.
- `mthumb`, tells the compiler to use the Thumb instruction set.

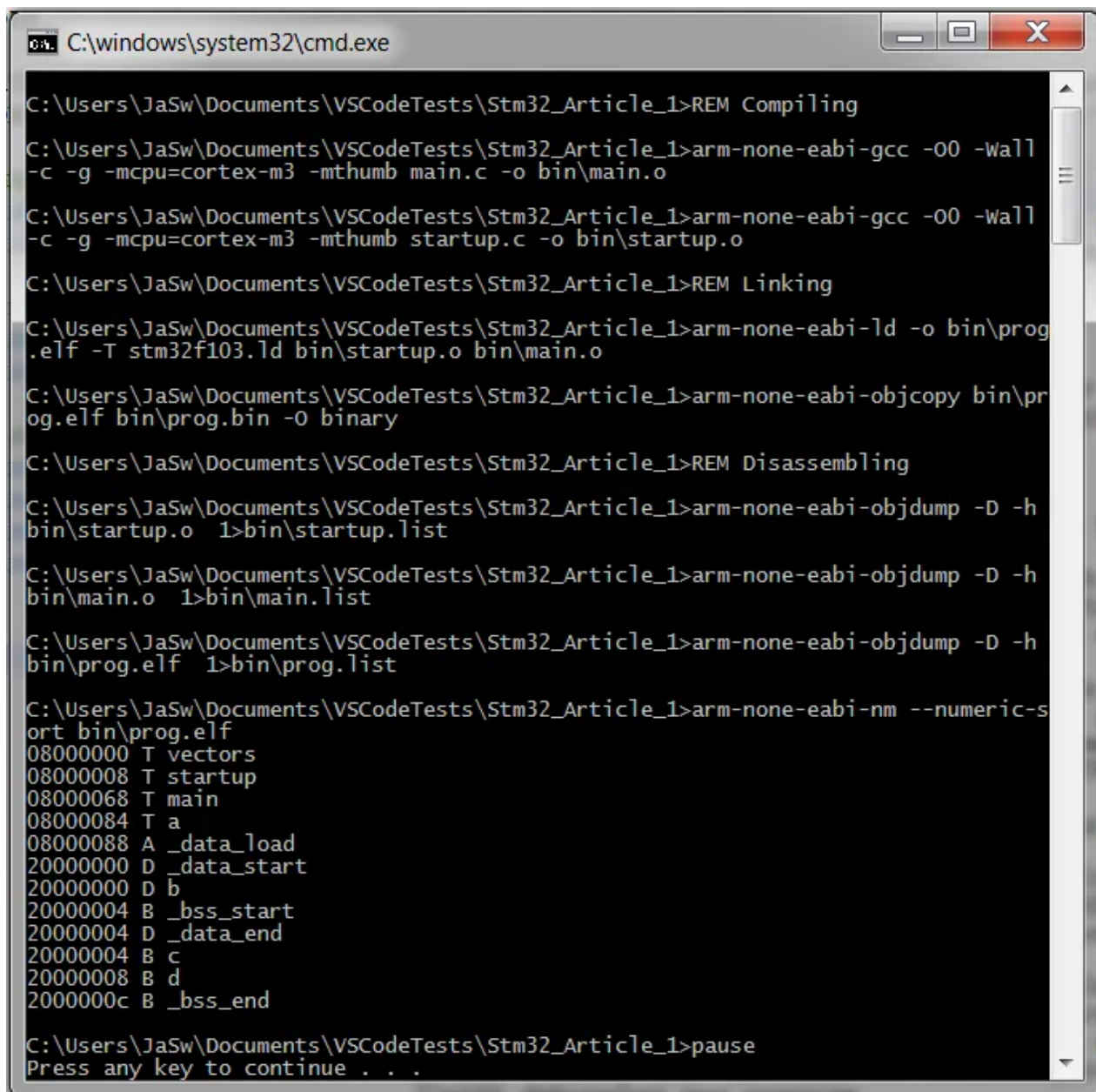
For more details on the options you can also have a look here (<https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>).

After the compiling is done the linking is performed with `ld`. We pass the linker script file after the `-T` option followed by the object files we just compiled. The output from the linker is the `prog.elf` executable file. Earlier we mentioned that we need a binary file when flashing to the MCU and therefore we convert the `.elf` to a `.bin` file with `objcopy`.

