# Compilation:

## Cross-compilation toolchains

- ✔ Toolchain or a cross-compilation toolchain is a collection of binaries which allows you to compile, assemble, link your applications.
- ✔ It also contains binaries to debug the application on the target
- ✔ Toolchain also comes with other binaries which help you to analyze the executables
  - dissect different sections of the executable
  - disassemble
  - extract symbol and size information
  - convert executable to other formats such as bin, ihex
  - Provides 'C' standard libraries

**Popular Tool-chains:**

1. GNU Tools(GCC) for ARM Embedded Processors (free and open-source)
2. armcc from ARM Ltd. (ships with KEIL , code restriction version, requires licensing)

**Throughout this course, we have been using,**
GNU's Compiler Collections(GCC) Toolchain

## Cross toolchain important binaries

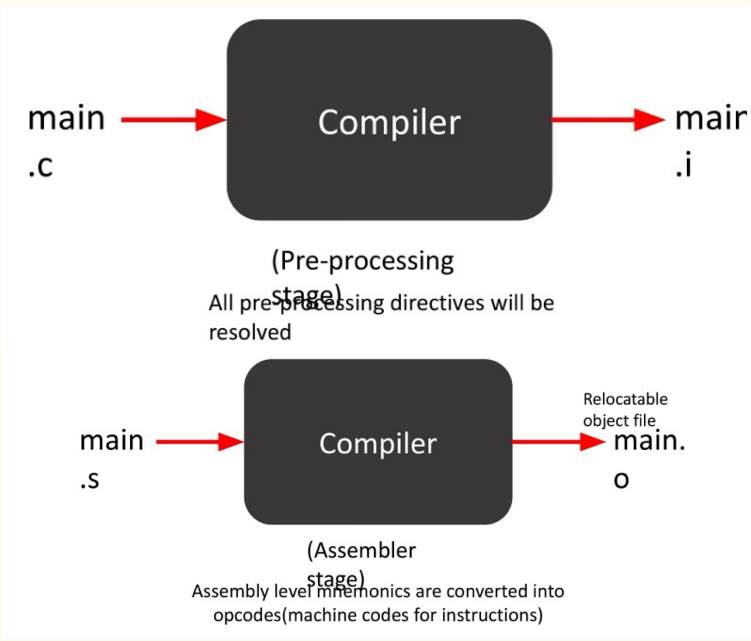
Compiler, linker, assembler
arm-none-eabi-gcc

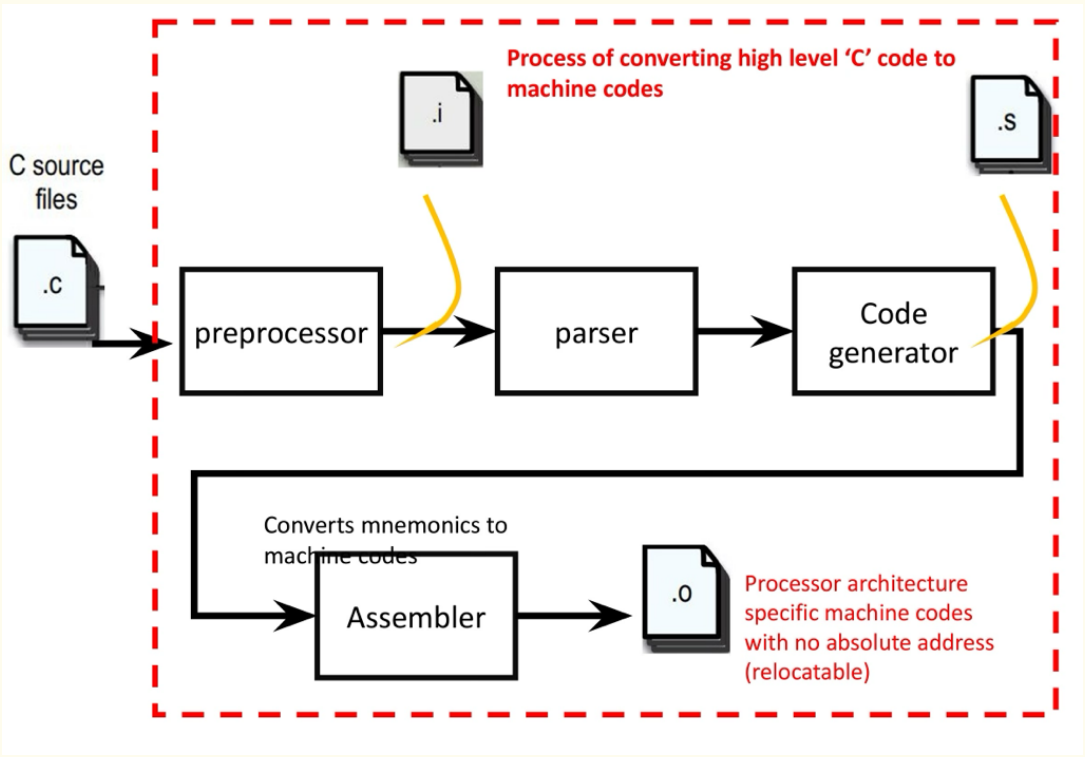
linker
arm-none-eabi-ld

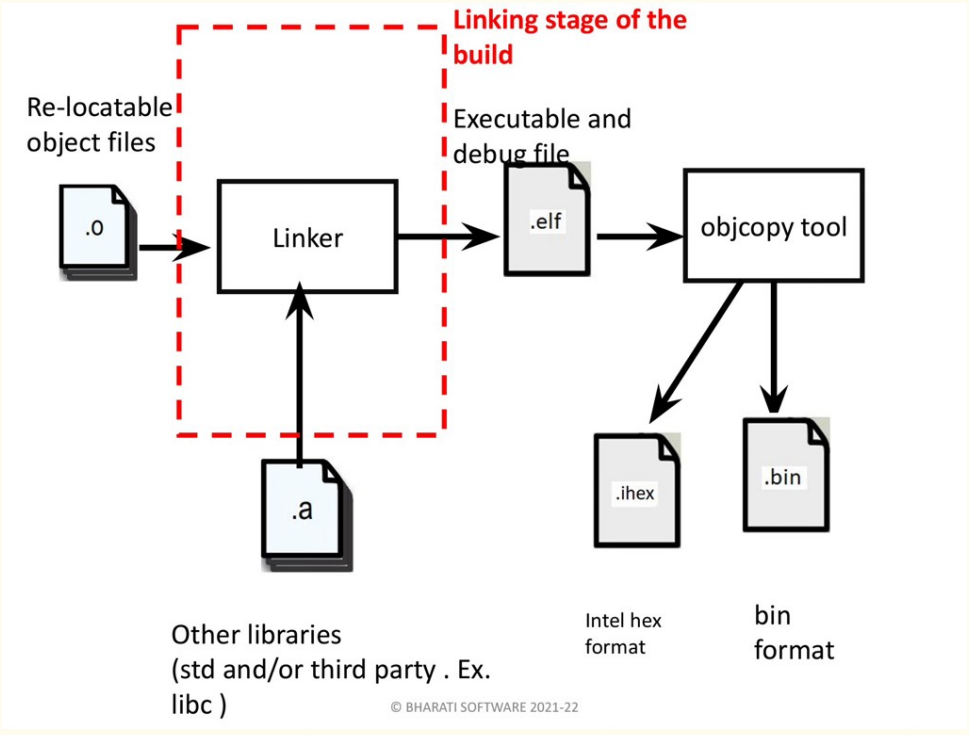
Assembler
arm-none-eabi-as

Elf file analyzer
arm-none-eabi-objdump
arm-none-eabi-readelf
arm-none-eabi-nm

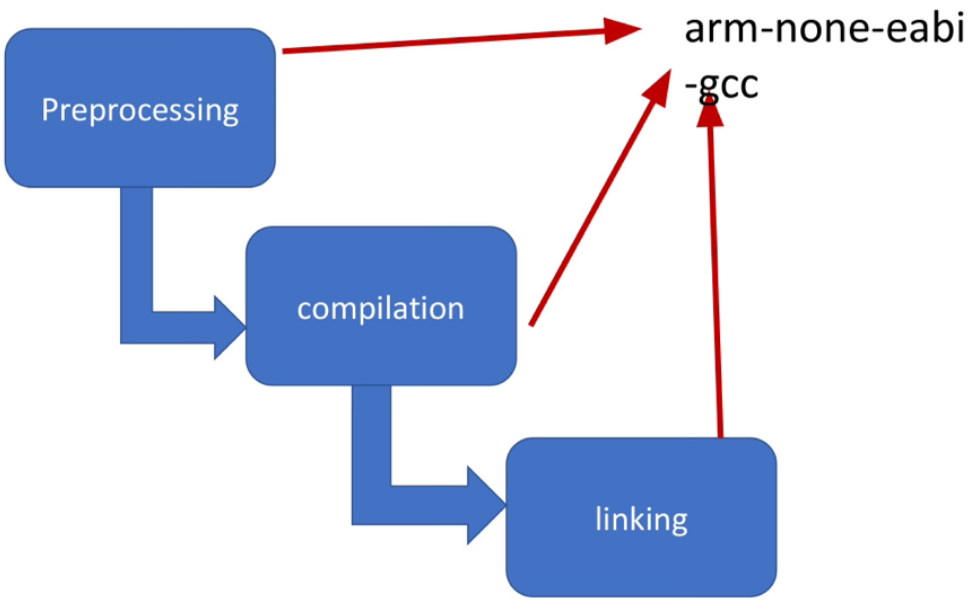
Format converter
arm-none-eabi-objcopy


main
.c
Compiler
mair
.i
(Pre-processing stage)
All pre-processing directives will be resolved
translating a source file into assembly language
main
.i
Compiler
main
.s
(Code generation stage)
Higher level language code statements will be converted into processor architectural level mnemonics
main
.s
Compiler
Relocatable object file
main.
o
(Assembler stage)
Assembly level mnemonics are converted into opcodes(machine codes for instructions)


Process of converting high level 'C' code to machine codes
.i
.s
C source files
.c
preprocessor
parser
Code generator
Converts mnemonics to machine codes
Assembler
.o
Processor architecture specific machine codes with no absolute address (relocatable)


Linking stage of the build
Re-locatable object files
Executable and debug file
.o
Linker
.elf
objcopy tool
.a
.ihex
.bin
Other libraries (std and/or third party . Ex. libc )
Intel hex format
bin format
© BHARATI SOFTWARE 2021-22

# Summary of build process




# Compiler options

`-c` Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix '`.c`', '`.i`', '`.s`', etc., with '`.o`'.
Unrecognized input files, not requiring compilation or assembly, are ignored.

