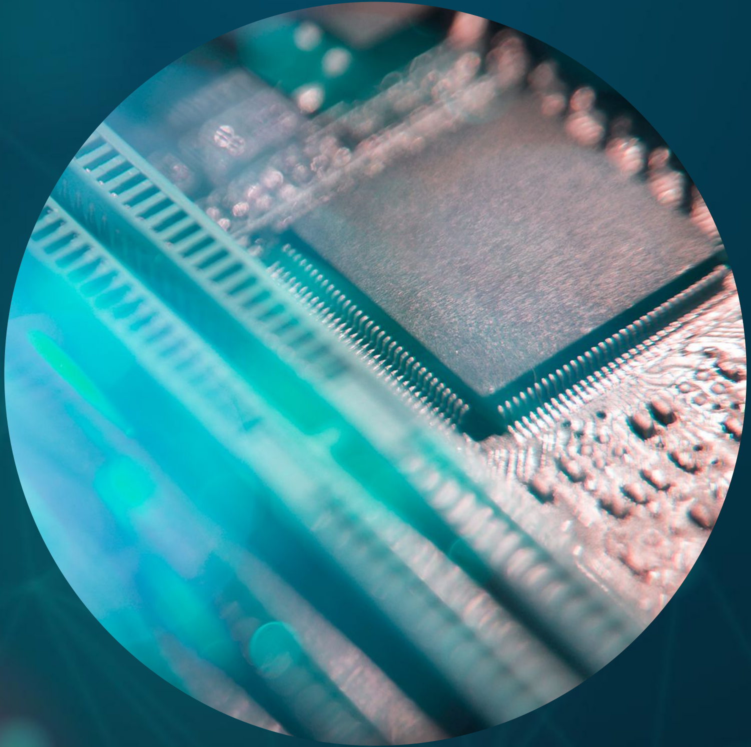
IMT SCHOOL

ARM Based Microcontroller

Story of Flashing

Lecture 19



Advanced
RISC
Machines

IMT SCHOOL

Building Process 01

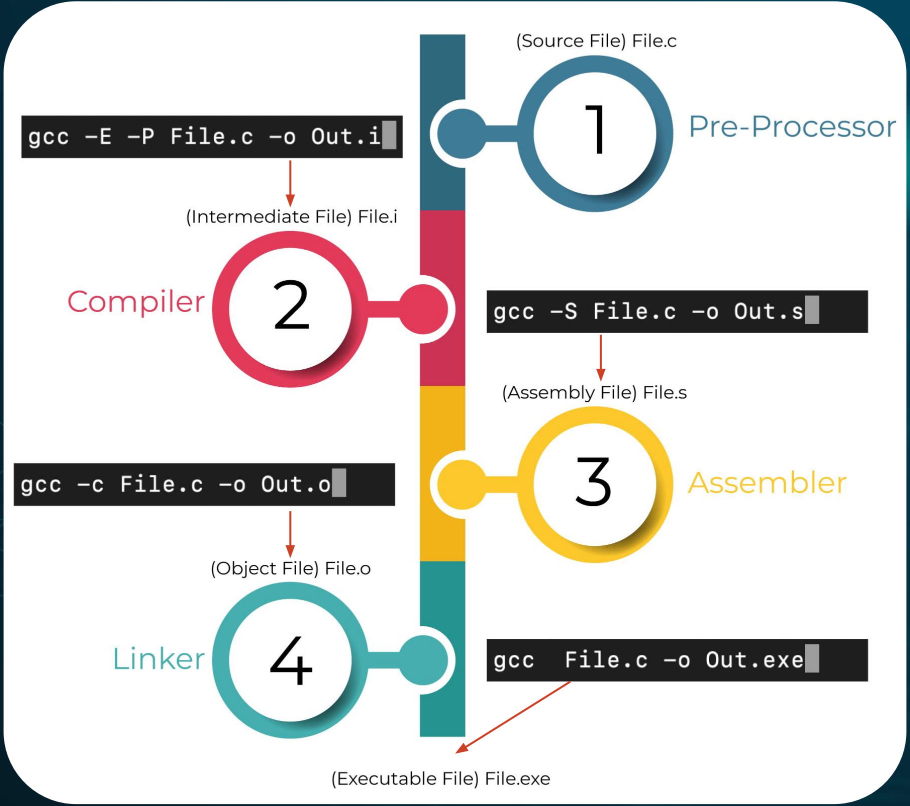
Flashing

C Building Process

C is a high-level language and it needs a compiler to convert it into an executable code so that the program can run on our machine. Starting from the source file written in C, till the executable file which is zeros and ones only, a set of tools work serially on the file, this set is called the toolchain.
There are many tools in the toolchain, but we will discuss the four main tools which exist in most toolchains.

