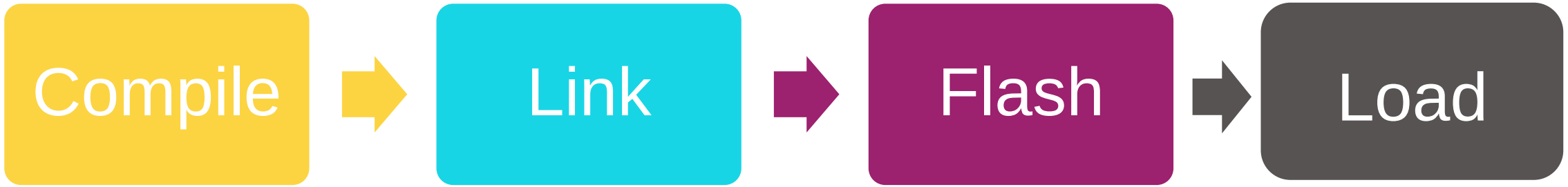
Embedded Systems Course - Makefile essentials

Prakash Balasubramanian

13/April/2024

Source to binary – Flow (RECAP)



C/C++ to OBJ
ASM to OBJ

OBJ to ELF

ELF programmed to Microcontroller memories

ELF sections loaded from flash to selective memories

Site of action (RECAP)

Host PC with cross-compiler for target CPU (e.g. ARM Cortex-M0)

Compilation, Linking and ELF generation

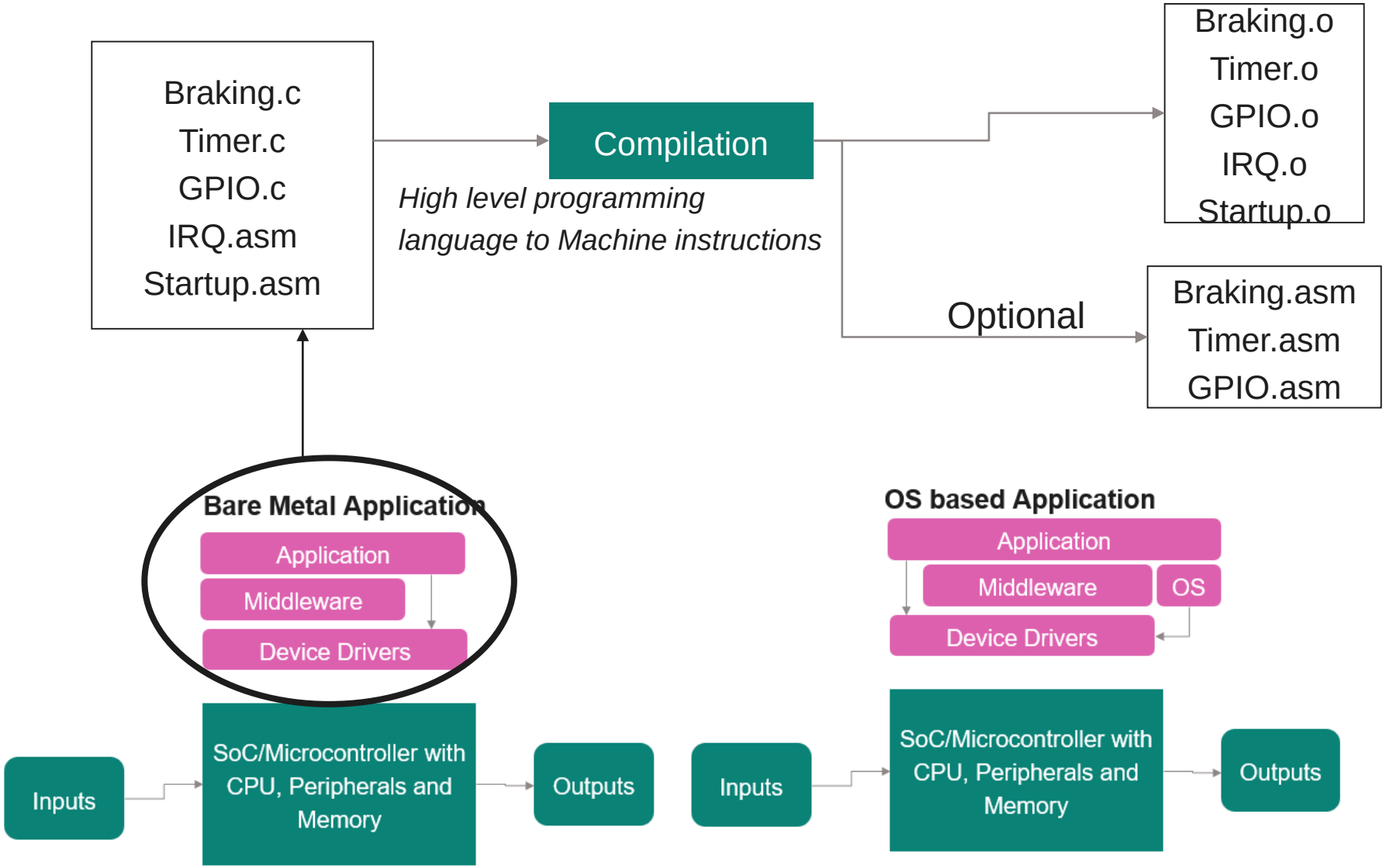
This debugger box helps in programming ELF sections of your application to Non-Volatile memories of the microcontroller