`-S` Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.
By default, the assembler file name for a source file is made by replacing the suffix '`.c`', '`.i`', etc., with '`.s`'.
Input files that don't require compilation are ignored.

`-E` Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.
Input files that don't require preprocessing are ignored.

`-o` *file* Place output in file *file*. This applies to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.
If '`-o`' is not specified, the default is to put an executable file in '`a.out`', the object file for '*`source.suffix`*' in '*`source`*`.o`', its assembler file in '*`source`*`.s`', a precompiled header file in '*`source.suffix`*`.gch`', and all preprocessed C source on standard output.

# Compilation

- <compiler > <source file name .s or .c > -o <filename.o>

Important flags

-mcpu=cortex-m4

-mthumb

-c

`-mthumb`
`-marm`

Select between generating code that executes in ARM and Thumb states. The default for most configurations is to generate code that executes in ARM state, but the default can be changed by configuring GCC with the `--with-mode=`*state* configure option.

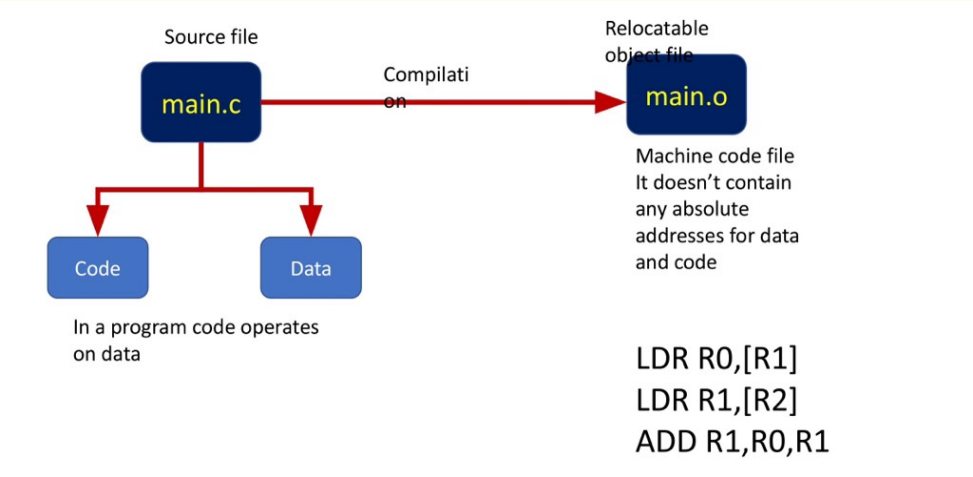
You can also override the ARM and Thumb mode for each function by using the `target("thumb")` and `target("arm")` function attributes (see ARM Function Attributes) or pragmas (see Function Specific Option Pragmas).

`-mcpu=`*name*`[+extension…]`

This specifies the name of the target ARM processor. GCC uses this name to derive the name of the target ARM architecture (as if specified by `-march`) and the ARM processor type for which to tune for performance (as if specified by `-mtune`). Where this option is used in conjunction with `-march` or `-mtune`, those options take precedence over the appropriate part of this option.

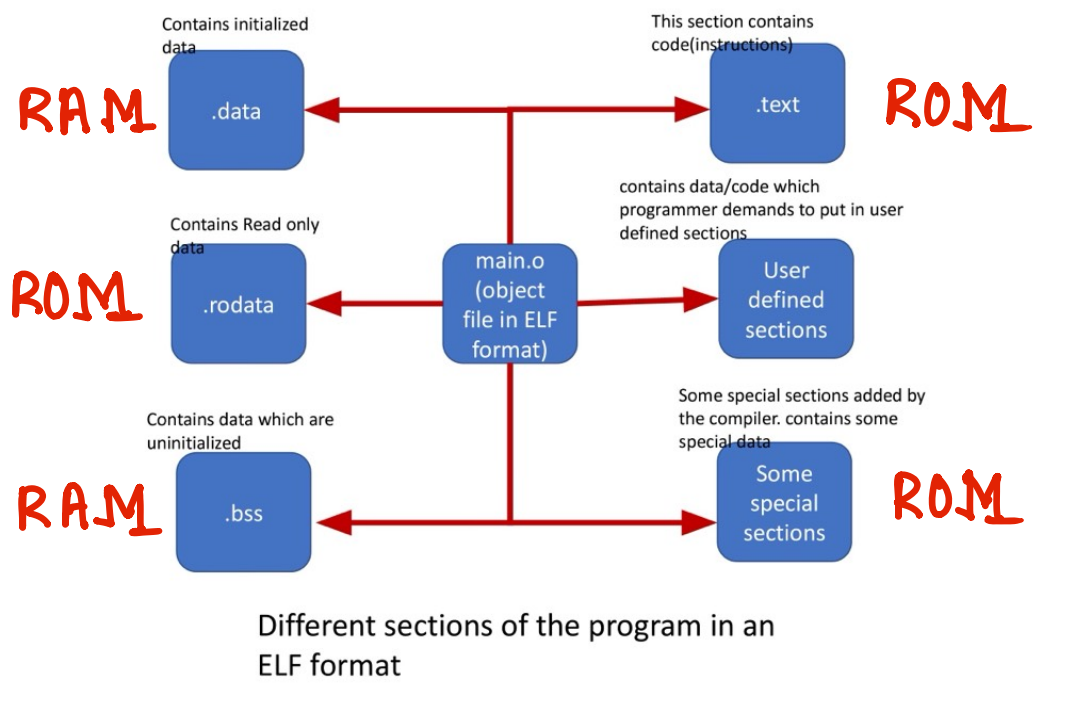
`-mtune=`*name*

This option specifies the name of the target ARM processor for which GCC should tune the performance of the code. For some ARM implementations better performance can be obtained by using this option. Permissible names are: ‘`arm7tdmi`’, ‘`arm7tdmi-s`’, ‘`arm710t`’, ‘`arm720t`’, ‘`arm740t`’, ‘`strongarm`’, ‘`strongarm110`’, ‘`strongarm1100`’, 0‘`strongarm1110`’, ‘`arm8`’, ‘`arm810`’, ‘`arm9`’, ‘`arm9e`’, ‘`arm920`’, ‘`arm920t`’, ‘`arm922t`’, ‘`arm946e-s`’, ‘`arm966e-s`’, ‘`arm968e-s`’, ‘`arm926ej-s`’, ‘`arm940t`’, ‘`arm9tdmi`’, ‘`arm10tdmi`’, ‘`arm1020t`’, ‘`arm1026ej-s`’, ‘`arm10e`’, ‘`arm1020e`’, ‘`arm1022e`’, ‘`arm1136j-s`’, ‘`arm1136jf-s`’, ‘`mpcore`’, ‘`mpcorenovfp`’, ‘`arm1156t2-s`’, ‘`arm1156t2f-s`’, ‘`arm1176jz-s`’, ‘`arm1176jzf-s`’, ‘`generic-armv7-a`’, ‘`cortex-a5`’, ‘`cortex-a7`’, ‘`cortex-a8`’, ‘`cortex-a9`’, ‘`cortex-a12`’, ‘`cortex-a15`’, ‘`cortex-a17`’, ‘`cortex-a32`’, ‘`cortex-a35`’, ‘`cortex-a53`’, ‘`cortex-a55`’, ‘`cortex-a57`’, ‘`cortex-a72`’, ‘`cortex-a73`’, ‘`cortex-a75`’, ‘`cortex-a76`’, ‘`cortex-a76ae`’, ‘`cortex-a77`’, ‘`ares`’, ‘`cortex-r4`’, ‘`cortex-r4f`’, ‘`cortex-r5`’, ‘`cortex-r7`’, ‘`cortex-r8`’, ‘`cortex-r52`’, ‘`cortex-m0`’, ‘`cortex-m0plus`’, ‘`cortex-m1`’, ‘`cortex-m3`’, ‘`cortex-m4`’, ‘`cortex-m7`’, ‘`cortex-m23`’, ‘`cortex-m33`’, ‘`cortex-m35p`’, ‘`cortex-m1.small-multiply`’, ‘`cortex-m0.small-multiply`’, ‘`cortex-m0plus.small-multiply`’, ‘`exynos-m1`’, ‘`marvell-pj4`’, ‘`neoverse-n1`’, ‘`xscale`’, ‘`iwmmxt`’, ‘`iwmmxt2`’, ‘`ep9312`’, ‘`fa526`’, ‘`fa626`’, ‘`fa606te`’, ‘`fa626te`’, ‘`fmp626`’, ‘`fa726te`’, ‘`xgene1`’.



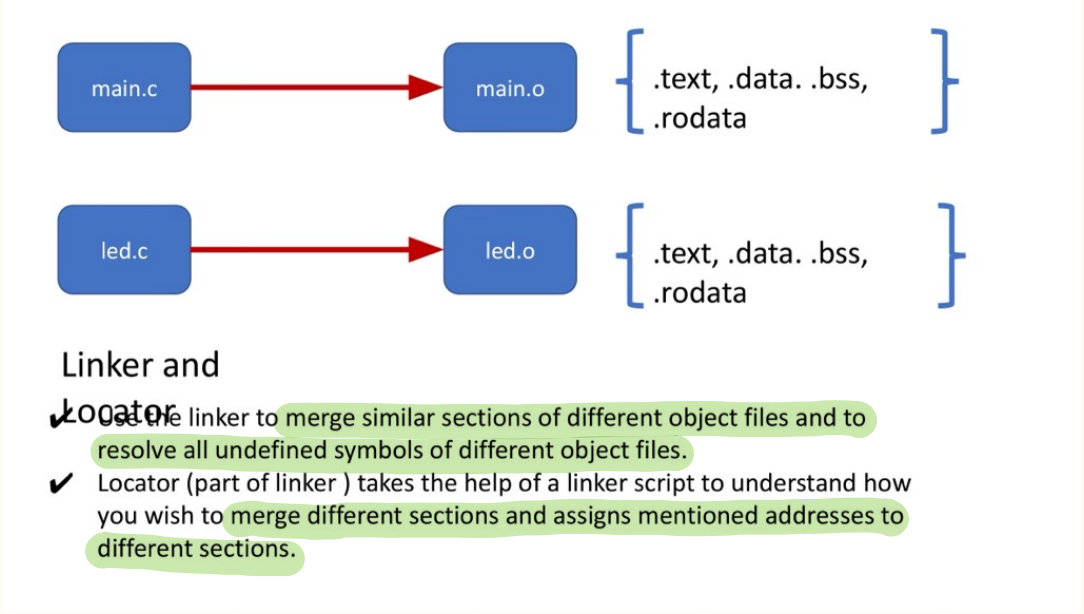

→ Instructions (CODES) are stored in FLASH.