The four main tools are:
1. Preprocessor
2. Compiler
3. Assembler
4. Linker

C Building Process



```
gcc -E -P File.c -o Out.i
```

```
gcc -S File.c -o Out.s
```

```
gcc -c File.c -o Out.o
```

```
gcc  File.c -o Out.exe
```

Pre-Processor

This is the first phase through which source code is passed. Pre-processing is mandatory in C. Pre-processing will handle '#' directives (except for #pragma).
This phase includes:

1. Removal of Comments.
2. Expansion of Macros.
3. Expansion of the included files.
4. Conditional compilation.

The preprocessor output is called the intermediate file and it is a file written in C but without any ‘**#**’ directive. To get the preprocessor output (intermediate file) we need to stop the compilation process after the preprocessor tool. This could happen by writing the following instruction in the command window (**cmd**):
(gcc –E file.c –o out.i).

Compiler

Compiler converts a high-level language into the specific instruction set of the target CPU. Then the compiler do the following steps:

1. Parse text (lexical + syntactical analysis).
2. Do language specific transformations.
3. Translate to internal representation units (IRs).
4. Optimization (reorder, merge, eliminate).
5. Replace IRs with pieces of assembler language.

The compilation output file is called Assembly file because it is a file written in Assembly language. To generate the Assembly file we need to stop the building process after the compiler tool. To stop the building process after the compiler tool we need to write the following instruction in the command window (**cmd**):
(gcc –S file.c –o out.s).

Assembler

Assembler translates assembly to binary. It creates a file called object file which is written in binary codes. At this phase, only existing code is converted into machine language. The Assembler output file is called object file. To generate 192 this object file we need to stop the building process after the Assembler tool by writing the following instruction in the command window (**cmd**):

(gcc -c file.c -o out.o).

Linker

Linker puts binary together with startup code and required libraries.
The linker makes an operation that we can call object verification which is done in two steps:

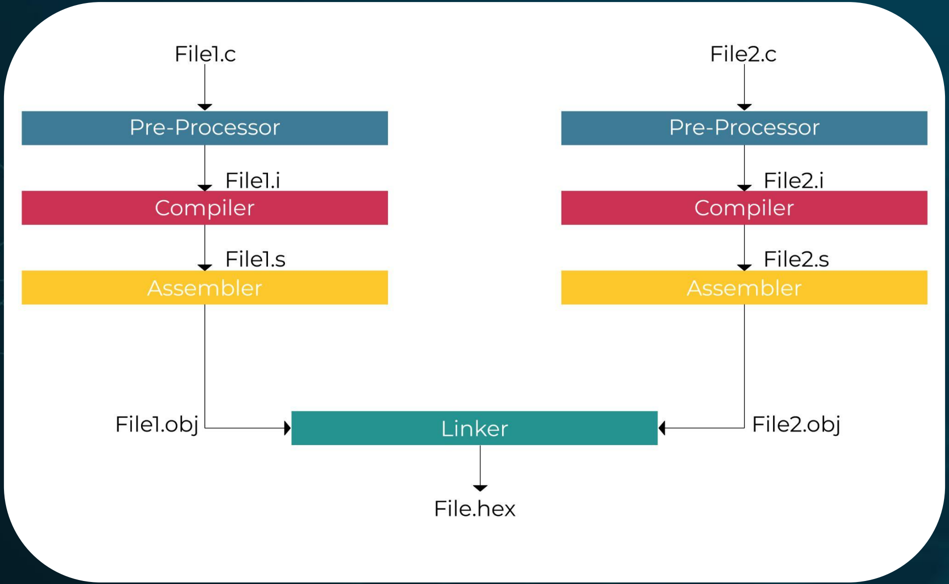
1. Ensure that every “Needed” object in a symbol table generated by the compiler is “Provided” in another symbol table.
2. Ensure that every “Provided” object is provided only once.

The Linking output file is called the executable file and it is called by this name because it is the file which the processor will execute (usually). To get this executable file we need to write the following instruction in the command window (**cmd**):
(gcc file.c –o out.exe).

Linker

Difference between object file and executable file

Object file and Executable file are both written in binary (0&1) but the difference is clear when we have more than one file (file1.c,file2.c,.....) in this case every c file has its own object file but one executable file for the whole project



Memory 02

Flashing

Memory Sections

Flash memory sections

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. A section also has a type that may be marked as loadable, which means that the contents should be loaded into RAM memory when the output file is running. A section may be allocatable, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out).