In the last couple of lines we disassemble the object and binary files again with `objdump` and `-D` option. The `-h` option also adds a header to the disassembled files which provides a summary of the memory layout. We only do this so that we can inspect the object and executable files to make sure everything was compiled and linked correctly.

Then finally the `nm` tool at the end just writes the symbols and their addresses in the `.elf` file to the console, this is also just to help with debugging our program.

If you run the batch file then there should be no warnings or errors and look something like this:



```
C:\windows\system32\cmd.exe

C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>REM Compiling
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-gcc -O0 -Wall
-c -g -mcpu=cortex-m3 -mthumb main.c -o bin\main.o
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-gcc -O0 -Wall
-c -g -mcpu=cortex-m3 -mthumb startup.c -o bin\startup.o
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>REM Linking
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-ld -o bin\prog
.elf -T stm32f103.ld bin\startup.o bin\main.o
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-objcopy bin\pr
og.elf bin\prog.bin -O binary
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>REM Disassembling
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-objdump -D -h
bin\startup.o 1>bin\startup.list
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-objdump -D -h
bin\main.o 1>bin\main.list
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-objdump -D -h
bin\prog.elf 1>bin\prog.list
C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>arm-none-eabi-nm --numeric-s
ort bin\prog.elf
08000000 T vectors
08000008 T startup
08000068 T main
08000084 T a
08000088 A _data_load
20000000 D _data_start
20000000 D b
20000004 B _bss_start
20000004 D _data_end
20000004 B c
20000008 B d
2000000c B _bss_end

C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1>pause
Press any key to continue . . .
```

Here we can see all the steps in the building process and at the end the symbols and their addresses in our prog.elf file. As expected we see the vector table starting at address 0x0800 0000 followed by our startup function, main function, constant named "a" and the start of DATA section values (remember we defined _data_load in the linker script). All these are in the address range of our FLASH memory. Then following along we see the start of the DATA section in SRAM at address 0x2000 0000 and this also where global variable "b" will live. Next comes the address to the end of the DATA section and start of the BSS section. We can then see that uninitialized global variables "c" and "d" and the end of the BSS section.

As a first overview we can see that our linker script worked correctly. We could now also open the .list files that we created with a text editor to view them in more detail. As an example we have the first few lines of the prog.list file below:

```

1
2 bin\prog.elf:      file format elf32-littlearm
3
4 Sections:
5 Idx Name          Size      VMA      LMA      File off  Algn
6  0  .text          00000088  08000000  08000000  00010000  2**2
7      CONTENTS, ALLOC, LOAD, READONLY, CODE
8  1  .data          00000004  20000000  08000088  00020000  2**2
9      CONTENTS, ALLOC, LOAD, DATA
10  2  .bss           00000008  20000004  0800008c  00020004  2**2
11      ALLOC
12  3  .debug_info    00000152  00000000  00000000  00020004  2**0
13      CONTENTS, READONLY, DEBUGGING
14  4  .debug_abbrev  000000e8  00000000  00000000  00020156  2**0
15      CONTENTS, READONLY, DEBUGGING
16  5  .debug_aranges 00000040  00000000  00000000  0002023e  2**0
17      CONTENTS, READONLY, DEBUGGING
18  6  .debug_line    0000009d  00000000  00000000  0002027e  2**0
19      CONTENTS, READONLY, DEBUGGING
20  7  .debug_str      00000112  00000000  00000000  0002031b  2**0
21      CONTENTS, READONLY, DEBUGGING
22  8  .comment       0000007f  00000000  00000000  0002042d  2**0
23      CONTENTS, READONLY
24  9  .ARM.attributes 00000033  00000000  00000000  000204ac  2**0
25      CONTENTS, READONLY
26 10  .debug_frame   00000058  00000000  00000000  000204e0  2**2
27      CONTENTS, READONLY, DEBUGGING
28
29 Disassembly of section .text:
30
31 08000000 <vectors>:
32 8000000: 20005000    andcs   r5, r0, r0
33 8000004: 08000009    stmdbaq r0, {r0, r3}
34
35 08000008 <startup>:
36 8000008: b580      push    {r7, lr}
37 800000a: b082      sub     sp, #8
38 800000c: af00      add     r7, sp, #0
39 800000e: 4b11      ldr     r3, [pc, #68] ; (8000054 <startup+0x4c>)
40 8000010: 607b      str     r3, [r7, #4]

```

The first part called “Sections” is the header which was added by the “-h” command line option. It lists the different sections with details about their size, address, offset and alignment. Looking at the “.data” section in the header we can again see the VMA and LMA address which off-course matches with the “nm” output we saw when building the program.