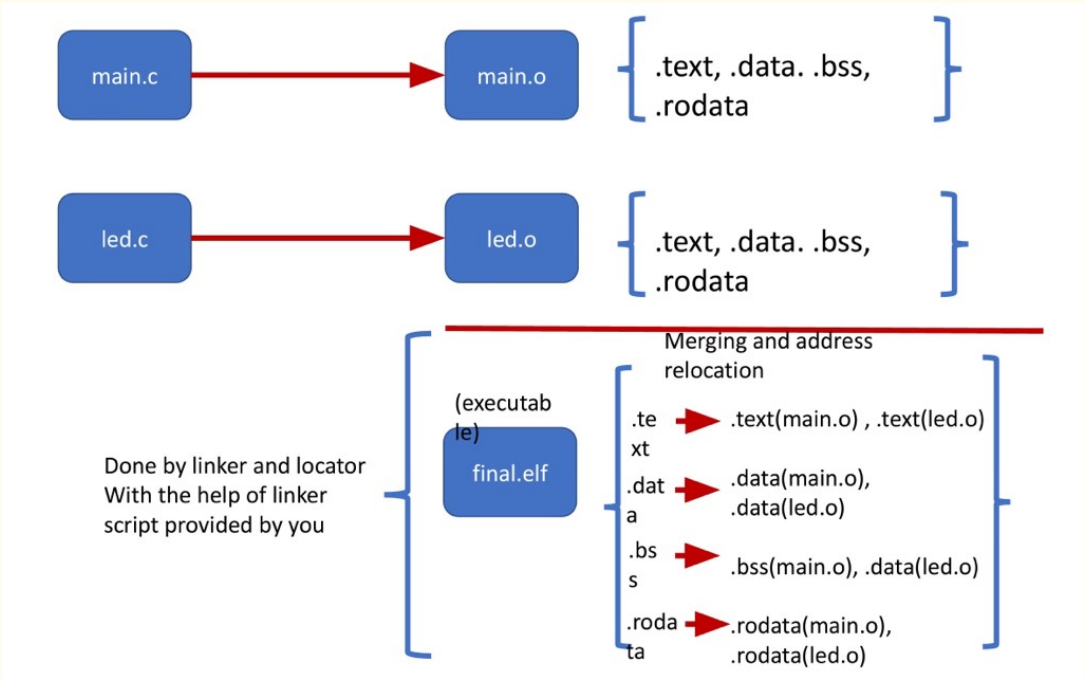
→ DATA (variables) are stored in RAM. as vars can change anytime.

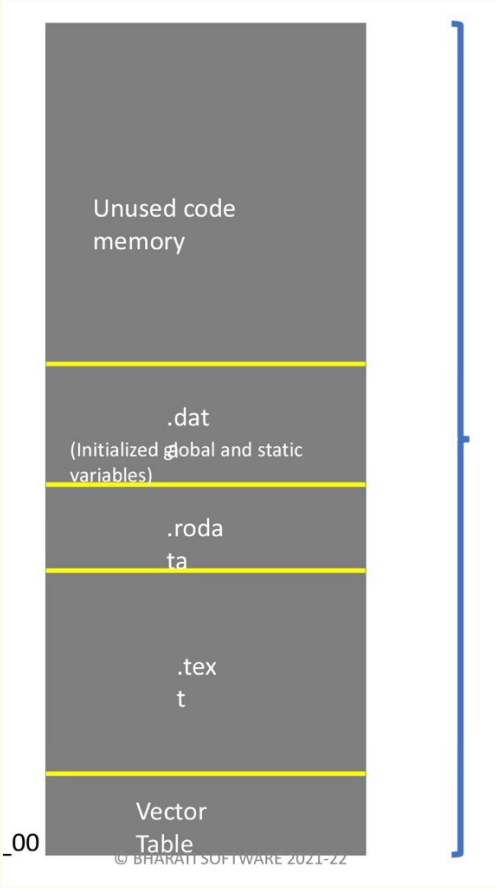


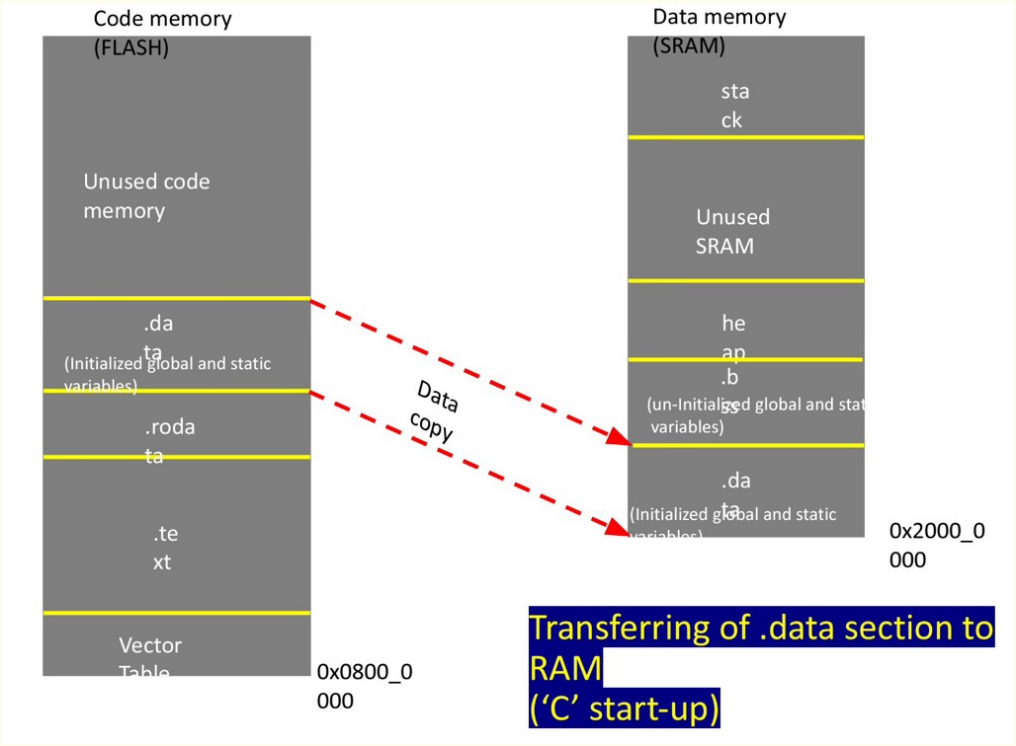

Different sections of the program in an ELF format

# Linker and Locator:





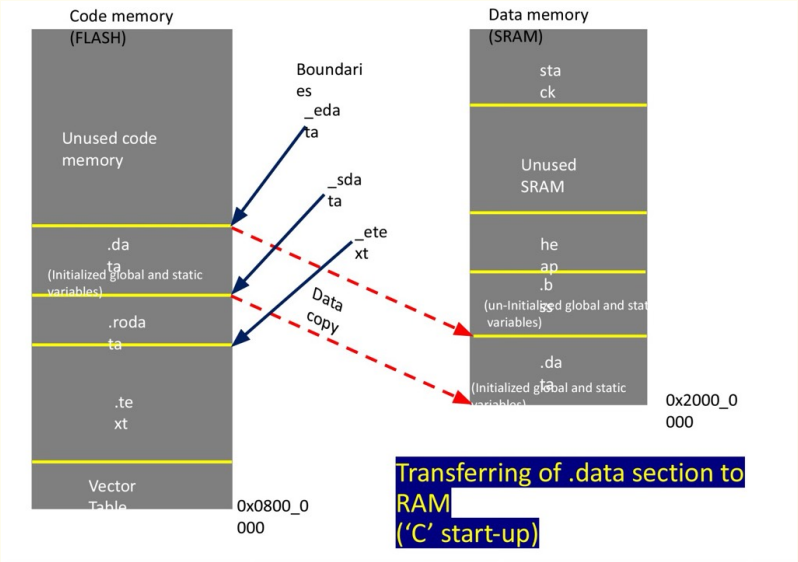




→ Flash memory ↓ for execution we need to copy some parts from FLASH to RAM

→ Initial addresses of code memory should be occupied by the vector table.

→ Transfering .data section from FLASH to RAM is done by c_startup code.

→ In order to copy .data from FLASH to RAM we need to know some boundaries. ⇒ _etext, _sdata, _edata




→ They hold boundary addresses info.

→ Initial address and size.

→ .bss section is directly placed into SRAM because uninitialized data that BSS holds may acquire unnecessary space in FLASH (ROM)

| Variable (Data) | LOAD time | RUN time | Section | Note |
|---|---|---|---|---|
| Global initialized | FLASH | RAM | .data | Should be copied from flash to ram by startup code |
| Global uninitialized | ---- | RAM | .bss | startup code reserves space for this data in RAM and initializes to zero |
| Global static initialized | FLASH | RAM | .data | Should be copied from flash to ram by startup code |
| Global static uninitialized | ---- | RAM | .bss | startup code reserves space for this data in RAM and initializes to zero |
| Local initialized | ---- | STACK(RAM) | ---- | Consumed at run time |
| Local uninitialized | ---- | STACK(RAM) | ---- | Consumed at run time |
| Local static initialized | FLASH | RAM | .data | Should be copied from flash to ram by startup code |
| Local static uninitialized | ---- | RAM | .bss | startup code reserves space for this data in RAM and initializes to zero |
| All global const | FLASH | ---- | .rodata | |

- LMA : Load Memory Address. → FLASH
- VMA : Virtual Memory Address. → RAM

## .bss( block started by symbol) and .data section

- All the uninitialized global variables and uninitialized static variables are stored in the **.bss** section.
- Since those variables do not have any initial values, they are not required to be stored in the **.data** section since the **.data** section consumes FLASH space. Imagine what would happen if there is a large global uninitialized array in the program, and if that is kept in the **.data** section, it would unnecessarily consume flash space yet carries no useful information at all.
- **.bss** section doesn't consume any FLASH space unlike **.data** section
- You must reserve RAM space for the **.bss** section by knowing its size and initialize those memory space to zero. This is typically done in startup code
- Linker helps you to determine the final size of the **.bss** section. So, obtain the size information from a linker script symbols.

```c
#include<stdint.h>

uint32_t d_count = 300000;

int g_un_data1;
int g_un_data2 = 0;
int g_i_data = 0x55;//.data
static int g_un_s_data;
static int g_i_s_data = 0x44; //.data


int main(void){

    printf("data = %d,%d,%d,%d,%d\n",g_un_data1,g_un_data2,g_i_data,g_un_s_data,g_i_s_data);

    int l_un_data;
    int l_i_data = 0x98;
    static int l_un_s_data;
    static int l_i_s_data = 0x24; //.data

    printf("sum = %d\n",l_un_data+l_i_data+l_un_s_data+l_i_s_data);

}
```