The executable file is divided into four main sections every section has name, address size and type. These sections differ from a toolchain to another, but there are 4 sections that are most common among toolchains.

Memory Sections

Flash memory sections

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. A section also has a type that may be marked as loadable, which means that the contents should be loaded into RAM memory when the output file is running. A section may be allocatable, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out).

The executable file is divided into four main sections every section has name, address size and type. These sections differ from a toolchain to another, but there are 4 sections that are most common among toolchains.

Memory Sections

1- (.data) section
This segment stores all global and static variables that have defined initial values different than zero.

2- (.bss) section
Here only resides a piece of information about the required size of non initialized global and static variables.

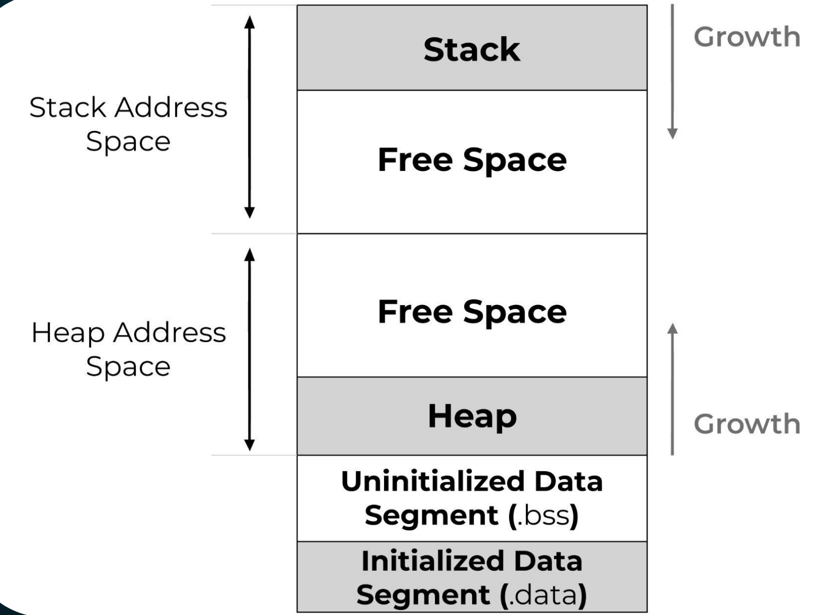
3- (.rodata) section
Here we can find the global constant variables.

4- (.text) section
Here we can find the code.

5- (.init) section
Here we can find the startup code, will be discussed later in this chapter.

RAM Memory Sections

RAM memory, also called Data memory, is the volatile memory used to store the data and variables that change throughout the program, those variables can have different storage classes so we can find different segments (sections) in the RAM.



RAM Memory Sections

1- (.data) section

This segment stores all global and static variables that have defined initial values different than zero. The segment’s size is determined by the size of the variables and is known at compile time. This segment is copied from the non-volatile memory it is initially stored (Flash) to RAM.

2- (.bss) section

This segment stores all global and static variables that are initialized to zero or not initialized at all. The memory locations of variables stored in this segment are initialized to zero before the execution of the program starts.

3- Heap section

This segment is used for dynamic memory allocation. When malloc, calloc, realloc is used for memory allocation, memory is allocated from this section.

4- Stack section

This segment is used for storing the local variables.

µC Memory 02

Flashing

μC Memory

System memory(ROM)

1. It has a size of 2KB.
2. It starts at address 0x1FFF F000.
3. It ends at address 0x1FFF F7FF.
4. It stores the ST booltloader.
5. Read-only memory.
6. By default, the microcontroller will not execute any code from this memory, but you can configure the microcontroller to boot or execute bootloader from this memory using boot pins.

Option bytes memory

1. It has a size of 16 bytes.
2. It is not for general use.
3. It has some flags which control the access to the flash memory.

Startup Code 03

Flashing

Startup code

Startup code is a small block of code (at least part of it is written in assembly language) that prepares the way of execution of software written in a high-level language.

The startup code is used to set up data memory such as global data variables. It also zero initializes part of the data memory for variables that are uninitialized at load time. For applications that use C functions like malloc(), the C startup code also needs to initialize the data variables controlling the heap memory. After this initialization, the C startup code branches to the beginning of the **main()** program.