After the header we see again the vectors section which holds the vector table. As expected the value of the first address points to the address of the top of the stack (which we provided in the linker script). The value of the second address in the vector table holds the address of the startup function, but if we cross reference this it might seem to be wrong... The address of our startup function is 0x0800 0008 but the vector table points to address 0x0800 0009, the reason for this is because we are using the

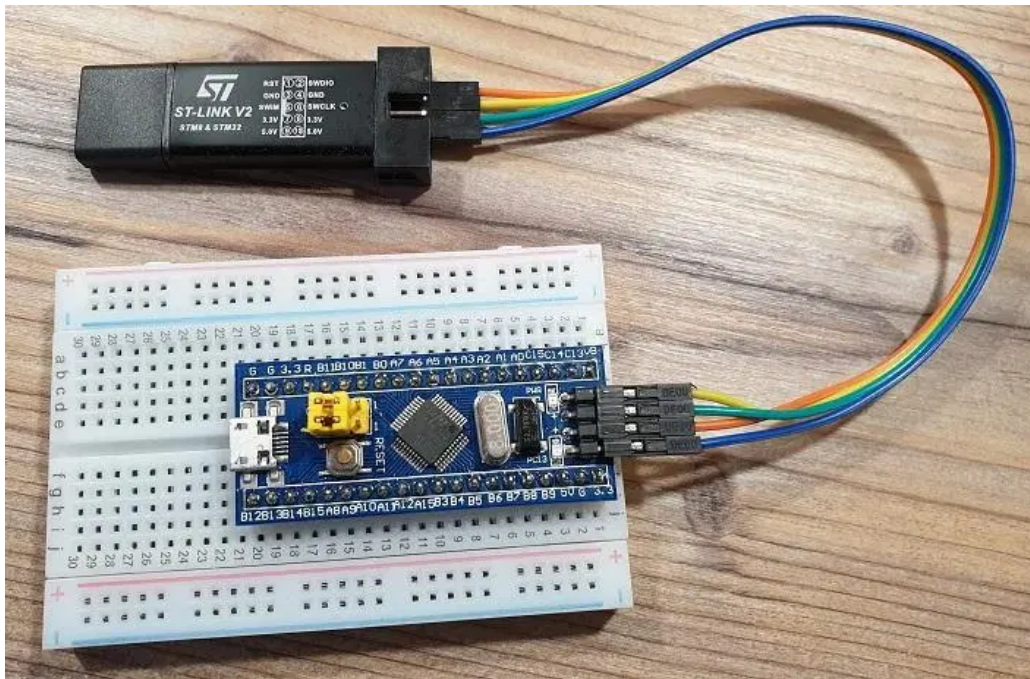
Thumb instruction set. Remember from our discussion regarding the use of the Thumb instruction set we said that on a branch instruction the LSB needs to be 1. This is exactly what is happening here and why we will always see this address + 1 value.

Looking at the startup function we can start to see the assembly instructions and their location in memory, here the first ones we can see are all 16 bit instructions due to their half word separation in memory.

As you can see these disassembled files can be very handy to determine if our build process went as expected or just to get a better idea of what our code is really doing. If you are interested in learning assembly these files are also very helpful.

Flashing the MCU

There are more than one way to flash your program to the microcontroller but for this project I preferred the ST-Link programmer since we can also use it to debug our program once it's flashed. These programmer can be found really cheap online and works really well.



In combination with the ST-Link programmer I also use the ST-Link application from Texan (<https://github.com/texane/stlink/releases/tag/1.3.0>). You might also need to install the ST-Link driver also which can be found here (https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-utilities/stsw-link009.html).

The programmer pin-out is marked clearly on the back and all we need to do is to connect them up with the board. If you look at the picture above yellow is GND, green

is SWCLK, orange is SWDIO and blue 3.3V. Once you have everything setup you can use the “st-flash” utility to flash your program. For this task I also created a small batch file:

```
ECHO OFF
SET PATH=%PATH%;E:\Stuff\Programs\STLink\TexaneVersion\stlink-1.3.0-win64\bin\
SET PATH=%PATH%;c:\Program Files (x86)\GNU Tools ARM Embedded\6 2017-q2-update\bin\
set WorkingDir=C:\Users\JaSw\Documents\VSCodeTests\Stm32_Article_1\
ECHO ON

arm-none-eabi-size --format=berkeley %WorkingDir%bin\prog.elf
st-flash write %WorkingDir%bin\prog.bin 0x8000000

pause
```

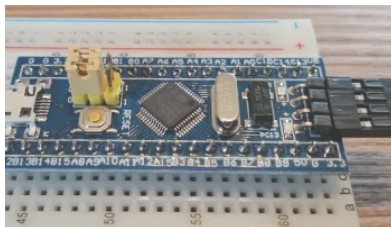
The “arm-none-eabi-size” tool provides us with the sizes of the different sections and is purely for extra information. The real flashing happens with the “st-flash” tool which we call with the “write” argument and provide it with our binary file (you would need to edit the path to point to your binary) and location in memory to flash it to.

So whats next...

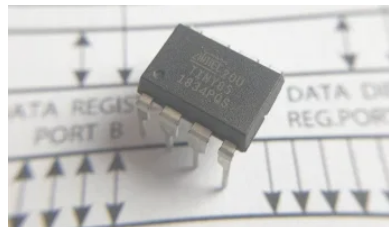
So now that our code is build and flashed to the microcontroller, how do we know it works. The example program is very simple and does not flash any LED or output anything, so we would not really see anything. We could fix this by just modifying the example code to flash a LED on one of the pins or to use a debugger and step through the instructions. But since this post has grown a bit longer than expected I would rather do a follow-up post in the future (maybe when using libopenocm3) and show how this can be done.

This brings me to the end of this post, I hope you found this topic just as exciting and interesting as I did and that it can be useful for your current or future projects.

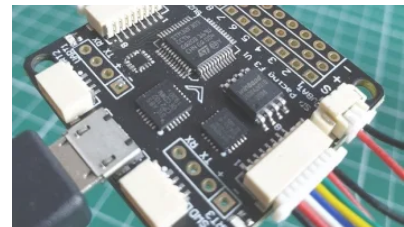
Related



(/getting-started-with-stm32f1?relatedposts_hit=1&



(/going-bare-metal-on-the-attiny85?relatedposts_hit=1&



(/low-cost-stm32f3-board?relatedposts_hit=1&

relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2

Low cost STM32F3 board
(/low-cost-stm32f3-
board?relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&

relatedposts_position=0&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=0&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=0&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=0&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=0&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=0&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=0)
17/06/2018
In "Electronics"

relatedposts_origin=1809&
relatedposts_position=1&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=1&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=1&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=1&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=1&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=1&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=1)
14/02/2021
In "Code"

relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2&
relatedposts_hit=1&
relatedposts_origin=1809&
relatedposts_position=2)
18/09/2020
In "Arduino"

← HIGH VOLTAGE ARDUINO PINS
(HTTPS://KAROOZA.NET/HIGH-
VOLTAGE-ARDUINO-PINS)

HOW TO INTERFACE A PS2
KEYBOARD →
(HTTPS://KAROOZA.NET/HOW-TO-
INTERFACE-A-PS2-KEYBOARD)

Leave a Reply

You must be logged in (https://karooza.net/wp-login.php?redirect_to=https%3A%2F%2Fkarooza.net%2Fgoing-bare-metal-on-stm32) to post a comment.