# startup File:

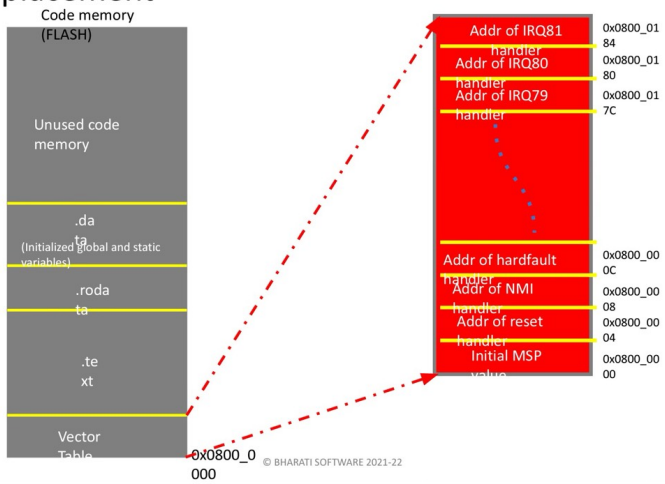
## Start-up file

1. Create a vector table for your microcontroller . Vector tables are MCU specific
2. Write a start-up code which initializes .data and .bss section in SRAM
3. Call main()

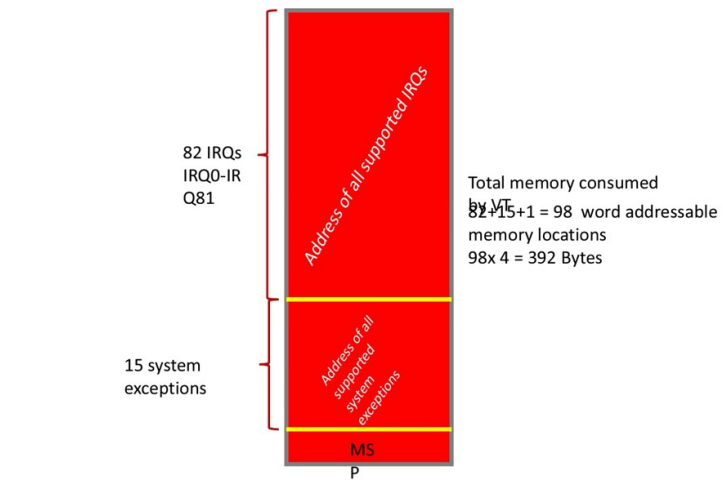
## Importance of start-up file

- The startup file is responsible for setting up the right environment for the main user code to run
- Code written in startup file runs before main(). So, you can say startup file calls main()
- Some part of the startup code file is the target (Processor) dependent
- Startup code takes care of vector table placement in code memory as required by the ARM cortex Mx processor
- Startup code may also take care of stack reinitialization
- Startup code is responsible of .data, .bss section initialization in main memory

Vector table placement




Exceptions and interrupts in STM32F4VGTx MCU




## Creating a Vector Table

- Create an array to hold MSP and handlers addresses.

```
uint32_t vectors[] = {store MSP and address of various handlers here};
```

- Instruct the compiler not to include the above array in .data section but in a different user defined section

stm_startup.c

```c
#include<stdint.h>

#define SRAM_START 0x20000000U
#define SRAM_SIZE (128*1024)
#define SRAM_END ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

void Reset_Handler(void);

void NMI_Handler        (void) __attribute__ ((weak, alias("Default_Handler")));

/*

Now we have to inform the compiler to not put this  vector table to .data section
and include it vector table section. we have use __attribute__ ((section("SECTION_NAME")))

*/

uint32_t vectors[] __attribute__((section(".isr_vector"))) = {
    STACK_START,
    (uint32_t)&Reset_Handler // funtion pointer so need to be typecasted to uint32
};

/*

or

uint32_t vectors[] = {
    STACK_START,
    (uint32_t)&Reset_Handler // funtion pointer so need to be typecasted to uint32
};__attribute__((section(".isr_vector")))

*/


void Reset_Handler(void)
{

    /*
    1. copy .data section to SRAM

    2. Init .bss section to zero in SRAM

    3. call init functions of std library

    4. call main()

    These 4 things to be implemented in Reset Handler and for copying and init it is
    necessary to know the boundaries of each section = _etext, _sdata, _edata.
    For that we need linker script
    */


    // copy .data section to SRAM

    // Init .bss section to zero in SRAM

    // call init functions of std library

    // call main()

}

void Default_Handler(void)
{
    while(1);
}
```

## variable attribute

`__attribute__((section("name")))`

- The section attribute specifies that a variable must be placed in a special section mentioned by the programmer
- Normally, the compiler places the data it generates in sections like .data and .bss. However, you might require additional data sections, or you might want a variable to appear in a special section, for example, to map to special hardware.

→ we do not want to put startup code's IRQ Handlers in .data section rather put it in isr_vector section

## Function attribute : weak and alias

Weak :

Lets programmer to override already defined weak function(dummy) with the same function name

→ required for vector table creation.

Alias :

Lets programmer to give alias name for a function

→ function definition is in main.c

# Linker Script:

## Linker scripts

- Linker script is a text file which explains how different sections of the object files should be merged to create an output file
- Linker and locator combination assigns unique absolute addresses to different sections of the output file by referring to address information mentioned in the linker script
- Linker script also includes the code and data memory address and size information.
- Linker scripts are written using the GNU linker command language.
- GNU linker script has the file extension of .ld
- You must supply linker script at the linking phase to the linker using –T option.

## Linker scripts commands

- ENTRY
- MEMORY
- SECTIONS
- KEEP
- ALIGN
- AT>

## ENTRY command

- This command is used to set the "***Entry point address***" information in the header of final elf file generated
- In our case, "Reset_Handler" is the entry point into the application. The first piece of code that executes right after the processor reset.
- The debugger uses this information to locate the first function to execute.
- Not a mandatory command to use, but required when you debug the elf file using the debugger (GDB)
- Syntax : Entry(_symbol_name_)

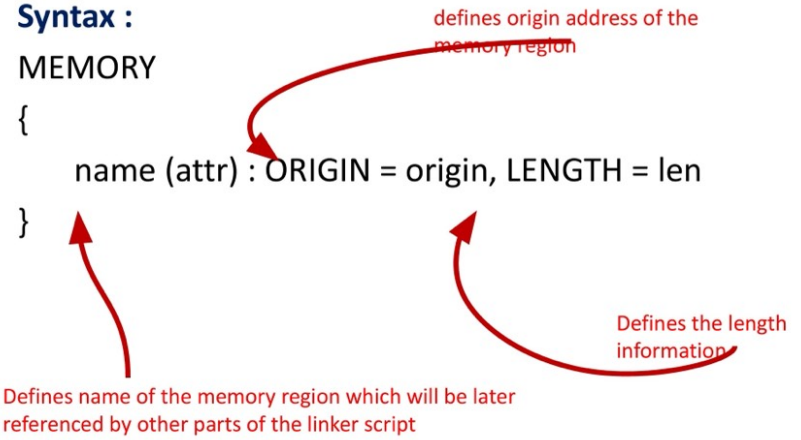
Entry(Reset_Handler)

# Memory command

- This command allows you to describe the different memories present in the target and their start address and size information
- The linker uses information mentioned in this command to assign addresses to merged sections
- The information is given under this command also helps the linker to calculate total code and data memory consumed so far and throw an error message if data, code, heap or stack areas cannot fit into available size
- By using memory command, you can fine-tune various memories available in your target and allow different sections to occupy different memory areas
- Typically one linker script has one memory command

→ In one linker script there should be one MEMORY command.

**Syntax :**




| Attribute letter | Meaning |
|---|---|
| R | Read-only sections |
| W | Read and write sections |
| X | Sections containing executable code. |
| A | Allocated sections |
| I | Initialized sections. |
| L | Same as 'I' |
| ! | Invert the sense of any of the following attributes. |

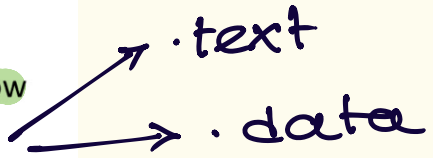
| Microcontroller | FLASH Size (in KB) | SRAM1 size (in KB) | SRAM2 size(in KB) |
|---|---|---|---|
| STM32F4VGT6 | 1024 | 112 | 16 |

stm_ld.ld

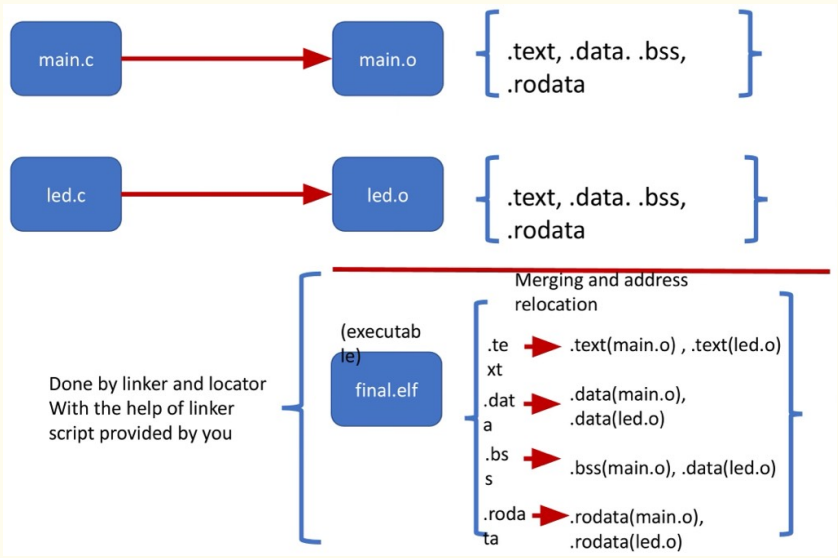
```
ENTRY(Reset_Handler)

MEMORY
{
    FLASH(rx) : ORIGIN = 0x80000000, LENGTH = 1024K
    SRAM(rwx) : ORIGIN = 0x20000000, LENGTH = 128K
}
```

# Sections command

- SECTIONS command is used to create different output sections in the final elf executable generated.
- Important command by which you can instruct the linker how to merge the input sections to yield an output section
- This command also controls the order in which different output sections appear in the elf file generated.
- By using this command, you also mention the placement of a section in a memory region. For example, you instruct the linker to place the .text section in the FLASH memory region, which is described by the MEMORY command.