When you power-up the CPU

The startup software

1. Copies data sections of your application from flash to memory,
2. Clears BSS addresses
3. Optionally copies time critical program sections from flash to SRAM
4. Passes control to your main()

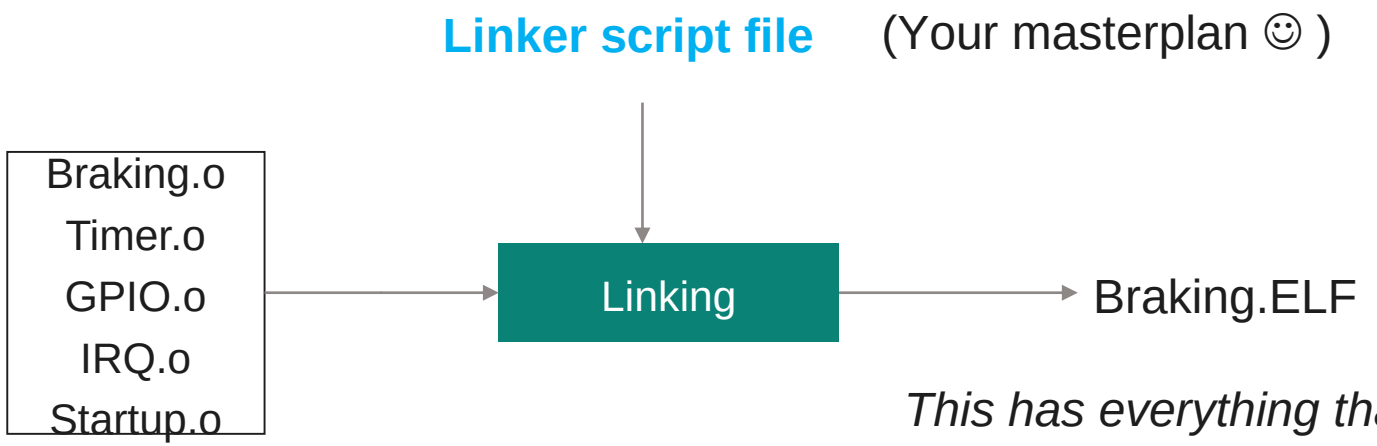
Source code compilation (RECAP)



The instructions and data in these object files are more or less ready for flashing.

BUT where must they be stored? What should be their address in memory?

Object linking (RECAP)



This has everything that your program loader needs to know!