Flashing 04

Flashing

Flashing Techniques

The flashing process is the process of updating the content of the flash memory, and since the flash is used to store the program code, the process is
also called Programming, Burning. Recall that flash is a non-volatile memory which
 consists of floating gate mosfets, this kind of mosfets needs high power to change its state, this high power can't be provided by the processor, so that it is a must to use an extra hardware circuit that can provide this high power to access the flash memory, this circuit is called the flash driver. So now the process of programming is obvious, we prepare our executable file then send it to the flash driver, the flash driver in turn will write it to the flash. Depending on the position of the flash interface, different programming techniques are found:

Flashing Techniques

- **Off-Circuit Programming.**

This is the basic old technique, here the flash driver is external outside the microcontroller (MC) and the flash memory programming pins are connected to some (MC) pins to be programmed through. A device called burner is used, this burner is usually a hardware (HW) kit that sometimes has a socket on which the (MC) can be placed, or simply connected to the required pins with jumpers. This burner kit includes the flash driver, it also includes the communication port with the computer i.e., USB port, and uses the computer USB power.

Flashing Techniques

So, the task of the burner is to get the executable file from the computer via its communication port, then it applies the required high power on the flash memory as if it uploads the file to flash memory. When the burner finishes uploading the file, the microcontroller can now be removed and connected to the application circuit.

So, to summarize this technique, to program the microcontroller, the microcontroller shall be removed from its application circuit, then connected to the burner which in turn uploads the program to the flash memory, and when it finishes, the microcontroller can now go back to its application circuit and start working, hence called off circuit programming as it requires removing the microcontroller from its application circuit. But what if the process of removing the microcontroller from its application circuit for reprogramming is not that easy?! For example, in firmware update applications, where the new program shall be uploaded to a lot of microcontrollers inside their machines, this will be a big time and effort consuming task, so that a better technique has arisen. This technique is called In-Circuit Programming.

Flashing Techniques

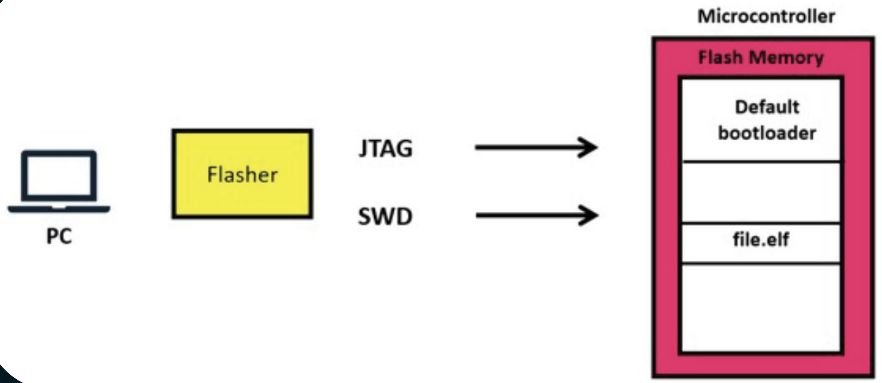
- **In-Circuit Programming.**

It’s Also called In System Programming (ISP), here the microcontroller has an internal flash driver, that can generate any necessary programming power from the system's main power supply to provide the required high power for the flash programming which is inside the (MC) also, so the flashing process is done inside the (MC) which means there is no need to remove the (MC) from the system circuit for programming, hence its name.

Of course this feature has provided a lot of advantages as now the (MC) can be programmed while installed in a complete system, rather than requiring the chip to be programmed prior to installing it into the system, hence it allows firmware updates to be delivered to the on-chip flash memory of microcontrollers without requiring specialist programming circuitry on the circuit board, which simplifies design work, also it allows manufacturers of electronic devices to integrate programming and testing into a single production phase, and save money.

Flashing Techniques

But how to communicate with this on-chip flash driver I.e., how to send it the required executable file? Firstly, the flash driver is needed to communicate with the external world to get the required application code, this is done by serial communication, so the flash driver manufacturer defines one or more communication protocols through which the flash driver can interface, for our stm32f103 there are 2 protocols that are provided to communicate with the in-circuit flash programmer: JTAG and SWD (Serial Wire Debug).