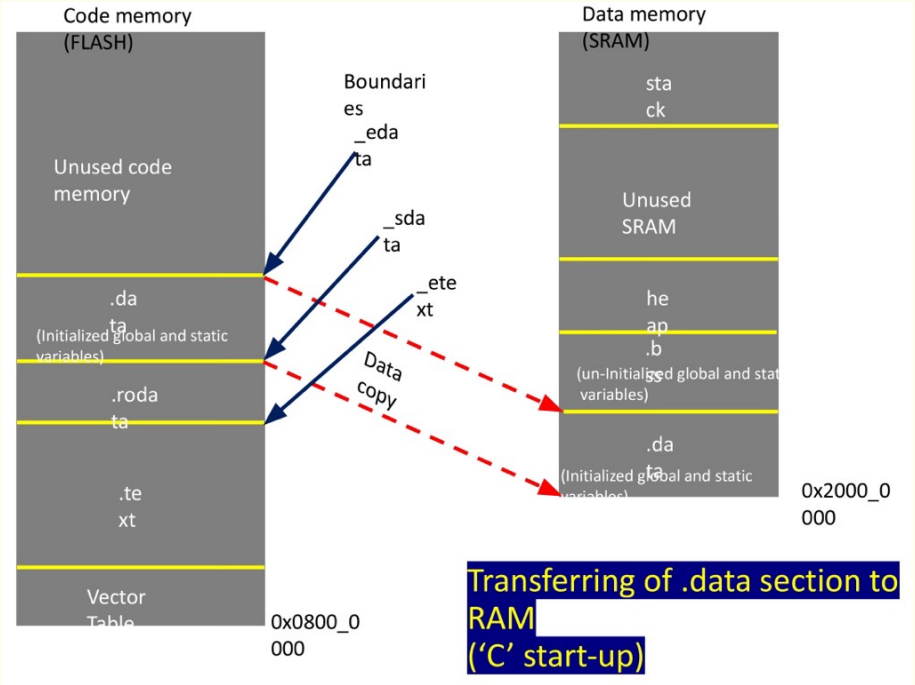

```
/* Sections */
SECTIONS
{
    /* This section should include .text section of all input files */
     .text :
     {
          //merge all .isr_vector section of all input files
          //merge all .text section of all input files
          //merge all .rodata section of all input files

     } >(vma) AT>(lma)

    /* This section should include .data section of all input files */
     .data :
     {

       //here merge all .data section of all input files

     } >(vma) AT>(lma)

}
```




stm_ld.ld

```
ENTRY(Reset_Handler)

MEMORY
{
    FLASH(rx) : ORIGIN = 0x80000000, LENGTH = 1024K
    SRAM(rwx) : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS
{
    .text :
    {
        *(.isr_vector)
        *(.text)
        *(.rodata)
    }> FLASH

    .data :
    {
        *(.data)
    }> SRAM AT> FLASH

    .bss :
    {
        *(.bss)
    }> SRAM
}
```

# Location Counter:

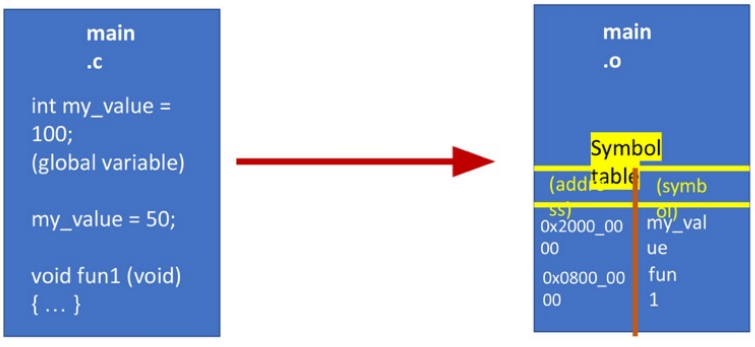



## Location counter (.)

- This is a special linker symbol denoted by a dot ' . '
- This symbol is called "location counter" since linker automatically updates this symbol with location(address) information
- You can use this symbol inside the linker script to track and define boundaries of various sections
- You can also set location counter to any specific value while writing linker script
- Location counter should appear only inside the SECTIONS command
- The location counter is incremented by the size of the output section

## Linker script symbol

- A symbol is the name of an address
- A symbol declaration is not equivalent to a variable declaration what you do in your 'C' application




→ In main.o compiler maintains a symbol table.

→ every symbol name assigned with an address.

```
1 ENTRY(Reset_Handler)
2
3 MEMORY
4 {
5   FLASH(rx):ORIGIN=0x08000000,LENGTH=1024K
6   SRAM(rwx):ORIGIN=0x20000000,LENGTH=128K
7 }
8
9 __max_heap_size = 0x400;  /* A symbol declaration . Not a variable !!! */
10 __max_stack_size = 0x200; /* A symbol declaration . Not a variable !!! */
11
12 SECTIONS
13 {
14   .text :
15   {
16     *(.isr_vector)
17     *(.text)
18     *(.rodata)
19     end_of_text = .; /* Store the updated location counter value in to a symbol 'end_of_test' */
20   }> FLASH
21
22   .data
23   {
24     start_of_data = 0x20000000; /* Assign a value to a symbol 'start_of_data' */
25     *(.data)
26   }> SRAM AT> FLASH
27
```

location-counter

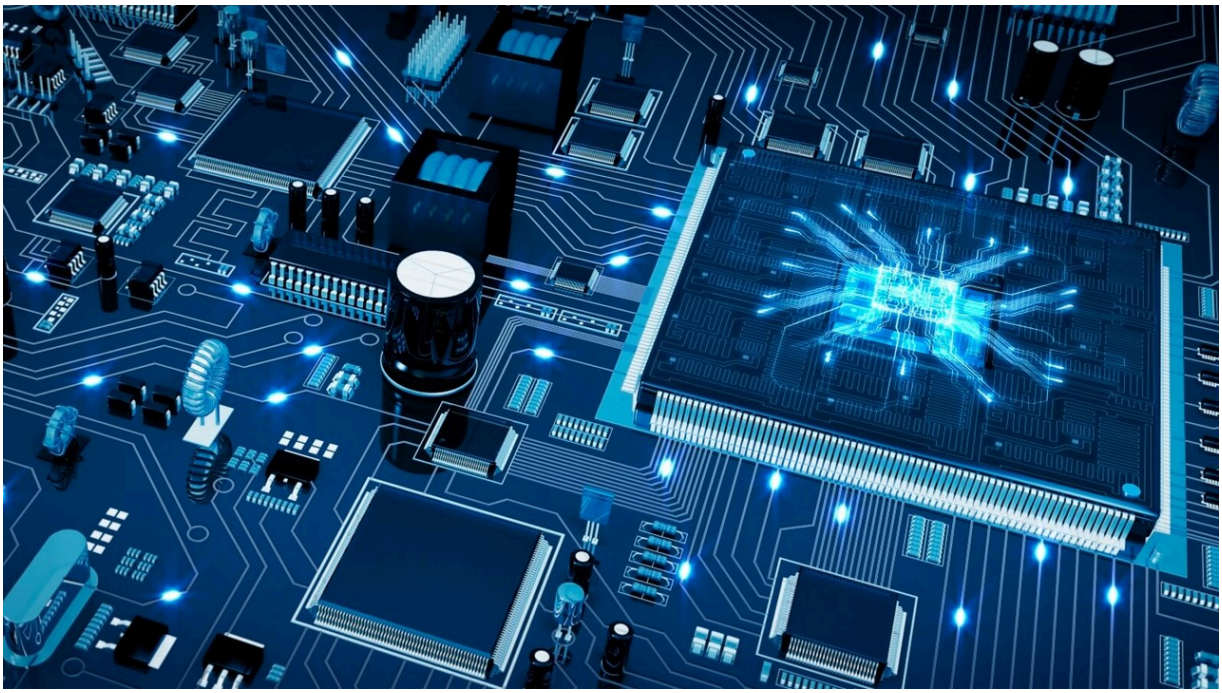
boundary addresses

→ Initially address = 0x80000000 (VMA)

→ then size of .text will be added with initial address.

→ and we will get the address at end_of_text

- You can create symbols inside the linker script and assign any values
- Symbols created inside can be accessed by a ‘C’ program using ‘extern’ keyword
- Symbols can be created anywhere in the linker script but the special symbol ‘.’(location counter) can only be used inside the SECTIONS command



# Bare Metal Implementation

03.06.2024

—

Sushovan Saha

## Makefile

```
#variables

CC = $(TOOLPATH_COMPILER)/bin/arm-eabi-gcc.exe
MACH = cortex-m4
STD = gnu11
CFLAGS = -c -mcpu=$(MACH) -mthumb -mfloat-abi=soft -std=$(STD) -O0

# Now add command and flags for linker
# -T is used for linker
# -nostdlib is used for not include statndard libary (printf,scanf won't
work)
# -mfloat-abi=soft : used for floating point calculation

LDFLAGS = -mcpu=$(MACH) -mthumb -mfloat-abi=soft --specs=nosys.specs -T
stm32_ls.ld -Wl,-Map=final.map

all: main.o led.o startup.o syscalls.o final.elf

# output file : source files needed for op file
#   then write the instruction or recipe

# $@ -> output file. Ex - main.o
# $^ -> source files. Ex - main.c ..

main.o : main.c
    $(CC) $(CFLAGS) -o $@ $^

led.o : led.c
    $(CC) $(CFLAGS) -o $@ $^

startup.o : stm32_startup.c
    $(CC) $(CFLAGS) -o $@ $^

syscalls.o : syscalls.c
    $(CC) $(CFLAGS) -o $@ $^

final.elf : main.o led.o startup.o syscalls.o
    $(CC) $(LDFLAGS) -o $@ $^
```

```
clean:
    rm -rf *.o *.elf

.PHONY: clean all
```

## Linker Descriptive File

We can instruct the Linker to generate a ***.map*** file. By analyzing that .map file we can see various resource allocation and placements in the memory. For that we have to add linker argument :  and we have to explicitly mention that this is a linker command using : ***-Wl,***

### 1. Memory Map File

`-Wl,-Map=final.map`

`main.c`

```
// const var will be included in .rodata
const uint32_t const_v1 = 100;
const uint32_t const_v2 = 200;
```

final.map

```
*(.rodata)
 *fill*         0x08000066        0x2
 .rodata        0x08000068        0x8 main.o
                0x08000068                const_v1
                0x0800006c                const_v2
                0x08000070                _etext = .
```

So to make alignment, there is 2B of padding. But if we add one more const var of uint8 then this alignment will change.

```
// const var will be included in .rodata
const uint32_t const_v1 = 100;
const uint32_t const_v2 = 200;
const uint8_t const_v3 = 50;
```

```
 *(.rodata)
 *fill*         0x08000066        0x2
 .rodata        0x08000068        0x9 main.o
                0x08000068                const_v1
                0x0800006c                const_v2
                0x08000070                const_v3
                0x08000071                _etext = .
```