Before Linking

func1()
func2()
func3()
Data1
Data2

Linking

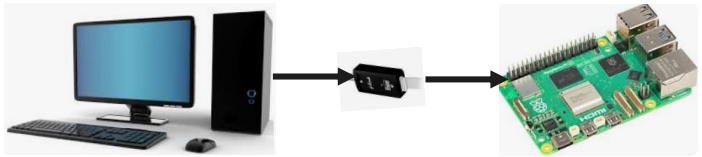
After Linking

func1() 0x20000010 – 0x20001FE
func2() 0x20000200 – 0x200005CB
func3() 0x200005D0 – 0x200005F0
Data1 0x80000000
Data2[8] 0x80000004 – 0x80000023

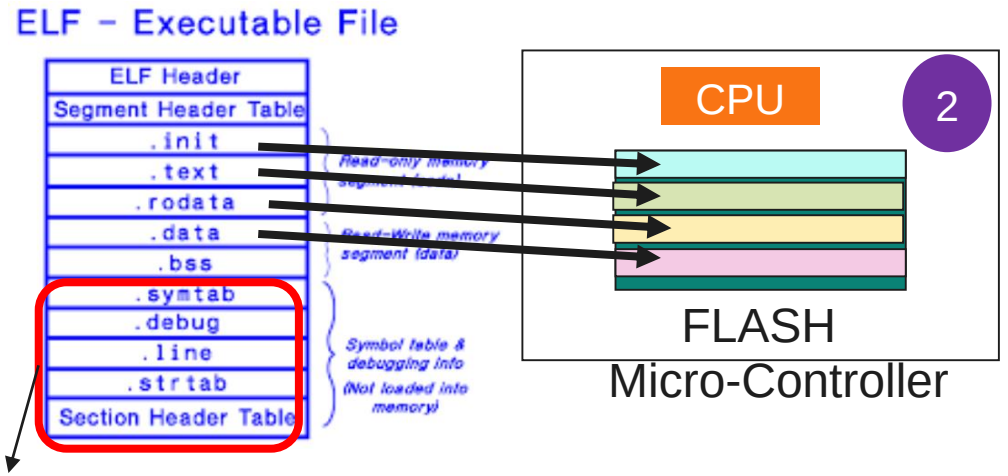
You specified the range in the linker script!

The linker has only followed your instructions faithfully.

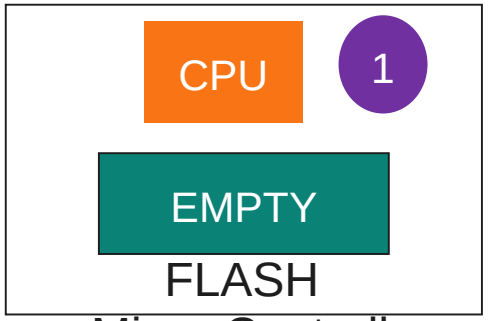
Putting them all together (RECAP)



After Flashing your application



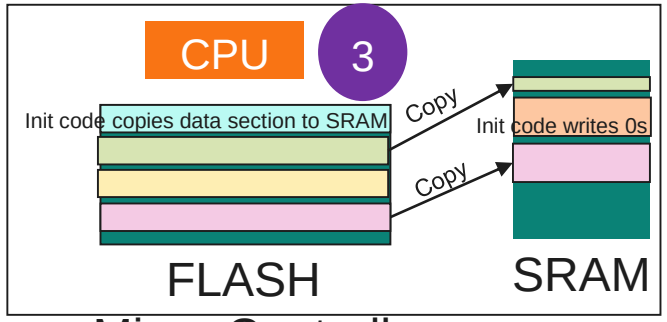
Your brand new development board



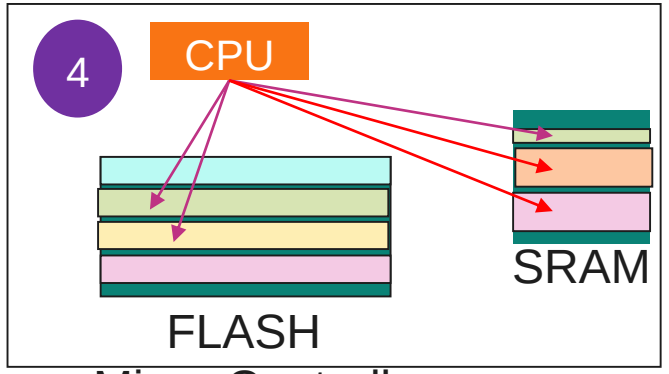
Not flashed.