Flashing Techniques

In our system the executable file will be sent to the in-circuit flash driver through one of these protocols then the flash driver will apply the required power to program the flash memory, but how to send the executable file from the computer using the USB protocol to the flash driver by the SWD? Again, the answer is that there must be a converter that translates the USB protocol to the SWD protocol, and this will be the only function of the external device that will be used during the programming process. This is the technique we use; we used the ST-Link programmer and debugger as shown in the last figure. One end connects into the computer. This allows us to transfer the executable file from the computer to the USBASP. The other end of the ST-LINK normally gets connected to 4 wires, which can then get hooked up easily to the programming pins on the KIT. Note that until this moment there is an extra device is used to implement the flashing process, as we needed a translator to send the executable file from the computer to the in-circuit flash driver, and this device is target-specific which means that changing the microcontroller to another one with different flash driver communication protocol will result in changing the translator device, this will be a big headache if it is required to reprogram a number of different microcontrollers in the same system, i.e., a car with 100 ECUS (electronic control units), this is one of the main reasons why the bootloader is used.

Flashing Techniques

- **Bootloader.**

In App Programming(IAP)

(IAP) allows the user to re-program the flash memory while the application is running. Nevertheless, part of the application has to be been programmed in the flash memory using (ICP) previously.

Steps:

- receive the file.
- Flash it (page by page).

There is two Applications APP1 and File.elf, We need a flash driver then the APP1 must be flashed ICP firstly, now we can use IAP.

Flashing Techniques

Booting (also known as booting up) is the initial set of operations that a computer system performs when electrical power is switched on. Bootloader Anyone who has turned on a computer might be familiar with the boot-up sequence as the computer flashes lines of text on screen before the Windows logo appears, what you are seeing is a bootloader in action, loading essential software to get the minimum running on the processor chip before higher-level software can run. As its name implies it can both "boot" the operating system or whatever application is to be run, and also as a second function provide a "loader" capability for developers.

Bootloader is a small OS, or application, designed to download firmware in MCU's internal or external Flash Memory).

Why do we need a Bootloader?

- To fix Bugs.
- Updating system with new features.

Flashing Techniques

Where are they stored?

They reside In ROM, or in the Flash Memory of MCU. (write protected). So, you can’t destroy them.

Bootloader Requirements:

- Needs Flash Driver (FPEC). “Responsible for program/erase flash”.
- Selects a communication Protocol (CAN). “to download programming data into memory”.
- Good design.

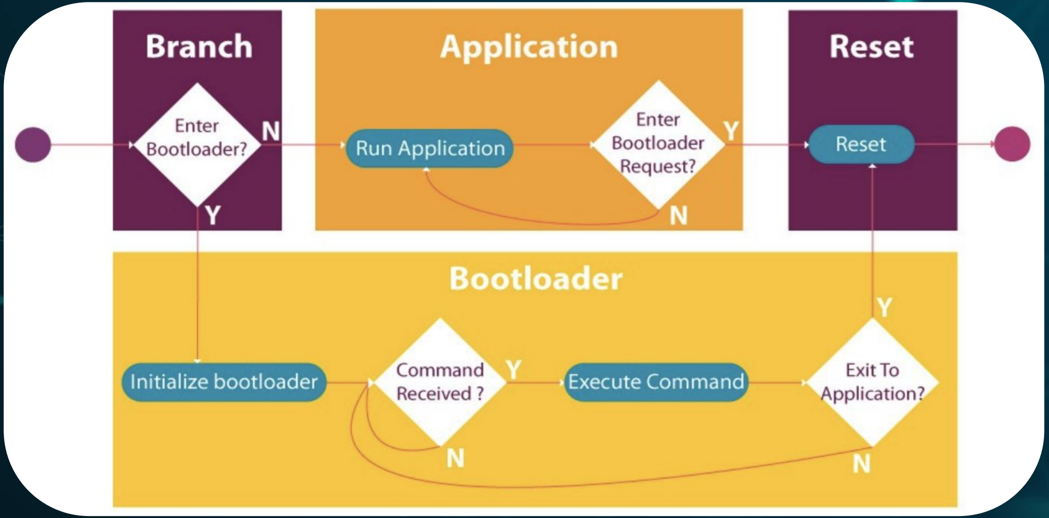
In embedded systems, This task can be simplified as the following:

1. The processor initializes a communication port in the microcontroller to make it ready to receive data.
2. A computer starts sending the application code through this communication port to our microcontroller.
3. the processor reads the data, then sends it to the in-circuit flash driver which in turn writes this data in the flash, eventually the target controller is updated with the code. (Flash the file page by page).

Flashing Techniques

Bootloaders are responsible for:

- Flash new application.
- Run Application.
- Flash new Bootloader.
- Select Application (if we have many applications to select).



IMT SCHOOL

STM32

Is AWESOME

IMT SCHOOL

THANKS!

Do you have any questions?

www.imtschool.com

www.facebook.com/imaketechnologyschool/