_etext = `0x08000071.` Start of the .data section is not aligned in the code memory. This is bad, we **should always have aligned addresses for all the sections**. And as it is the **end of the section, linker will not fix it using *fill***. We have to do this manually by using **align** command. So we will add the align command in the .ld file. And we will do **word boundary alignment** here. **The value must be power of 2.**

```
SECTIONS
{
    .text :
    {
        *(.isr_vector)
        *(.text)
        *(.rodata)

        /*
        For any assignment ; will be used.
        new address of the location counter will be updated with word
boundary
        using ALIGN(4) command and then it will be stored to _etext
        */
        . = ALIGN(4);
        /*
        Location Counter only tracks VMA not LMA.
        It will contain updated addresses for .text section
        */
        _etext = .;
    }> FLASH
```

Updated final.map file :

```
*(.rodata)
 *fill*         0x08000066        0x2
 .rodata        0x08000068        0x9 main.o
                0x08000068                const_v1
                0x0800006c                const_v2
                0x08000070                const_v3
                0x08000074                . = ALIGN (0x4)
 *fill*         0x08000071        0x3
                0x08000074                _etext = .
```

As we can see .fill is added for padding and proper alignment. Same thing will be done for .data section.

main.c

```
//init var will be included in .data
uint8_t x = 7;
```

final.map

```
.data           0x20000000        0x1 load address 0x08000074
                0x20000000                _sdata = .
 *(.data)
 .data          0x20000000        0x1 main.o
                0x20000000                x
 .data          0x20000001        0x0 led.o
 .data          0x20000001        0x0 startup.o
                0x20000001                _edata = .
```

stm32_ls.ld

```
.data :
    {
        _sdata = .;
        *(.data)
        . = ALIGN(4);
        _edata = .;
    }> SRAM AT> FLASH  /*>VMA AT> LMA*/
```

final.map

```
.data           0x20000000        0x4 load address 0x08000074
```

```
                0x20000000                _sdata = .
 *(.data)
 .data          0x20000000        0x1 main.o
                0x20000000                x
 .data          0x20000001        0x0 led.o
 .data          0x20000001        0x0 startup.o
                0x20000004                . = ALIGN (0x4)
 *fill*         0x20000001        0x3
                0x20000004                _edata = .
```

- **Before ending any section, always add . = ALIGN(4)**

```
// if global var initialized with 0 or NULL then it goes to .bss not
.data
//uninit var will be included in .bss
int a = 0;

typedef struct
{
    uint8_t rollno;
    uint8_t age;
    uint8_t class;
}Student;

Student studarr[8];
```

```
.bss            0x20000004       0x1c load address 0x08000078
                0x20000004                _sbss = .
 *(.bss)
 .bss           0x20000004        0x4 main.o
                0x20000004                a
 .bss           0x20000008        0x0 led.o
 .bss           0x20000008        0x0 startup.o
                0x20000008                _ebss = .
 COMMON         0x20000008       0x18 main.o
                0x20000008                studarr
```

There is one more section COMMON which is created by the compiler and here contribution comes from main.o. But we want the section to be in the .bss section. For that we have to slightly modify our linker script. Explicitly add COMMON to .bss in .ld file stm32_ls.ld

```
.bss :
    {
        _sbss = .;
        *(.bss)
        *(.COMMON)
        . = ALIGN(4);
        _ebss = .;
    }> SRAM
```

final.map

```
.bss            0x20000004       0x1c load address 0x08000078
                0x20000004                _sbss = .
 *(.bss)
 .bss           0x20000004        0x4 main.o
                0x20000004                a
 .bss           0x20000008        0x0 led.o
 .bss           0x20000008        0x0 startup.o
 *(.COMMON)
                0x20000008                . = ALIGN (0x4)
                0x20000008                _ebss = .
 COMMON         0x20000008       0x18 main.o
                0x20000008                studarr
```

## Implement Reset_Handler in Startup Code

```
1. copy .data section to SRAM
2. Init .bss section to zero in SRAM
3. call init functions of std library
4. call main()
```

These 4 things to be implemented in **Reset_Handler** and for copying and init it is necessary to know the boundaries of each section = **_etext, _sdata, _edata**. For that we need linker script.

Now all the boundaries are known and we can implement **Reset_Handler**

## 1. Copy .data Section to SRAM

We access the boundaries in the ***stm_startup.c*** using the **extern** keyword. To see all the symbols in the applications, we can run the command : `arm-eabi-nm final.elf`

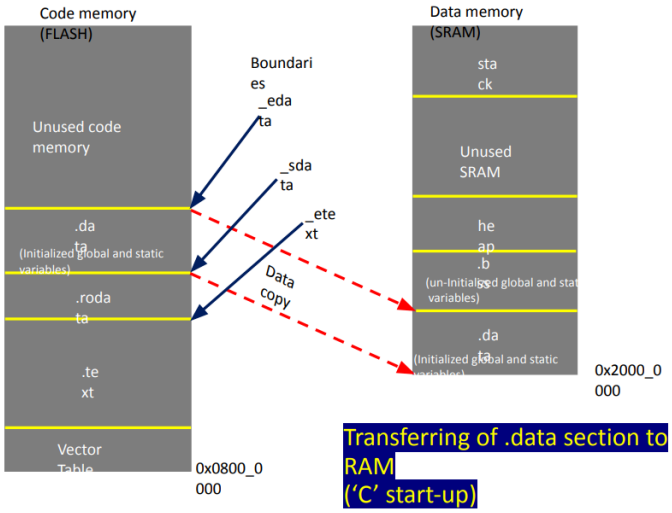
Output : **Symbol Table**

```
20000008 B _ebss
20000004 D _edata
08000074 T _etext
20000004 B _sbss
20000000 D _sdata
20000004 B a
08000068 T const_v1
0800006c T const_v2
08000070 T const_v3
08000060 T Default_Handler
08000048 T func
0800002c T main
08000060 W NMI_Handler
08000054 T Reset_Handler
20000008 B studarr
08000024 T vectors
20000000 D x
```

1. First calculate the size(no. Of bytes) of the section to be transferred. Here we use &_edata as we want its address not value inside it

```
// size of the section
    uint32_t size = (&_edata - &_sdata);
```

2. Now we copy data from FLASH to SRAM. for that we'll use 2 pointers : src_pointer and dst_pointer




```
// src and dst pointer for copying
    uint8_t *pDst = (uint8_t*)&_sdata; // SRAM
    uint8_t *pSrc = (uint8_t*)&_etext; // FLASH
```

3. Start copying from FLASH to SRAM

```
// start copying from FLASH to SRAM
    for(uint32_t i = 0; i<size; i++)
    {
        *pDst++ = *pSrc++;
    }
```

## 2. Init .bss section to zero in SRAM

```
extern uint32_t _sbss;
extern uint32_t _ebss;


/*2. Init .bss section to zero in SRAM*/
    size = &_ebss - &_sbss;
    pDst = (uint8_t*)&_sbss;

    for(uint32_t i = 0;i<size;i++)
    {
        *pDst++ = 0;
    }
```

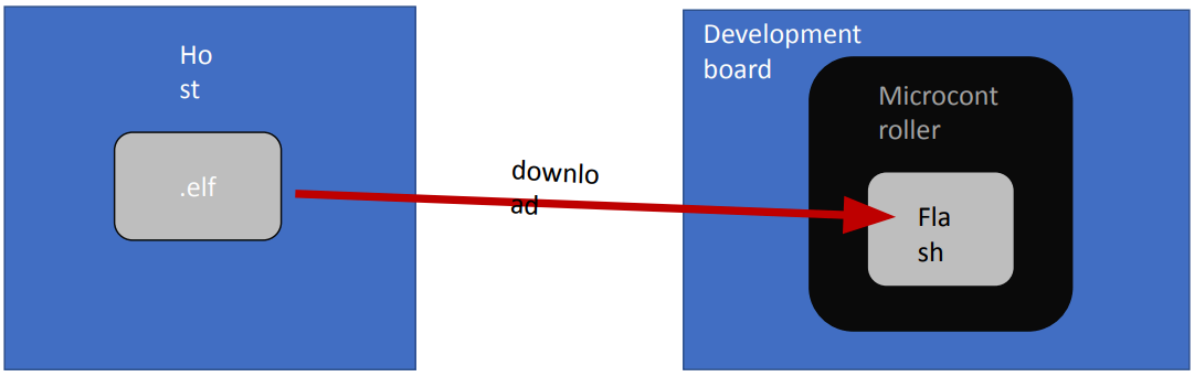
## 3. Call main()

**Add the prototype also for main() in startup.c file.**

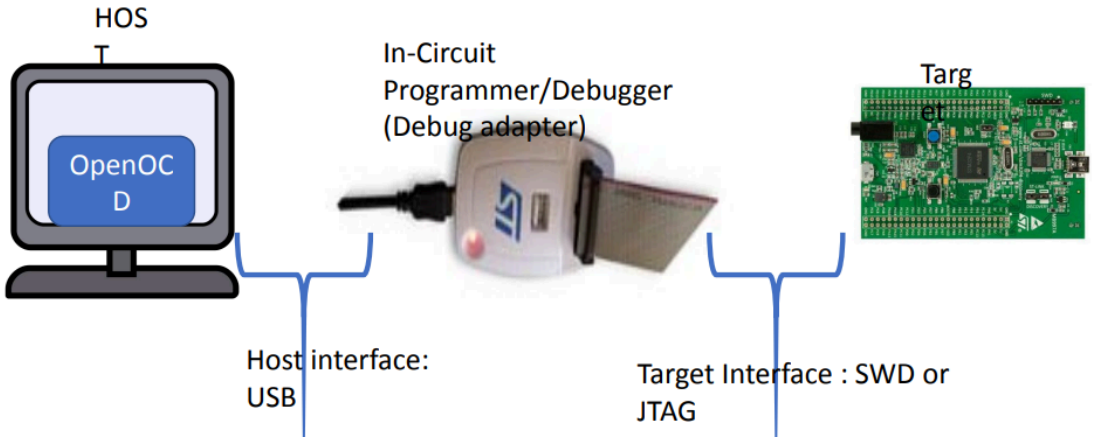
```
// Prototype of main
int main(void);
/*3. call main()*/
main();
```