Tags

[#IMU \(https://karooza.net/tag/imu\)](https://karooza.net/tag/imu)[#MPU6050 \(https://karooza.net/tag/mpu6050\)](https://karooza.net/tag/mpu6050)[3D \(https://karooza.net/tag/3d\)](https://karooza.net/tag/3d)[486 \(https://karooza.net/tag/486\)](https://karooza.net/tag/486)[accelerometer \(https://karooza.net/tag/accelerometer\)](https://karooza.net/tag/accelerometer)[ADF4351 \(https://karooza.net/tag/adf4351\)](https://karooza.net/tag/adf4351)[Arduino \(https://karooza.net/tag/arduino\)](https://karooza.net/tag/arduino)[atmega328p \(https://karooza.net/tag/atmega328p\)](https://karooza.net/tag/atmega328p)[Beplanning \(https://karooza.net/tag/beplanning\)](https://karooza.net/tag/beplanning)[Bitbucket \(https://karooza.net/tag/bitbucket\)](https://karooza.net/tag/bitbucket)[Blue pill \(https://karooza.net/tag/blue-pill\)](https://karooza.net/tag/blue-pill)[Broken Sword \(https://karooza.net/tag/broken-sword\)](https://karooza.net/tag/broken-sword)[complimentary filter \(https://karooza.net/tag/complimentary-filter\)](https://karooza.net/tag/complimentary-filter)[Detroit \(https://karooza.net/tag/detroit\)](https://karooza.net/tag/detroit)[Drone \(https://karooza.net/tag/drone\)](https://karooza.net/tag/drone)[Duke Nukem 3D \(https://karooza.net/tag/duke-nukem-3d\)](https://karooza.net/tag/duke-nukem-3d)[Eclipse \(https://karooza.net/tag/eclipse\)](https://karooza.net/tag/eclipse)[ESP8266 \(https://karooza.net/tag/esp8266\)](https://karooza.net/tag/esp8266)[gyroscope \(https://karooza.net/tag/gyroscope\)](https://karooza.net/tag/gyroscope)[Leer \(https://karooza.net/tag/leer\)](https://karooza.net/tag/leer)[Linux \(https://karooza.net/tag/linux\)](https://karooza.net/tag/linux)[makefile \(https://karooza.net/tag/makefile\)](https://karooza.net/tag/makefile)[Matrikse \(https://karooza.net/tag/matrikse\)](https://karooza.net/tag/matrikse)[MinGW \(https://karooza.net/tag/mingw\)](https://karooza.net/tag/mingw)[Nexus 5X \(https://karooza.net/tag/nexus-5x\)](https://karooza.net/tag/nexus-5x)[NodeMCU \(https://karooza.net/tag/nodemcu\)](https://karooza.net/tag/nodemcu)[OpenTTD \(https://karooza.net/tag/openttd\)](https://karooza.net/tag/openttd)[Pentium \(https://karooza.net/tag/pentium\)](https://karooza.net/tag/pentium)[PXFMini \(https://karooza.net/tag/pxfmini\)](https://karooza.net/tag/pxfmini)[Raspberry Pi \(https://karooza.net/tag/raspberry-pi\)](https://karooza.net/tag/raspberry-pi)[RetroPie \(https://karooza.net/tag/retroPie\)](https://karooza.net/tag/retroPie)[Self-balancing robot \(https://karooza.net/tag/self-balancing-robot\)](https://karooza.net/tag/self-balancing-robot)[Selfoon \(https://karooza.net/tag/selfoon\)](https://karooza.net/tag/selfoon)[Signal Generator \(https://karooza.net/tag/signal-generator\)](https://karooza.net/tag/signal-generator)[Starcraft \(https://karooza.net/tag/starcraft\)](https://karooza.net/tag/starcraft)[STM32duino \(https://karooza.net/tag/stm32duino\)](https://karooza.net/tag/stm32duino)[STM32F1 \(https://karooza.net/tag/stm32f1\)](https://karooza.net/tag/stm32f1)[Tomb Raider \(https://karooza.net/tag/tomb-raider\)](https://karooza.net/tag/tomb-raider)[Toolchain \(https://karooza.net/tag/toolchain\)](https://karooza.net/tag/toolchain)[Transport Tycoon \(https://karooza.net/tag/transport-tycoon\)](https://karooza.net/tag/transport-tycoon)

Tutoriaal (<https://karooza.net/tag/tutoriaal>)

Tutorial (<https://karooza.net/tag/tutorial>)

USB Mikroskoop (<https://karooza.net/tag/usb-mikroskoop>)

Vergrootglas (<https://karooza.net/tag/vergrootglas>)

Visual Studio (<https://karooza.net/tag/visual-studio>)