These are needed only for debugging

At run time (After powering up)



At run time (After program loading)

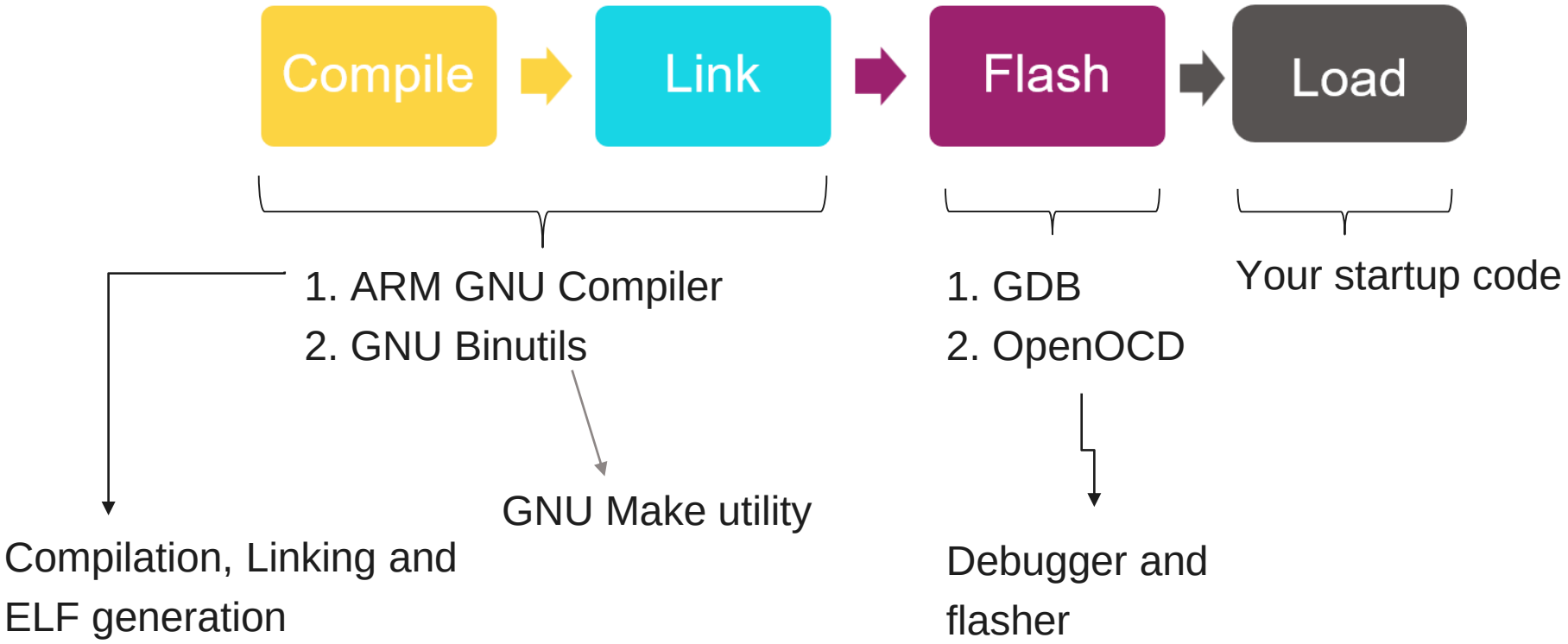


CPU fetches TEXT and RODATA from flash

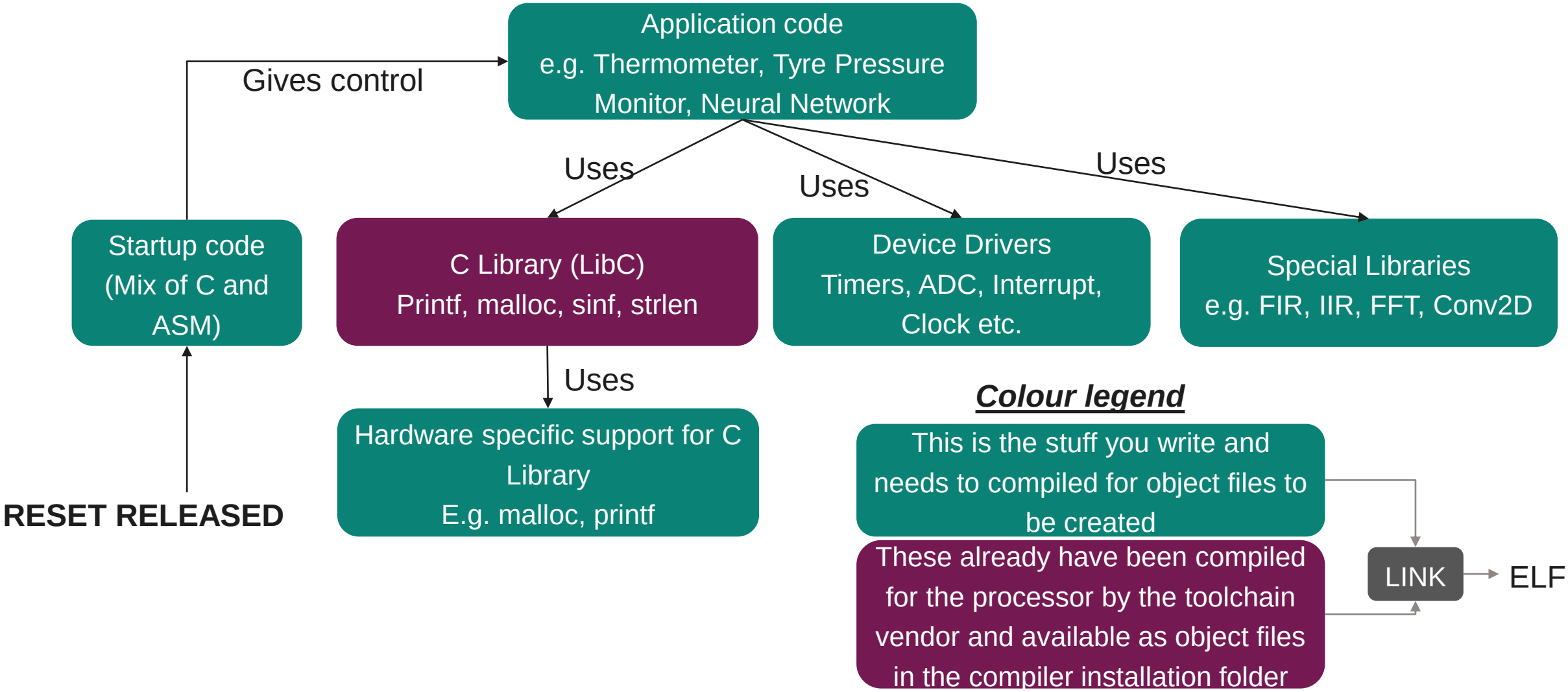
CPU accesses DATA and BSS in SRAM

CPU may fetch instructions of time critical TEXT segment which has been coped to SRAM.

Tools to install



A typical baremetal source code tree



How is a source file compiled?

- REMEMBER THIS
  - Regardless of your source file type (C or CPP or Assembly), you will use ONLY **arm-none-eabi-gcc** command
  - Even the linker will need to be invoked only with **arm-none-eabi-gcc** command

- TYPICAL USAGE
  - Compiling a C/C++ source file to generate a corresponding object file
    - **arm-none-eabi-gcc** <Compiler Options> -c Your_C_File.c –o Your_C_File.o

  - Linking multiple object files to create your application ELF
    - **arm-none-eabi-gcc** –T Your_Linker_Script.ld <Linker options> <List of object files> -o Your_Application_Name.elf

- COMPILE OPTIONS
  - -O2 -ffunction-sections -fdata-sections -Wall -std=gnu99 -mcpu=cortex-m0plus -mthumb -g -I$(INCPATH)

- LINK OPTIONS
  - -nostartfiles -Xlinker --gc-sections -specs=nano.specs -specs=nosys.specs -Wl,-Map,$(MAP_FILE)

Exercise – 1/2

- Create a special folder Esc-Practice-Assignments and in that folder:
- Exercise-1
  - Create a file “Add.c”
  - Now define a C function Add_2_Integers
  - This function accepts two unsigned integers as input and returns their sum
  - Compile this file and generate the corresponding object file

- Exercise-2
  - Build upon your previous work
  - In a new file “Sub.c”, define a C function Sub_2_Integers
  - This function accepts two unsigned integers as arguments, subtracts the smaller from the larger and returns the result
  - Compile the two files and generate two object files

- Your compiler options are: -mcpu=cortex=m0plus -mthumb

Exercise – 2/2

- Exercise-3
  - Now, create a third file App.c and code it as follows
  - Define an array of 5 unsigned integers called my_data and store in it the following values
    - 25, 37,1187,2,14
  - Create a function **my_math.** This function neither accepts any arguments nor returns any data
  - This function however calls the **Add_2_Integers** and **Sub_2_Integers** functions one after another
    - Remember that these are the same functions written as part of Exercises 1 and 2
  - The first two data elements of my_data are given as arguments to the **Add_2_Integers** function.
  - The subsequent two elements (i.e. the third and the fourth) of my_data are given as arguments to **Sub_2_Integers** function

  - **NOTE:** App.c has no way of knowing the definitions of Add_2_Integers and Sub_2_Integers functions. You must first declare these two functions in App.c using the **extern** keyword. This is something that you will have to do almost regularly as an embedded systems engineer
  - Compile App.c and generate App.o using the same compiler options listed on the previous page

  - Execute the following command: arm-none-eabi-objdump –h App.o
  - Understand the output produced by the objdump command

  - Now execute arm-none-eabi-objdump –S App.o
  - What do you see now?

Makefile - Motivation

- Build automation infrastructure to manage large projects
- With Makefiles, you can automate
  - Compilation of a large number of source code files, Linking of objects and generation of ELF
  - Other ancillary tasks such as flashing and many more (really!)
- You define dependencies and implicitly the order in which compilation and linking may be done
- You define the rules/commands for each task (rule for compilation, rule for linking, rule for sending SMS…)
- Makefile will start with the top level target that you have defined, understand dependencies among various files and execute the rules set by you!

Dependencies and rules

- ADC.o is an object file
- It is created from Adc.c
- Therefore Adc.o is dependent on Adc.c

- This dependency is expressed in the Makefile as:

Adc.o : Adc.c **DEPENDENCY**

- So a construct that you will regularly see is in a Makefile:

File.o : File.c

arm-none-eabi-gcc <Compile Options> –c File.c –o File.o **RULE**

Application.Elf : File.o Main.o

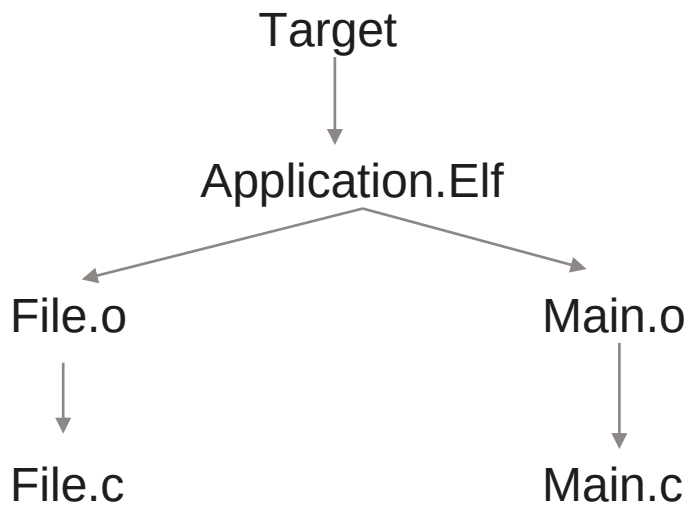
arm-none-eabi-gcc <Link Options> File.o Main.o –o Application.ELF