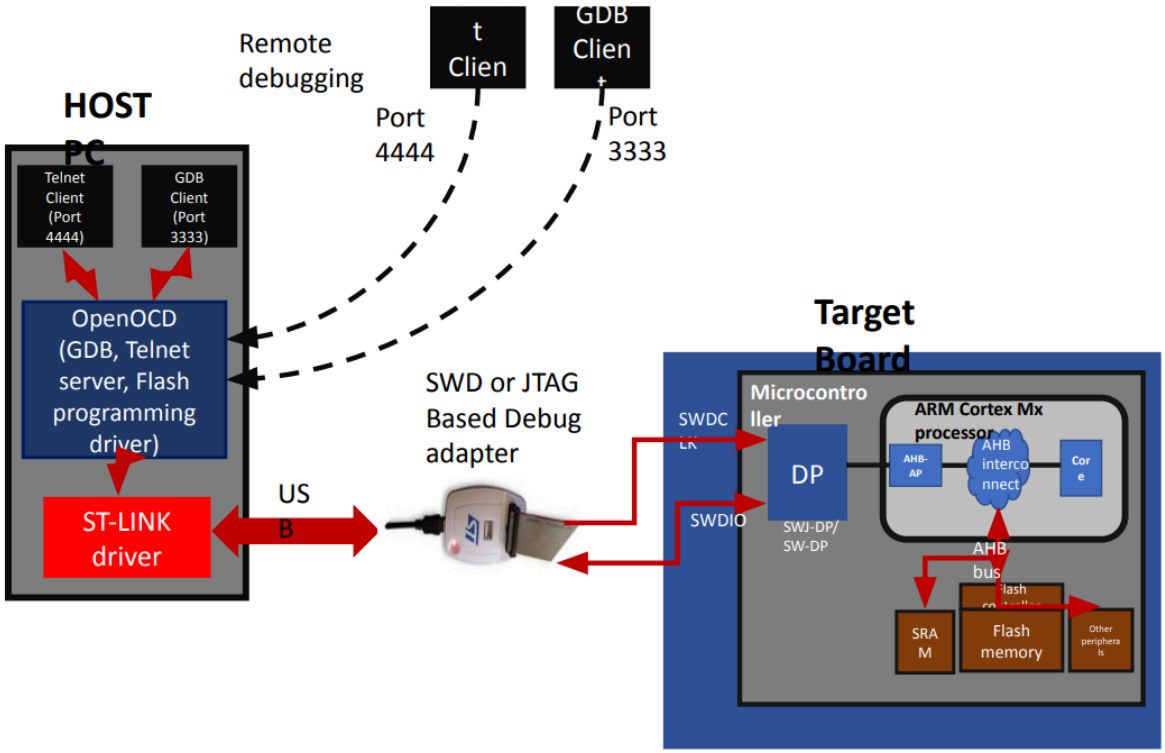
## Download .efl in the Target

Downloading and debugging executable




Downloading executable to Target






**USB Packet -> SWD Packet(carries Address and Data) -> AHB Interconnect -> AHB Bus -> FLASH Controller -> FLASH Memory**

## C Standard Library Integration

To link our project with the C standard library we need **newlib** and **newlib-nano.**

# Newlib

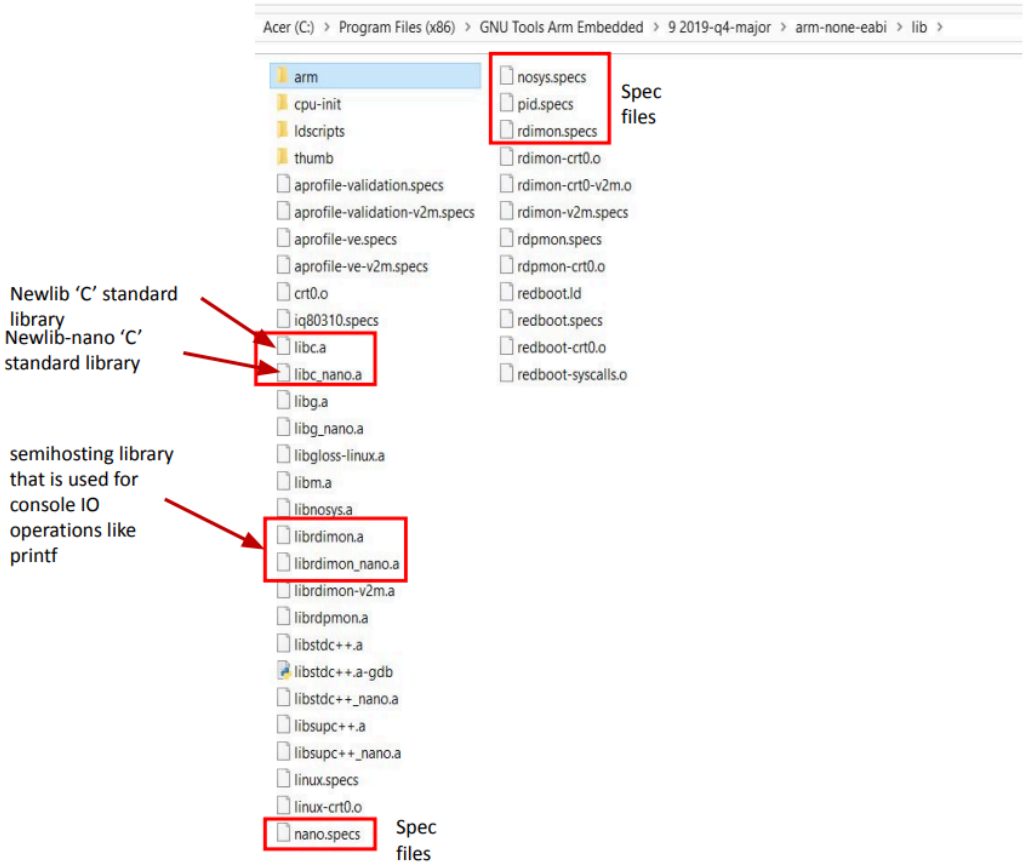
- Newlib is a ‘C’ standard library implementation intended for use on embedded systems, and it is introduced by Cygnus Solutions (now Red Hat)
- “Newlib” is written as a Glibc(GNU libc) replacement for embedded systems. It can be used with no OS (“bare metal”) or with a lightweight RTOS
- Newlib ships with gnu ARM toolchain installation as the default C standard library
- GNU libc (glibc) includes ISO C, POSIX, System V, and XPG interfaces. uClibc provides ISO C, POSIX and System V, while Newlib provides only ISO C

# Newlib-nano

- Due to the increased feature set in newlib, it has become too bloated to use on the systems where the amount of memory is very much limited.
- To provide a C library with a minimal memory footprint, suited for use with micro-controllers, ARM introduced newlib-nano based on newlib

**newlib-nano** does not support **float** by default. For that we have set some extra compiler arguments.

# Locating newlib and newlib nano




`.spec` Files are used in the linking phase to **link our applications with these lib files.**

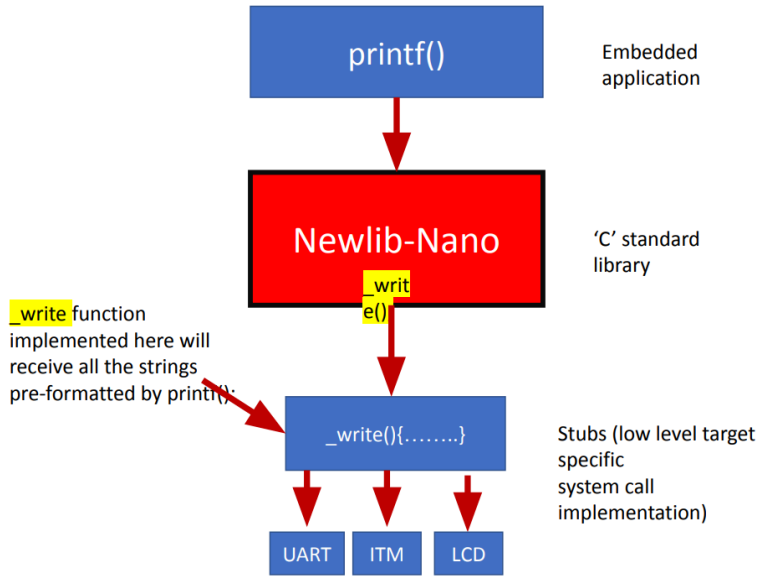
## Low Level SystemCalls

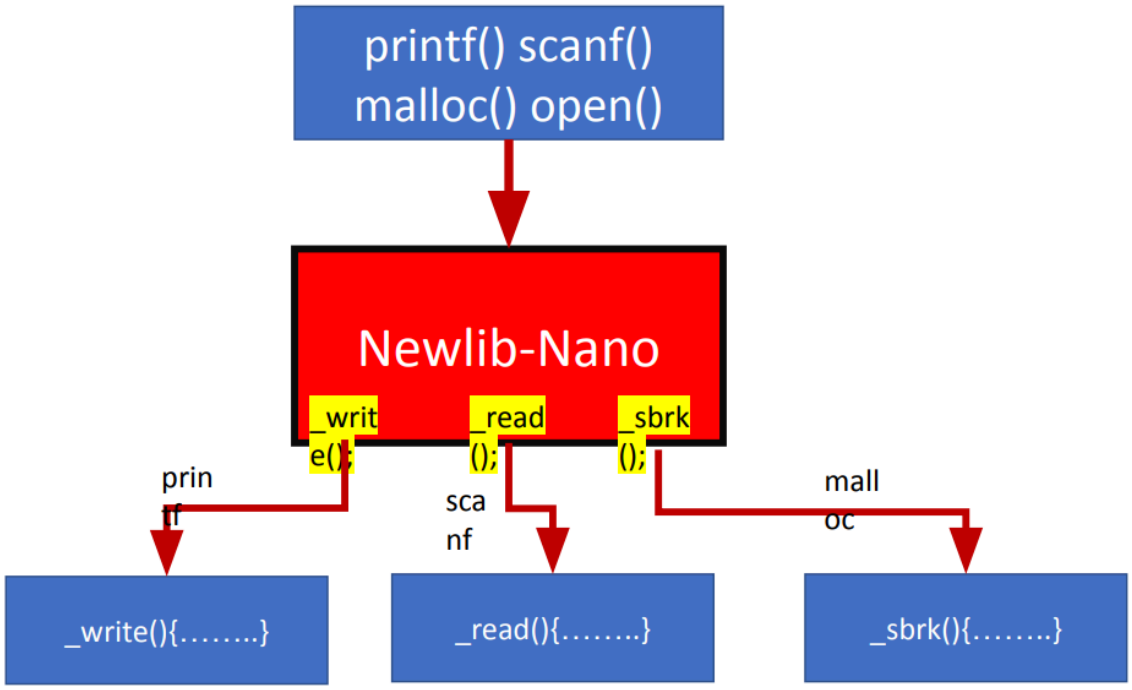
# Low level system Calls

- The idea of Newlib is to implement the hardware-independent parts of the standard C library and rely on a few low-level system calls that must be implemented with the target hardware in mind.
- When you are using newlib , you must implement the system calls appropriately to support devices, file-systems, and memory management.