Target : Application.Elf

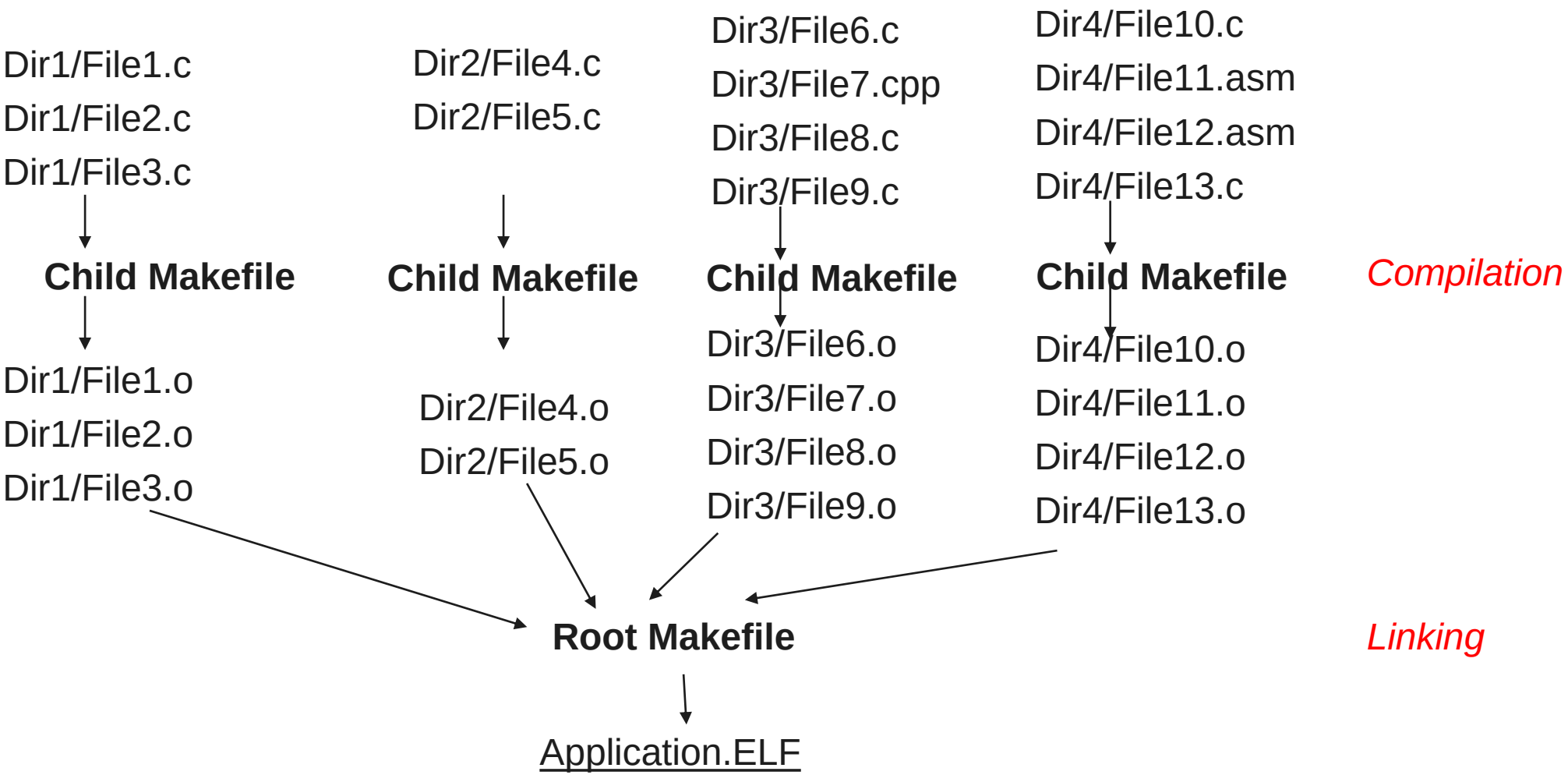
How to invoke a Makefile?

C:\windows_prompt>make Target

How does Makefile establish dependency?



Hierarchical Makefile




Infineon