Ex : we want to perform printf(). **printf()** is provided by **c standard library**. Then **newlib** calls **_write() low-level systemcall**. Now this _write() should be implemented by us. What we want to do in _write() that must be provided by us and it depends on the HW(Target). We can write in UART or LCD or ITM.

**newlib** redirects **printf()** to the lowlevel **_write()**






Stubs (low level target specific system call implementation)

# System Calls

- Please download the system calls implementation file syscalls.c attached with this lecture and place it in your workspace

We have to modify the makefile to add syscalls.c

```
all: main.o led.o startup.o syscalls.o final.elf

syscalls.o : syscalls.c
    $(CC) $(CFLAGS) -o $@ $^

final.elf : main.o led.o startup.o syscalls.o
    $(CC) $(LDFLAGS) -o $@ $^
```

Then we need to modify the linker flags. In the linker flags we have to mention newlib specific specs files.

```
LDFLAGS = --specs=nano.specs -T stm32_ls.ld -Wl,-Map=final.map
```

If not working then use :

```
LDFLAGS = --specs=nosys.specs -T stm32_ls.ld -Wl,-Map=final.map
```

Remove `-nostdlib` and add `--specs=nano.specs` . This will link our project with newlib-nano C standard library. And now we can enable all the printf().

Now we get the error in syscall.c :

```
D:\Git\TOOLS\compiler\gcc-linaro-6.3.1-2017.05-i686-mingw32_arm-eabi/
bin/arm-eabi-gcc.exe --specs=nosys.specs -T stm32_ls.ld
-Wl,-Map=final.map -o final.elf main.o led.o startup.o syscalls.o
d:/git/tools/compiler/gcc-linaro-6.3.1-2017.05-i686-mingw32_arm-eabi/
bin/../arm-eabi/libc/usr/lib/crt0.o: In function `_start':
/home/tcwg-buildslave/workspace/tcwg-make-release/builder_arch/amd64/
label/tcwg-x86_64-build/target/arm-eabi/snapshots/newlib.git~linaro-l
ocal~linaro-2.4-branch/newlib/libc/sys/arm/crt0.S:405: undefined
reference to `__bss_start__'
/home/tcwg-buildslave/workspace/tcwg-make-release/builder_arch/amd64/
label/tcwg-x86_64-build/target/arm-eabi/snapshots/newlib.git~linaro-l
ocal~linaro-2.4-branch/newlib/libc/sys/arm/crt0.S:405: undefined
reference to `__bss_end__'
/home/tcwg-buildslave/workspace/tcwg-make-release/builder_arch/amd64/
label/tcwg-x86_64-build/target/arm-eabi/snapshots/newlib.git~linaro-l
ocal~linaro-2.4-branch/newlib/libc/sys/arm/crt0.S:405: undefined
reference to `__end__'
syscalls.o: In function `_sbrk':
syscalls.c:(.text+0x258): undefined reference to `end'
collect2.exe: error: ld returned 1 exit status
make: *** [makefile.mak:36: final.elf] Error 1
```

To resolve this issue, we have to modify the linker script file and add reference for the missing variables.

```
syscalls.o: In function `_sbrk':
syscalls.c:(.text+0x258): undefined reference to `end'
```

'end' : this symbol helps memory management function to **locate the end of HEAP**. That is why we have to mention the **start of Heap to this symbol.**

syscalls.c

```
caddr_t _sbrk(int incr)
{
    extern char end asm("end");
    static char *heap_end;
    char *prev_heap_end;

    if (heap_end == 0)
        heap_end = &end;
```

Here this 'end' should come from linker script. `end = .;` after .bss ends.

```
.bss :
    {
        _sbss = .;
        __bss_start__ = _sbss;
        *(.bss)
        *(.COMMON)
        . = ALIGN(4);
        _ebss = .;
        __bss_end__ = _ebss;
        . = ALIGN(4);
        end = .;
        __end__ = .;
    }> SRAM
```

Now run : `arm-eabi-objdump -h final.elf` to check all the sections with all c std libraries.

Every function is considered as a subtext section. Now we have to resolve these into a single .text section. For that add : `*(.text.*)` and `*(.data.*)` in the .ld file.

```
SECTIONS
{
    .text :
    {
        *(.isr_vector)
        *(.text)
        *(.text.*)
        *(.rodata)
        . = ALIGN(4);
        _etext = .;
    }> FLASH

    .data :
    {
        _sdata = .;
        *(.data)
        *(.data.*)
        . = ALIGN(4);
        _edata = .;
    }> SRAM AT> FLASH  /*>VMA AT> LMA*/
```

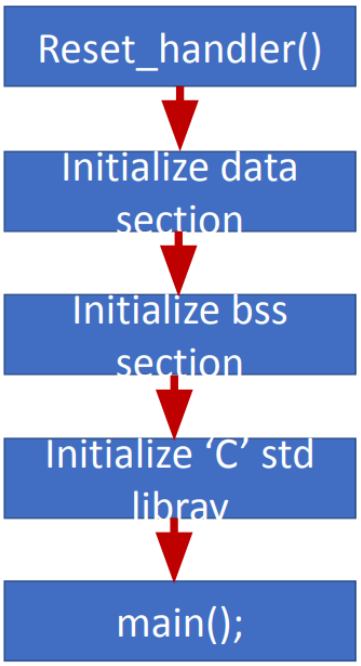
Now we further merge few more sections to .text where VMA and LMA are same.

```
final.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .note.gnu.build-id 00000024  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         000033fc  08000024  08000024  00010024  2**2
```

```
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .init          00000018  08003420  08003420  00013420  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .fini          00000018  08003438  08003438  00013438  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .eh_frame      00000004  08003450  08003450  00013450  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 5 .ARM.exidx     00000008  08003454  08003454  00013454  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .rodata.str1.4 00000008  0800345c  0800345c  0001345c  2**2
```

If there is any unmerged section in the application, then linker will include them as it is in the output file. That is why so many sections are there in op file.

When microcontroller resets, first control comes to the **Reset_Handler()**. Then we initialize .data , .bss. And now we initialize **C std Library** in the **startup_file.c**.




__libc_init_array(
);

For that we have to call : **__libc_init_array();**
Before calling main() in Reset_Handler(). And add prototype of the above function in startup.c file. `void __libc_init_array(void);`

```
 /*3.call init functions of std library*/

    __libc_init_array(); // add prototype also

    /*3. call main()*/

    main();
}
```

Let's say, I write one printf statement here.Here, printf("implementation of simple task scheduler"). Now the question is how do you see this printf when the code is running? You can't see, isn't it? Because, we don't have any output terminal or output device by which we can see this text message. When you call printf newlib calls write system call, which is there in syscalls.c _write.In the _write.

Now we have to modify the makefile for floating point operations 👍

```
#variables
CC = $(TOOLPATH_COMPILER)/bin/arm-eabi-gcc.exe
MACH = cortex-m4
STD = gnu11
CFLAGS = -c -mcpu=$(MACH) -mthumb -std=$(STD) -O0
# Now add command and flags for linker
# -T is used for linker
# -nostdlib is used for not include statndard libary (printf,scanf won't work)
LDFLAGS = --specs=nosys.specs -T stm32_ls.ld -Wl,-Map=final.map
```

We have to add compiler flags to Linker flags as well for C standard Library to work.

Then we have to add option for floating point from [ARM Options (Using

the GNU Compiler Collection (GCC))

`-mfloat-abi=name`

Specifies which floating-point ABI to use. Permissible values are: '`soft`', '`softfp`' and '`hard`'.

Specifying '`soft`' causes GCC to generate output containing library calls for floating-point operations. '`softfp`' allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. '`hard`' allows generation of floating-point instructions and uses FPU-specific calling conventions.

The default depends on the specific target configuration. Note that the hard-float and soft-float ABIs are not link-compatible; you must compile your entire program with the same ABI, and link with a compatible set of libraries.

We will be adding 'soft' option. Software support via Library. Hard -> Hardware Support. `-mfloat-abi=soft`

```
#variables
CC = $(TOOLPATH_COMPILER)/bin/arm-eabi-gcc.exe
MACH = cortex-m4
STD = gnu11
CFLAGS = -c -mcpu=$(MACH) -mthumb -mfloat-abi=soft -std=$(STD) -O0
# Now add command and flags for linker
# -T is used for linker
# -nostdlib is used for not include statndard libary (printf,scanf
won't work)
LDFLAGS = -mcpu=$(MACH) -mthumb -mfloat-abi=soft --specs=nosys.specs
-T stm32_ls.ld -Wl,-Map=final.map
```

Here if we load .elf in target, we can get Hard-fault because, we have resolved the .text and .data section address in startup.c and linker file.

Now we need to modify the .data section address, as new sections get added.

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .note.gnu.build-id 00000024  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         0000342c  08000024  08000024  00010024  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .eh_frame     00000004  08003450  08003450  00013450  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .ARM.exidx    00000008  08003454  08003454  00013454  2**2
```

```
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .rodata.str1.4 00000008  0800345c  0800345c  0001345c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .data          00000960  20000000  08003464  00020000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  6 .init_array    00000004  20000960  08003dc4  00020960  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Between .text and .data sections, other sections are there : So now **.data section does not start exactly after the .text section.**

stm32_ls.ld

```
/*this will give laod address of the data section. and have to modify satrtup.c accordingly*/

    _la_data = LOADADDR(.data);

    .data :
    {
        _sdata = .;
        *(.data)
        *(.data.*)
        . = ALIGN(4);
        _edata = .;
    }> SRAM AT> FLASH  /*>VMA AT> LMA*/
```

Extern this address variable to startup.c and replace _etext with _la_data `extern uint32_t _la_data;`

```
/*1. copy .data section to SRAM*/

// size of the section
uint32_t size = (&_edata - &_sdata);

// src and dst pointer for copying
uint8_t *pDst = (uint8_t*)&_sdata; // SRAM
uint8_t *pSrc = (uint8_t*)&_etext; // FLASH

// start copying from FLASH to SRAM
for(uint32_t i = 0; i<size; i++)
{
    *pDst++ = *pSrc++;
}
```

```
/*1. copy .data section to SRAM*/

// size of the section
uint32_t size = (&_edata - &_sdata);

// src and dst pointer for copying
uint8_t *pDst = (uint8_t*)&_sdata; // SRAM
uint8_t *pSrc = (uint8_t*)&_la_data; // FLASH

// start copying from FLASH to SRAM
for(uint32_t i = 0; i<size; i++)
{
    *pDst++ = *pSrc++;
}
```

## Semi Hosting: