Building and running bare metal executables for ARM target using GNU tools

INDEX:

1. Introduction

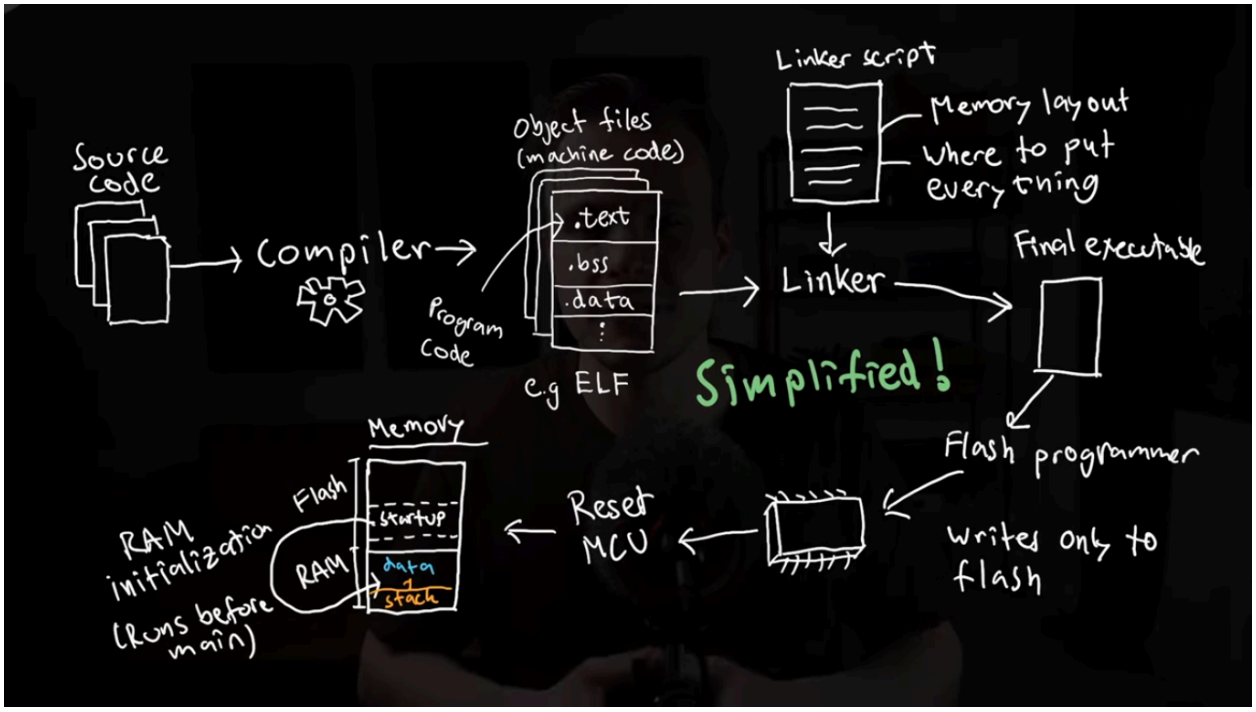
1.1 Memory types

1. Flash : non volatile , data stay even if power is off
2. RAM : volatile , data erased after power is off
3. EEPROM : smaller type of memory used for storing the configuration data

Flash is a kind of EEPROM technically

Flash	**RAM**
Store the program code like instructions CPU execution	Stores the program variables and program data
The inside data of flash remain the same throughout the execution of program	data changes depending on the function you are currently in
Larger than RAM (512Kb)	Smaller than flash (96Kb)

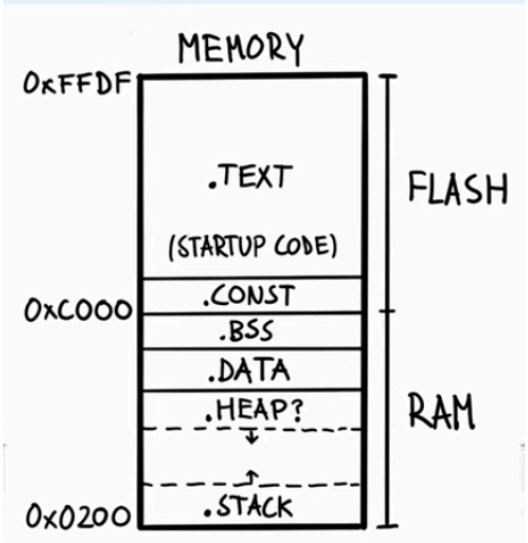
1.2 Source code to machine code conversion



1. You have written a bunch of code (C files)
2. Compile this code into machine code (object files) using compiler
3. Compiler is also going to divide this machine code into different sections which depends on the particular compiler and object format its using
4. The command object format used by many compilers is the object format called elf (executable and linkable format)
5. The program code is stored in the section called .text and data stored into different sections like .bss , .rodata and .data
6. We cant put the machine code into the microcontroller because we havent specified where inside the memory the machine code should end up that will be the main job of the linker
7. Linker is going to take the linker script (it describes the memory layout of the microcontroller , it contains information about where the flash memory starts at what address and how big it is and same information for RAM also . It also explains where the sections of the object files should end up inside the memory. Linker takes this linker script together with the object files and turns everything into single final executable file (.exe)
8. The flash programmer then takes the machine code of this executable file and writes it to the flash memory but it doesn't touch the RAM because the RAM is not touched until we reset my controller and CPU starts executing instructions

9. The first thing that the CPU does when the controller boots up is like before the main function is that it's going to run a piece of code called startup code
10. The big part of the startup code is to
    - initialize the RAM with variables like global variables initialized with some values
    - initialize the uninitialized variables with zero.
    - Set the stack pointer to initialize stack which is the section of memory used for storing the context of a function

1.3 Sections of Code



Variable type	**Stored location in memory**
Initialized global variables	(.data + .const) > RAM+Flash
constant variables	(.const) >Flash
global variable uninitialized	(.bss) >RAM
local variable initialized	(.stack + .const) > RAM+Flash
static variable initialized	(.data+.const) > RAM+Flash

program code	(.text) > Flash

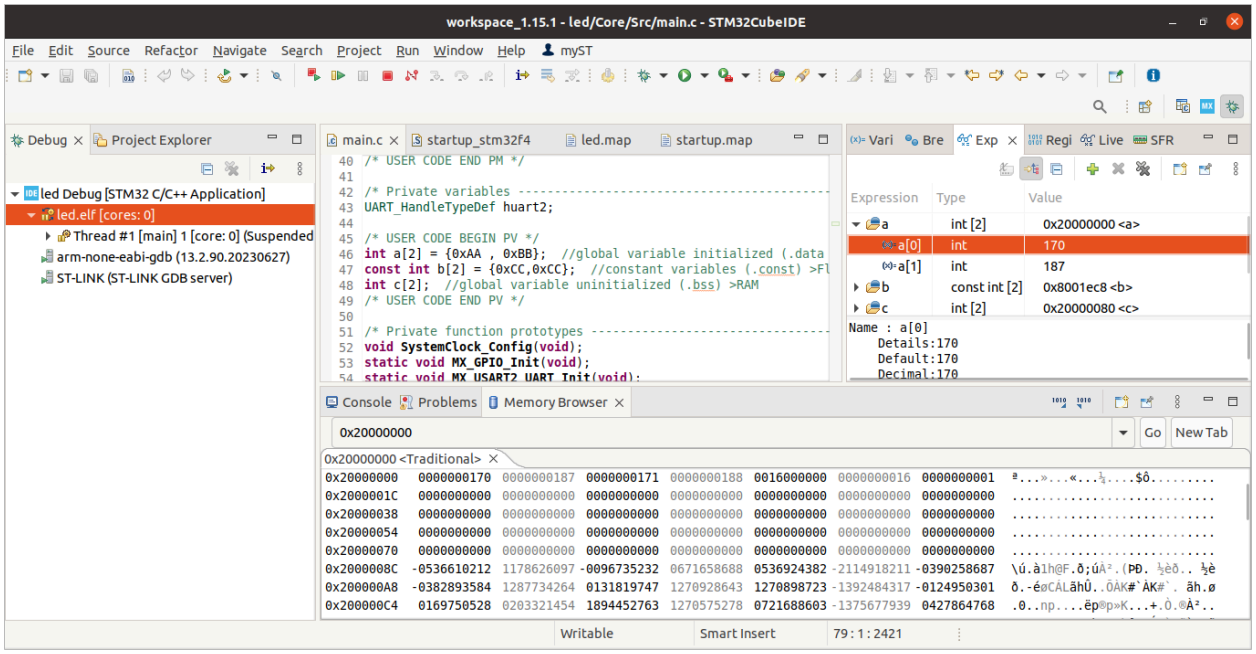
Example Code :

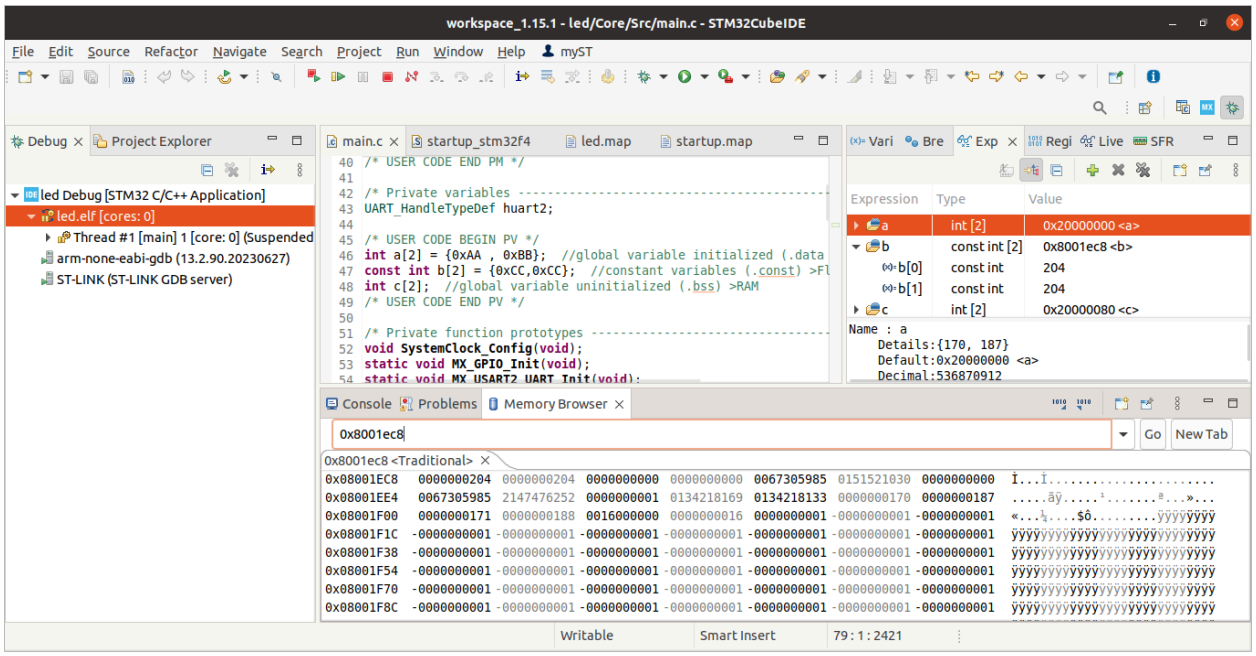
```
#include <stdio.h>

int a[2] = {0xAA , 0xBB};  //global variable initialized (.data + .const)  > RAM+Flash
Const int b[2] = {0xCC,0xCC};  //constant variables (.const) >Flash
Int c[2];  //global variable uninitialized (.bss) >RAM

void main(void)
{
        int d[2] = {0xDD,0xDD}; //local variable initialized (.stack + .const) > RAM+Flash
        Static int e[2] = {0xAB,0xBC};  //static variable initialized (.data+.const) > RAM+Flash

        Volatile unsigned int i;

        int sum = 0;  //local variable initialized (.stack) > RAM
        for(i=0;i<2;i++){ //program code (.text) > Flash
                sum += a[i]+b[i]+c[i]+d[i]+e[i];  //program code (.text) > Flash
        }
        while(1)
        {
                for(i=1000;i>0;i- -);
        }
}
```

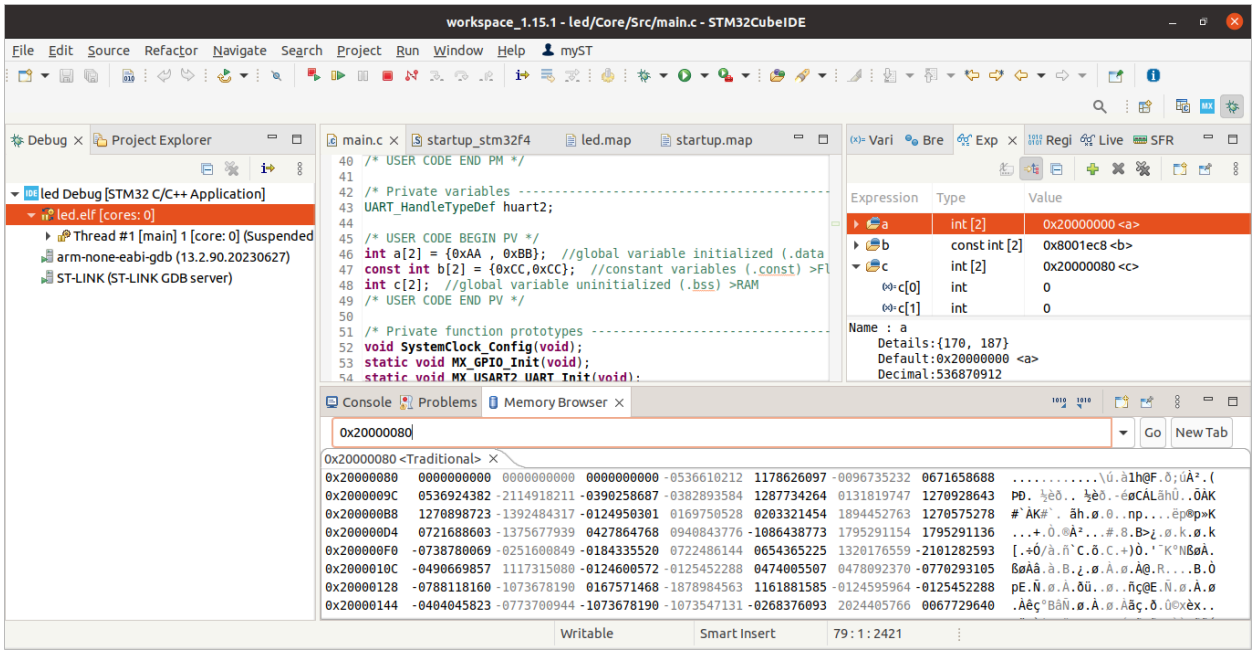
The copy instructions will be part of .text (not the data)

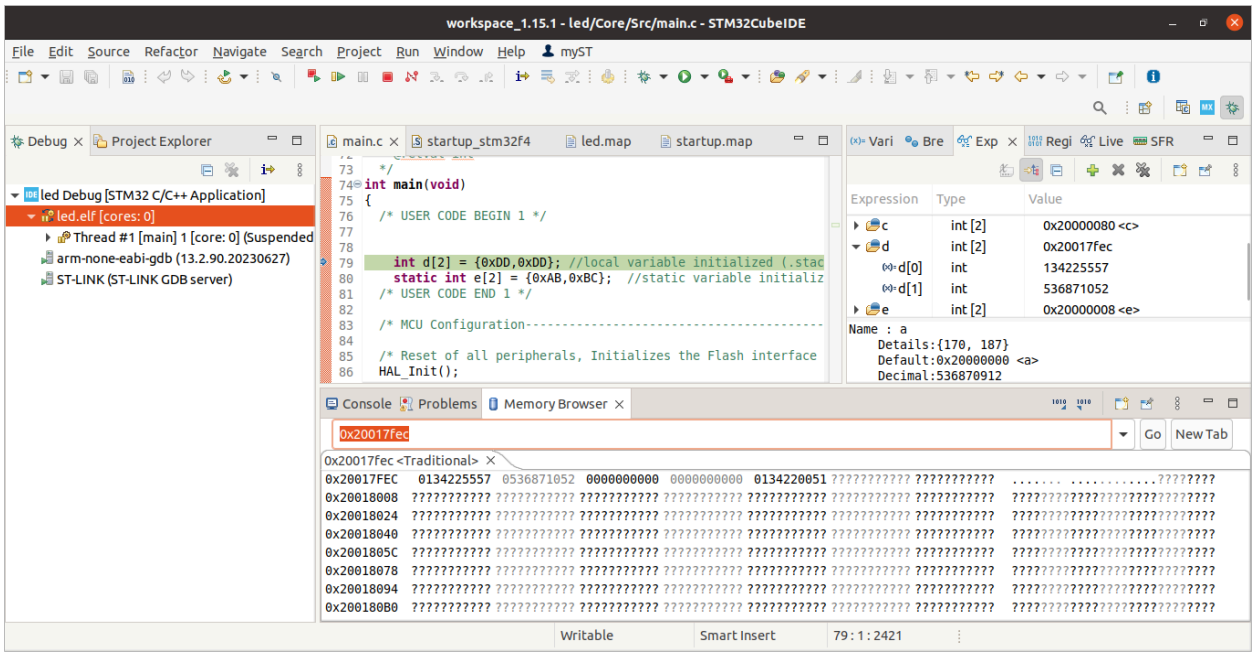
1.4 Map file

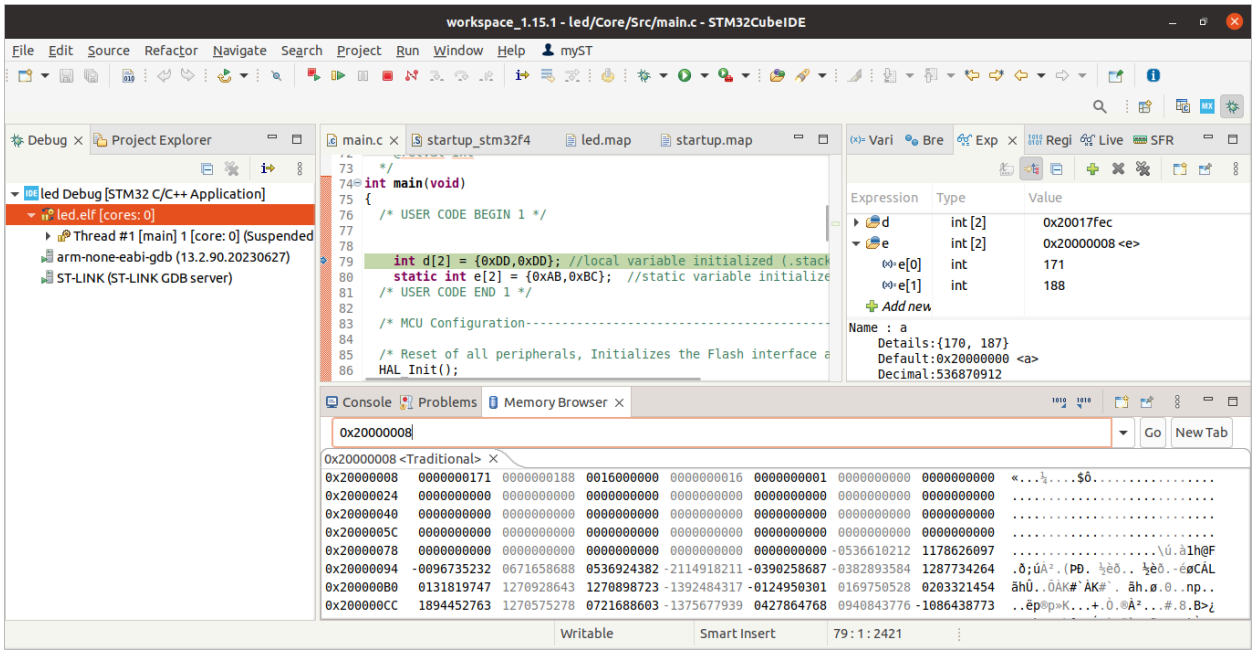
It tells us where everything ended up in memory after linker is done


workspace_1.15.1 - led/Core/Src/main.c - STM32CubeIDE
File Edit Source Refactor Navigate Search Project Run Window Help myST
Debug Project Explorer
led Debug [STM32 C/C++ Application]
led.elf [cores: 0]
Thread #1 [main] 1 [core: 0] (Suspended
arm-none-eabi-gdb (13.2.90.20230627)
ST-LINK (ST-LINK GDB server)
main.c startup_stm32f4 led.map startup.map
40 /* USER CODE END PM */
41
42 /* Private variables ----------
43 UART_HandleTypeDef huart2;
44
45 /* USER CODE BEGIN PV */
46 int a[2] = {0xAA , 0xBB}; //global variable initialized (.data
47 const int b[2] = {0xCC,0xCC}; //constant variables (.const) >Fl
48 int c[2]; //global variable uninitialized (.bss) >RAM
49 /* USER CODE END PV */
50
51 /* Private function prototypes ----------
52 void SystemClock_Config(void);
53 static void MX_GPIO_Init(void);
54 static void MX_USART2_UART_Init(void);
Vari Bre Exp Regi Live SFR
Expression Type Value
a int [2] 0x20000000 <a>
a[0] int 170
a[1] int 187
b const int [2] 0x8001ec8 <b>
c int [2] 0x20000080 <c>
Name : a[0]
Details:170
Default:170
Decimal:170
Console Problems Memory Browser
0x20000000
Go New Tab
0x20000000 <Traditional>
0x20000000 0000000170 0000000187 0000000171 0000000188 0016000000 0000000016 0000000001
0x2000001C 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000
0x20000038 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000
0x20000054 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000
0x20000070 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000
0x2000008C -0536610212 1178626097 -0096735232 0671658688 0536924382 -2114918211 -0390258687
0x200000A8 -0382893584 1287734264 0131819747 1270928643 1270898723 -1392484317 -0124950301
0x200000C4 0169750528 0203321454 1894452763 1270575278 0721688603 -1375677939 0427864768
Writable Smart Insert 79 : 1 : 2421


workspace_1.15.1 - led/Core/Src/main.c - STM32CubeIDE
File Edit Source Refactor Navigate Search Project Run Window Help myST
Debug Project Explorer
led Debug [STM32 C/C++ Application]
led.elf [cores: 0]
Thread #1 [main] 1 [core: 0] (Suspended
arm-none-eabi-gdb (13.2.90.20230627)
ST-LINK (ST-LINK GDB server)
main.c startup_stm32f4 led.map startup.map
40 /* USER CODE END PM */
41
42 /* Private variables ----------
43 UART_HandleTypeDef huart2;
44
45 /* USER CODE BEGIN PV */
46 int a[2] = {0xAA , 0xBB}; //global variable initialized (.data
47 const int b[2] = {0xCC,0xCC}; //constant variables (.const) >Fl
48 int c[2]; //global variable uninitialized (.bss) >RAM
49 /* USER CODE END PV */
50
51 /* Private function prototypes ----------
52 void SystemClock_Config(void);
53 static void MX_GPIO_Init(void);
54 static void MX_USART2_UART_Init(void);
Vari Bre Exp Regi Live SFR
Expression Type Value
a int [2] 0x20000000 <a>
b const int [2] 0x8001ec8 <b>
b[0] const int 204
b[1] const int 204
c int [2] 0x20000080 <c>
Name : a
Details:{170, 187}
Default:0x20000000 <a>
Decimal:536870912
Console Problems Memory Browser
0x8001ec8
Go New Tab
0x8001ec8 <Traditional>
0x08001EC8 0000000204 0000000204 0000000000 0000000000 0067305985 0151521030 0000000000
0x08001EE4 0067305985 2147476252 0000000001 0134218169 0134218133 0000000170 0000000187
0x08001F00 0000000171 0000000188 0016000000 0000000016 0000000001 -0000000001 0000000001
0x08001F1C -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001
0x08001F38 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001
0x08001F54 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001
0x08001F70 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001
0x08001F8C -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001 -0000000001
Writable Smart Insert 79 : 1 : 2421


workspace_1.15.1 - led/Core/Src/main.c - STM32CubeIDE
File Edit Source Refactor Navigate Search Project Run Window Help myST
Debug Project Explorer
led Debug [STM32 C/C++ Application]
led.elf [cores: 0]
Thread #1 [main] 1 [core: 0] (Suspended
arm-none-eabi-gdb (13.2.90.20230627)
ST-LINK (ST-LINK GDB server)
main.c startup_stm32f4 led.map startup.map
40 /* USER CODE END PM */
41
42 /* Private variables ----------
43 UART_HandleTypeDef huart2;
44
45 /* USER CODE BEGIN PV */
46 int a[2] = {0xAA , 0xBB}; //global variable initialized (.data
47 const int b[2] = {0xCC,0xCC}; //constant variables (.const) >Fl
48 int c[2]; //global variable uninitialized (.bss) >RAM
49 /* USER CODE END PV */
50
51 /* Private function prototypes ----------
52 void SystemClock_Config(void);
53 static void MX_GPIO_Init(void);
54 static void MX_USART2_UART_Init(void);
Vari Bre Exp Regi Live SFR
Expression Type Value
a int [2] 0x20000000 <a>
b const int [2] 0x8001ec8 <b>
c int [2] 0x20000080 <c>
c[0] int 0
c[1] int 0
Name : a
Details:{170, 187}
Default:0x20000000 <a>
Decimal:536870912
Console Problems Memory Browser
0x20000080
Go New Tab
0x20000080 <Traditional>
0x20000080 0000000000 0000000000 0000000000 -0536610212 1178626097 -0096735232 0671658688
0x2000009C 0536924382 -2114918211 -0390258687 -0382893584 1287734264 0131819747 1270928643
0x200000B8 1270898723 -1392484317 -0124950301 0169750528 0203321454 1894452763 1270575278
0x200000D4 0721688603 -1375677939 0427864768 0940843776 -1086438773 1795291154 1795291136
0x200000F0 -0738780069 -0251600849 -0184335520 0722486144 0654365225 1320176559 -2101282593
0x2000010C -0490669857 1117315080 -0124600572 -0125452288 0474005507 0478092370 -0770293105
0x20000128 -0788118160 -1073678190 0167571468 -1878984563 1161881585 -0124595964 -0125452288
0x20000144 -0404045823 -0773700944 -1073678190 -1073547131 -0268376093 2024405766 0067729640
Writable Smart Insert 79 : 1 : 2421


workspace_1.15.1 - led/Core/Src/main.c - STM32CubeIDE
File Edit Source Refactor Navigate Search Project Run Window Help myST
Debug Project Explorer
led Debug [STM32 C/C++ Application]
led.elf [cores: 0]
Thread #1 [main] 1 [core: 0] (Suspended
arm-none-eabi-gdb (13.2.90.20230627)
ST-LINK (ST-LINK GDB server)
main.c startup_stm32f4 led.map startup.map
73 */
74 int main(void)
75 {
76 /* USER CODE BEGIN 1 */
77
78
79 int d[2] = {0xDD,0xDD}; //local variable initialized (.stac
80 static int e[2] = {0xAB,0xBC}; //static variable initializ
81 /* USER CODE END 1 */
82
83 /* MCU Configuration----------
84
85 /* Reset of all peripherals, Initializes the Flash interface
86 HAL_Init();
Vari Bre Exp Regi Live SFR
Expression Type Value
c int [2] 0x20000080 <c>
d int [2] 0x20017fec
d[0] int 134225557
d[1] int 536871052
e int [2] 0x20000008 <e>
Name : a
Details:{170, 187}
Default:0x20000000 <a>
Decimal:536870912
Console Problems Memory Browser
0x20017fec
Go New Tab
0x20017fec <Traditional>
0x20017FEC 0134225557 0536871052 0000000000 0000000000 0134220051 ?????????? ??????????
0x20018008 ?????????? ?????????? ?????????? ?????????? ?????????? ?????????? ??????????
0x20018024 ?????????? ?????????? ?????????? ?????????? ?????????? ?????????? ??????????
0x20018040 ?????????? ?????????? ?????????? ?????????? ?????????? ?????????? ??????????
0x2001805C ?????????? ?????????? ?????????? ?????????? ?????????? ?????????? ??????????
0x20018078 ?????????? ?????????? ?????????? ?????????? ?????????? ?????????? ??????????
0x20018094 ?????????? ?????????? ?????????? ?????????? ?????????? ?????????? ??????????
0x200180B0 ?????????? ?????????? ?????????? ?????????? ?????????? ?????????? ??????????
Writable Smart Insert 79 : 1 : 2421



2. Cross compilation and toolchains

2.1 What is cross compilation?

Cross-compilation is a process in which the cross-toolchain uns on the host machine(PC) and creates executables that run on different machine (ARM)

In our case the host machine is our PC and target machine is out STM32 microcontroller

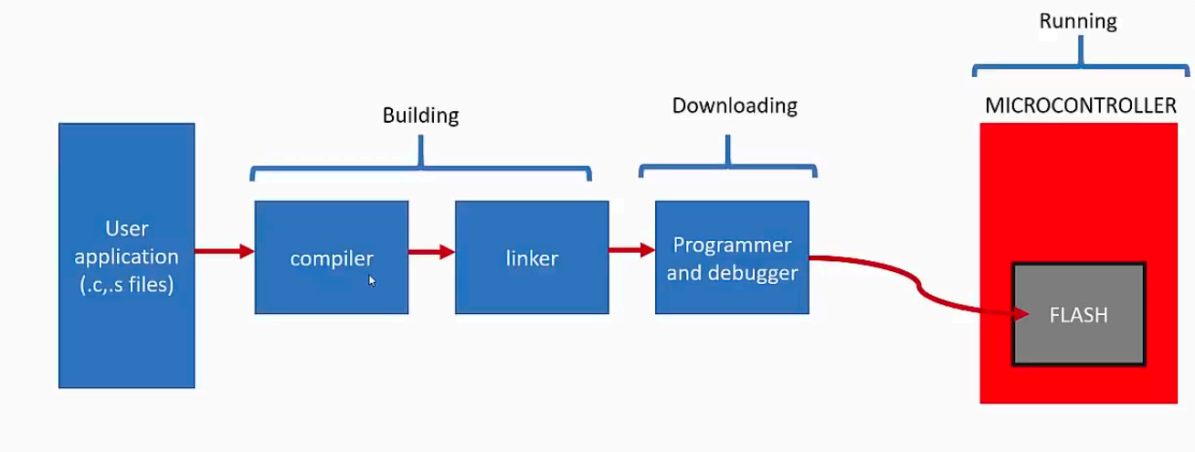
2.2 Cross-compilation toolchains

- Toolchain or a cross-compilation toolchain is a collection of binaries which allows you to compile, assemble and 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 executables
    - Disassemble
    - Extract symbol and size information

- Convert executable to other formats such as bin,ihex(intel hex)
- Provides ‘C’ standard libraries

2.3 Popular toolchains for ARM architecture

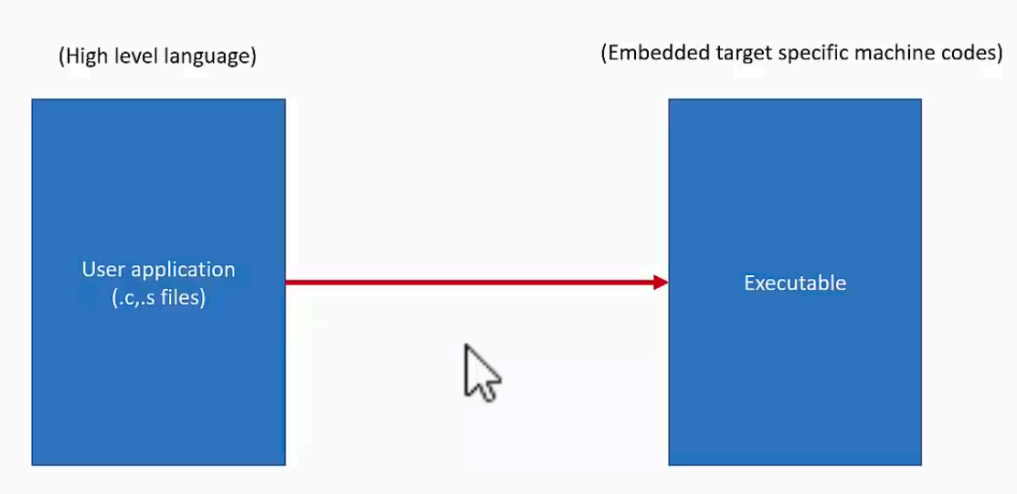
1. GNU tools (GCC) for ARM embedded processors (free and open source)
2. Armcc from ARM ltd. (ships with KEIL , code restriction version , require licensing)

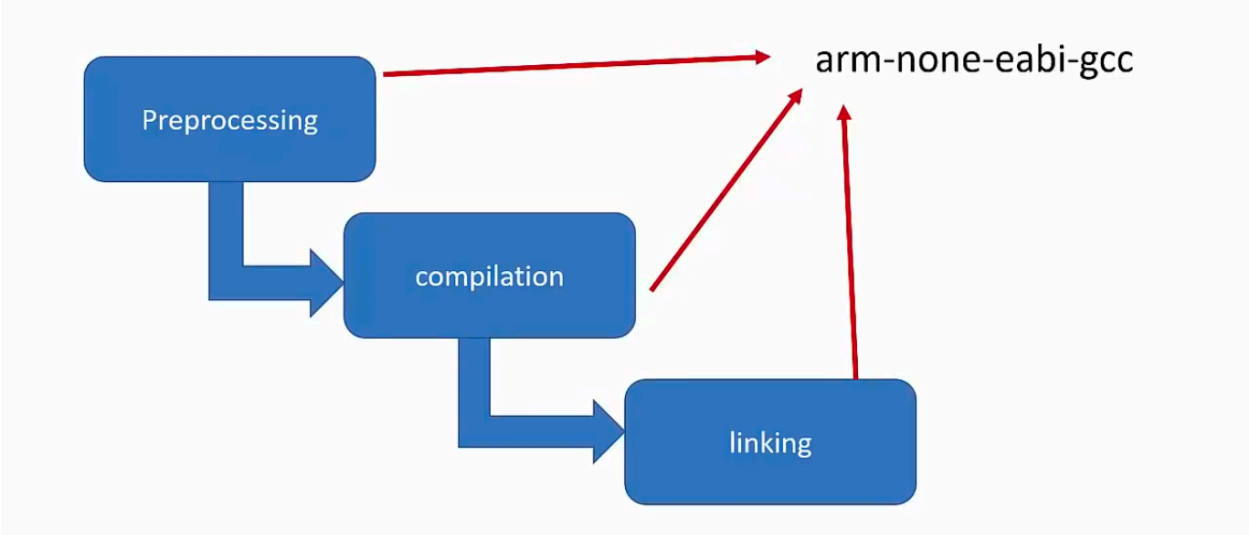


2.4 Cross toolchain important binaries

Toolchain	**Purpose of Use**
arm-none-eabi-gcc	Compiler, linker, assembler
arm-none-eabi-ld	Explicit linker
arm-none-eabi-as	Explicit assembler (convert assembly level language to machine codes)
arm-none-eabi-objdump arm-none-eabi-readelf arm-none-eabi-nm	ELF file analyzer
arm-none-eabi-objcopy	Format converter (convert from one executable format to other)

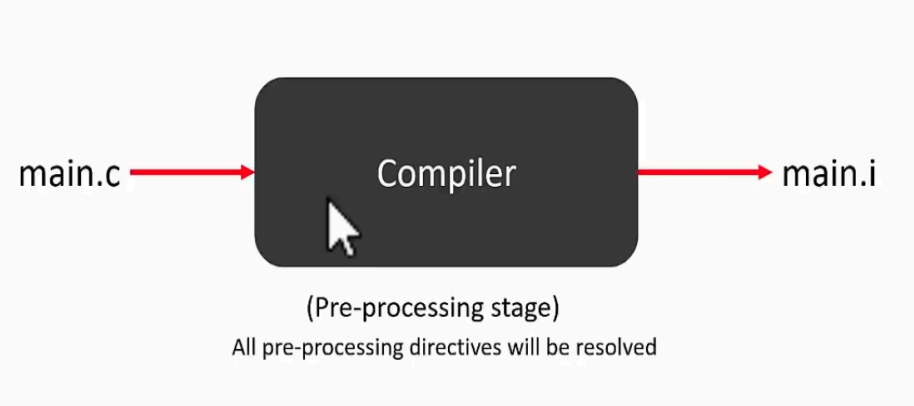
3. Build Process





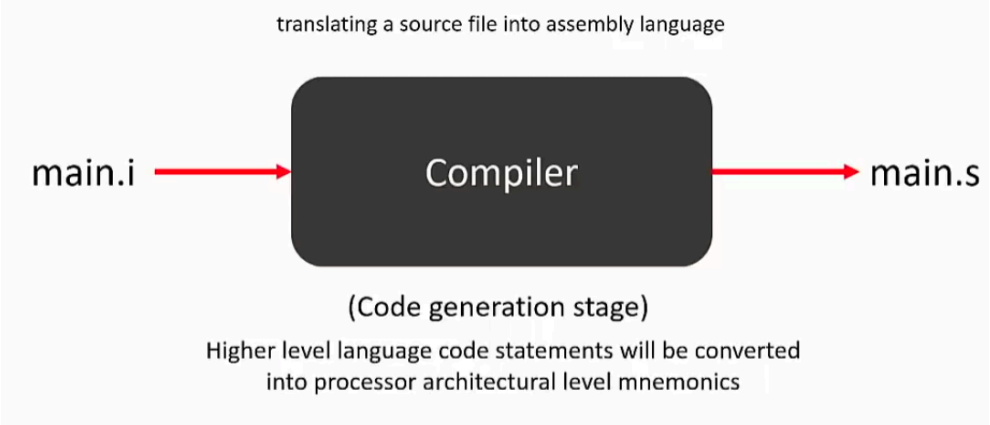
3.1 Conversion of high-level language to machine code

3.1.1 Preprocessing stage of compilation



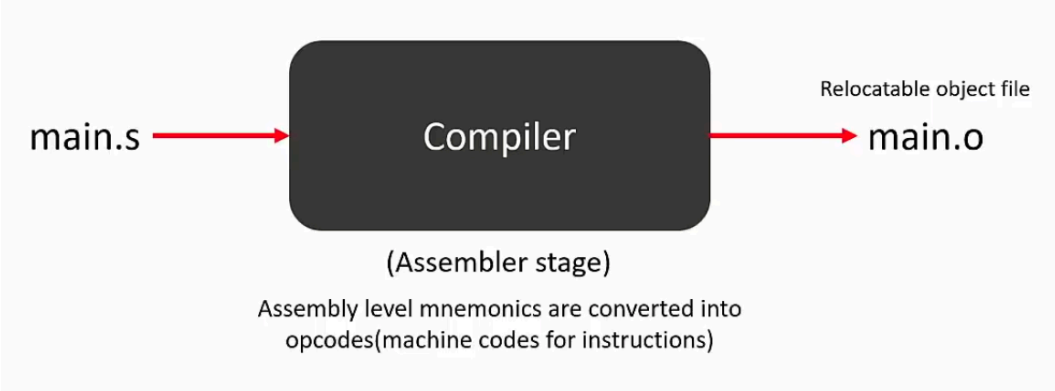
1. First when you invoke the compiler to compile your source file , first stage is the preprocessing stage.
2. In this stage all preprocessing directives of the source file will be resolved and outfile main.i is created
3. The preprocessor directives such as #includes , C macros or conditional compilation macros of the source file will be resolved and preprocessed file is created that have extension of ".i"

3.1.2 Code generation stage

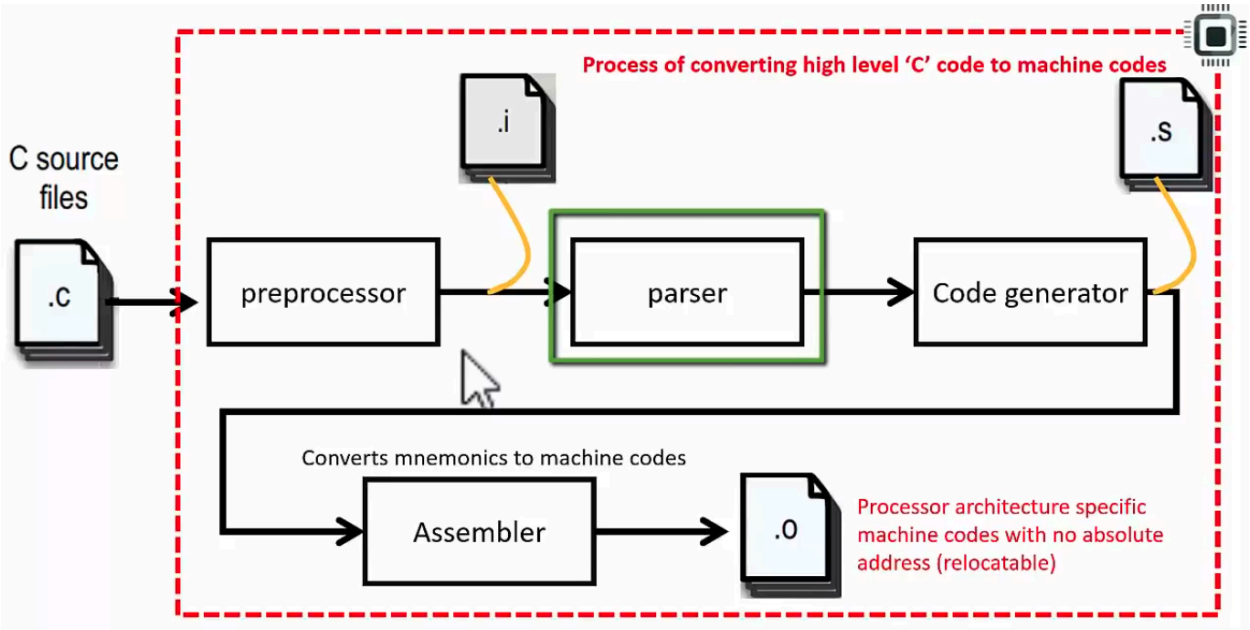


1. In this stage the source file is translated to assembly language
2. Higher level language code statements will be converted into processor architectural level mnemonics
3. Here high level means the C language statements will be converted assembly level language thats what we call as mnemonics

3.1.3 Assembler stage

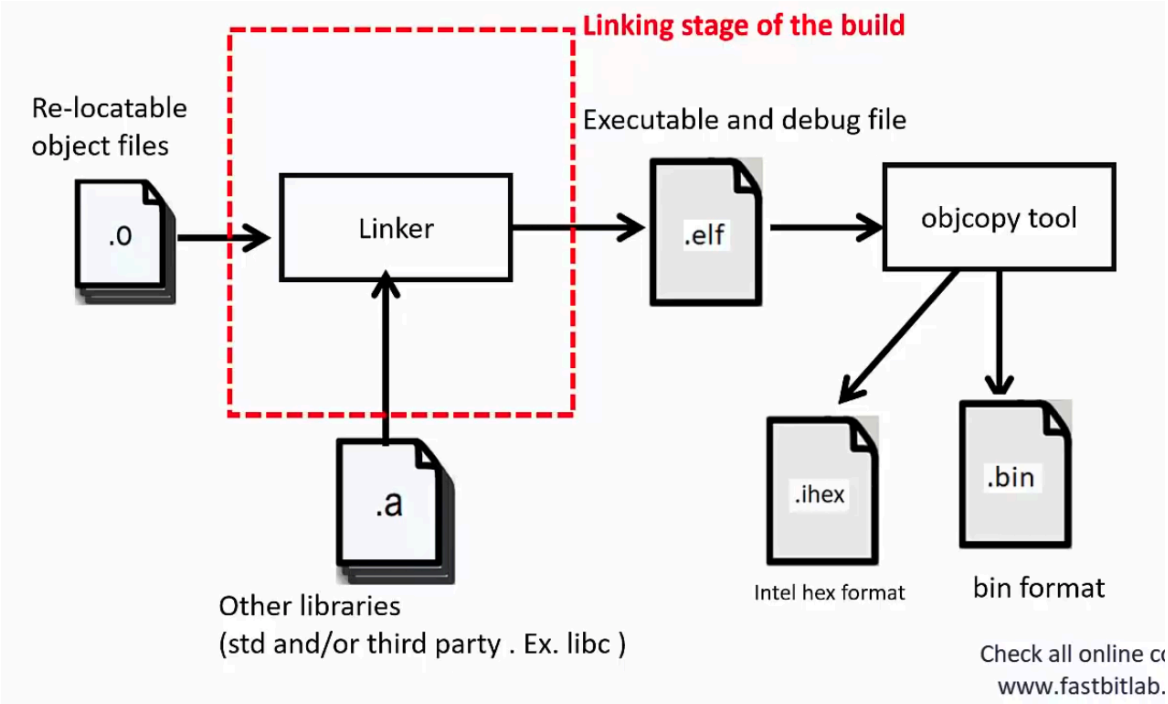


1. In this stage all things will be done by the compiler itself , you need not to invoke any other command
2. Here arm-none-eabi-gcc is our compiler as we are doing cross compilation
3. In assembler stage, the assembly level mnemonics are converted into opcodes nothing but machine codes for various instructions as our processor understands numbers or machine code
4. It creates output file called relocatable object file “main.o”



- Here in the parser,some checks will be carried out further of C syntax
- .o file contains processor architectures specific machine codes with no absolute addresses that's why it is called relocatable
- The machine codes which are stored in .o files they don't carry absolute address of the program memory, they carry an address which can be relocatable
- .i and .s files are not seen until we instruct the compiler to save those files

3.1.4 Linking stage



- In this stage all the relocatable object files will be taken by linker it will resolve all the symbols and other information
- It will merge different sections of .o files to create one executable that is final executable file(.elf : executable and linkable format)
- .elf file can be taken to create some other formats such as binary (.bin) and intel hex(.ihex) using the tool object copy

vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-gcc -c main.c -o main.o
/tmp/cchocbao.s: Assembler messages:
/tmp/cchocbao.s:309: Error: selected processor does not support requested special purpose register -- `msr MSP,r3'
/tmp/cchocbao.s:661: Error: selected processor does not support requested special purpose register -- `msr PSP,R0'
/tmp/cchocbao.s:670: Error: selected processor does not support requested special purpose register -- `msr CONTROL,R0'
/tmp/cchocbao.s:729: Error: selected processor does not support requested special purpose register -- `msr PRIMASK,R0'
/tmp/cchocbao.s:762: Error: selected processor does not support requested special purpose register -- `msr PRIMASK,R0'
/tmp/cchocbao.s:791: Error: selected processor does not support requested special purpose register -- `mrs R0,PSP'

/tmp/cchocbao.s:812: Error: selected processor does not support requested special purpose register -- `msr PSP,R0'

>>> here the assembler couldn't understand these assembly mnemonics, because we didn't mention the processor architecture

>>> **-mcpu = name[+extension]**

This name specifies the target ARM processor, GCC uses this name of the target ARM architecture

>>> **-mthumb** , if you don't use this option -marm option is taken by default

It helps the compiler to make a selection between generating code that executes in ARM and thumb states

ARM Cortex M processors only support thumb state

vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb main.c -o main.o
vlab@HYVLAB8:~/Desktop/cross-compilation$ ls
led.c led.h main.c main.h **main.o**
vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-gcc -S -mcpu=cortex-m4 -mthumb main.c -o main.s
vlab@HYVLAB8:~/Desktop/cross-compilation$ ls
led.c led.h main.c main.h main.o **main.s**

4. MakeFile and analyzing relocatable object file

4.1 MakeFile

```
CC = arm-none-eabi-gcc
MACH=cortex-m4
CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -O0

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

- CC is the compiler we are using arm-none-eabi-gcc
- MACH = type of the machine on which we are compiling
- CFLAGS = options we are using in command
- -O0 = optimization set to zero
- main.o = target file, also denoted by $@
- main.c = dependency file, also denoted by $^

vlab@HYVLAB8:~/Desktop/cross-compilation$ gvim MakeFile.mk
vlab@HYVLAB8:~/Desktop/cross-compilation$ mv MakeFile.mk Makefile
vlab@HYVLAB8:~/Desktop/cross-compilation$ ls
led.c led.h main.c main.h Makefile
vlab@HYVLAB8:~/Desktop/cross-compilation$ make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 main.c -o main.o
vlab@HYVLAB8:~/Desktop/cross-compilation$ ls
led.c led.h main.c main.h main.o Makefile

4.2 Analyzing .o files (relocatable object files)

- Main.o is in elf format (executable and linker format)
- ELF is a standard file format for object files and executable files use GCC
- A file format standard describes a way of organizing various elements (data,read only data,code,uninitialized data,etc) of a program in different sections

Other file formats

- The common object file format (COFF) : introduced by UNIX system V
- Arm image format (AIF) : introduced by ARM
- SRECORD : introduced by motorola

4.3 Relocatable object files

- It mainly contains machine code and the data of program , it doesn't contain any absolute addresses for data and code of the program
- In order to analyze the object file you can use command
    - **arm -none-eabi-objdump**

      vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump
      Usage: arm-none-eabi-objdump <option(s)> <file(s)>
       Display information from object <file(s)>.
       At least one of the following switches must be given:
        -a, --archive-headers   Display archive header information
        -f, --file-headers   Display the contents of the overall file header
        -p, --private-headers   Display object format specific file header contents
        -P, --private=OPT,OPT... Display object format specific contents
        -h, --[section-]headers Display the contents of the section headers

```
-x, --all-headers        Display the contents of all headers
-d, --disassemble        Display assembler contents of executable sections
-D, --disassemble-all    Display assembler contents of all sections
   --disassemble=<sym>  Display assembler contents from <sym>
-S, --source             Intermix source code with disassembly
   --source-comment[=<txt>] Prefix lines of source code with <txt>
-s, --full-contents      Display the full contents of all sections requested
-g, --debugging          Display debug information in object file
-e, --debugging-tags     Display debug information using ctags style
-G, --stabs              Display (in raw form) any STABS info in the file
-W[lLiaprmfFsoRtUuTgAckK] or
--dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
       =frames-interp,=str,=loc,=Ranges,=pubtypes,
       =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
       =addr,=cu_index,=links,=follow-links]
                   Display DWARF info in the file
--ctf=SECTION            Display CTF info from SECTION
-t, --syms               Display the contents of the symbol table(s)
-T, --dynamic-syms       Display the contents of the dynamic symbol table
-r, --reloc              Display the relocation entries in the file
-R, --dynamic-reloc      Display the dynamic relocation entries in the file
@<file>                  Read options from <file>
-v, --version            Display this program's version number
-i, --info               List object formats and architectures supported
-H, --help               Display this information
```

4.3.1 -h option

- **It displays different sections of the object file**

vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump -h main.o

```
main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000520  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000001  00000000  00000000  00000554  2**0
                  CONTENTS, ALLOC, LOAD, DATA
```

```
2 .bss          00000004  00000000  00000000  00000558  2**2
                ALLOC
3 .rodata       000000c9  00000000  00000000  00000558  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .comment      0000005a  00000000  00000000  00000621  2**0
                CONTENTS, READONLY
5 .ARM.attributes 0000002e  00000000  00000000  0000067b  2**0
                CONTENTS, READONLY
```

- Here, we can see the file format as “elf32-littlearm”
- There are various sections of file
  - .text : it holds the code or instructions of the program
  - .data : it holds initialized data
  - .bss : it holds the uninitialized data
  - .rodata : it holds the readonly data of program
  - .comment , .ARM.attributes : these sections added by the compiler for us
- From these sections you can add your own user-defined sections

4.3.2 -d option

- - -disassemble : Display assembler contents of executable sections
- This option helps us to understand various assembly level instructions generated for different functions of our program

vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump -d main.o > main.log
vlab@HYVLAB8:~/Desktop/cross-compilation$ cat main.log

main.o: file format elf32-littlearm

Disassembly of section .text:

00000000 **<main>**:

```
  0:    b580            push    {r7, lr}
  2:    af00            add     r7, sp, #0
  4:    f7ff fffe       bl      278 <enable_processor_faults>
  8:    f7ff fffe       bl      0 <initialise_monitor_handles>
```

```
 c:   4809          ldr     r0, [pc, #36]     ; (34 <main+0x34>)
 e:   f7ff fffe     bl      154 <init_scheduler_stack>
12:   4809          ldr     r0, [pc, #36]     ; (38 <main+0x38>)
14:   f7ff fffe     bl      0 <puts>
18:   f7ff fffe     bl      15e <init_tasks_stack>
1c:   f7ff fffe     bl      0 <led_init_all>
20:   f44f 707a     mov.w   r0, #1000         ; 0x3e8
24:   f7ff fffe     bl      ec <init_systick_timer>
28:   f7ff fffe     bl      384 <switch_sp_to_psp>
2c:   f7ff fffe     bl      42 <task1_handler>
30:   e7fe      b.n 30 <main+0x30>
32:   bf00          nop
34:   2001ec00      .word   0x2001ec00
38:   0000000c      .word   0x0000000c
      .
      .
      .
      .
      .
```

4.3.3 -s option

- --full-contents Display the full contents of all sections requested

vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump -s main.o > sections_data.txt
vlab@HYVLAB8:~/Desktop/cross-compilation$ cat sections_data.txt

main.o: file format elf32-littlearm

Contents of section .text:
```
 0000 80b500af fff7feff fff7feff 0948fff7  .............H..
 0010 feff0948 fff7feff fff7feff fff7feff  ...H............
 0020 4ff47a70 fff7feff fff7feff fff7feff  O.zp............
 0030 fee700bf 00ec0120 0c000000 80b400af  ....... ........
 0040 fee780b5 00af0948 fff7feff 0c20fff7  .......H..... ..
 0050 feff4ff4 7a70fff7 feff0c20 fff7feff  ..O.zp..... ....
 0060 4ff47a70 fff7feff ede700bf 34000000  O.zp........4...
 0070 80b500af 0848fff7 feff0d20 fff7feff  .....H..... ....
 0080 4ff47a70 fff7feff 0d20fff7 feff4ff4  O.zp..... ....O.
 0090 7a70fff7 feffede7 48000000 80b500af  zp......H.......
 00a0 0748fff7 feff0f20 fff7feff fa20fff7  .H..... ..... ..
 00b0 feff0f20 fff7feff fa20fff7 feffefe7  ... ..... ......
 00c0 5c000000 80b500af 0748fff7 feff0e20  \........H.....
```

```
00d0 fff7feff 7d20fff7 feff0e20 fff7feff  ....} ..... ....
00e0 7d20fff7 feffefe7 70000000 80b487b0  } ......p.......
00f0 00af7860 144b7b61 144b3b61 144a7b68  ..x`.K{a.K;a.J{h
0100 b2fbf3f3 013bfb60 7b690022 1a607b69  .....;.`{i.".`{i
0110 1a68fb68 1a437b69 1a603b69 1b6843f0  .h.h.C{i.`;i.hC.
0120 02023b69 1a603b69 1b6843f0 04023b69  ..;i.`;i.hC...;i
0130 1a603b69 1b6843f0 01023b69 1a6000bf  .`;i.hC...;i.`..
0140 1c37bd46 80bc7047 14e000e0 10e000e0  .7.F..pG........
0150 0024f400 034683f3 08887047 00bf80b4  .$...F....pG....
0160 85b000af 394b0022 1a72384b 00221a76  ....9K.".r8K.".v
0170 364b0022 83f82820 344b0022 83f83820  6K."..( 4K."..8
0180 324b0022 83f84820 304b314a 1a602f4b  2K."..H 0K1J.`/K
0190 304a1a61 2d4b304a 1a622c4b 2f4a1a63  0J.a-K0J.b,K/J.c
01a0 2a4b2f4a 1a64294b 2e4ada60 274b2e4a  *K/J.d)K.J.`'K.J
01b0 da61264b 2d4ada62 244b2d4a da63234b  .a&K-J.b$K-J.c#K
01c0 2c4ada64 0023bb60 37e0204a bb681b01  ,J.d.#.`7. J.h..
01d0 13441b68 fb60fb68 043bfb60 fb684ff0  .D.h.`.h.;.`.hO.
01e0 80721a60 fb68043b fb60184a bb681b01  .r.`.h.;.`.J.h..
01f0 13440c33 1b681a46 fb681a60 fb68043b  .D.3.h.F.h.`.h.;
0200 fb60fb68 6ff00202 1a600023 7b6008e0  .`.ho....`.#{`..
0210 fb68043b fb60fb68 00221a60 7b680133  .h.;.`.h.".`{h.3
0220 7b607b68 0c2bf3dd fa680849 bb681b01  {`{h.+...h.I.h..
0230 0b441a60 bb680133 bb60bb68 042bc4dd  .D.`.h.3.`.h.+..
0240 00bf00bf 1437bd46 80bc7047 00000000  .....7.F..pG....
0250 00f00120 00000220 00fc0120 00f80120  ... ... ... ...
0260 00f40120 00000000 00000000 00000000  ... ............
0270 00000000 00000000 80b483b0 00af0c4b  ...............K
0280 7b607b68 1b6843f4 80327b68 1a607b68  {`{h.hC..2{h.`{h
0290 1b6843f4 00327b68 1a607b68 1b6843f4  .hC..2{h.`{h.hC.
02a0 80227b68 1a6000bf 0c37bd46 80bc7047  ."{h.`...7.F..pG
02b0 24ed00e0 80b400af 044b1b78 044a1b01  $........K.x.J..
02c0 13441b68 1846bd46 80bc7047 00000000  .D.h.F.F..pG....
02d0 00000000 80b483b0 00af7860 054b1b78  ..........x`.K.x
02e0 054a1b01 13447a68 1a6000bf 0c37bd46  .J...Dzh.`...7.F
02f0 80bc7047 00000000 00000000 80b483b0  ..pG............
0300 00afff23 7b600023 3b6024e0 1a4b1b78  ...#{`.#;`$..K.x
0310 0133dab2 184b1a70 174b1a78 174ba3fb  .3...K.p.K.x.K..
0320 02139908 0b469b00 0b44d31a dab2124b  .....F...D.....K
0330 1a70114b 1b78124a 1b011344 08331b78  .p.K.x.J...D.3.x
0340 7b607b68 002b03d1 0b4b1b78 002b06d1  {`{h.+...K.x.+..
0350 3b680133 3b603b68 042bd7dd 00e000bf  ;h.3;`;h.+......
0360 7b68002b 02d0044b 00221a70 00bf0c37  {h.+...K.".p...7
0370 bd4680bc 704700bf 00000000 cdcccccc  .F..pG..........
0380 00000000 00b5fff7 feff80f3 09885df8  ..............].
```

```
 0390 04eb4ff0 020080f3 14887047 00bf80b4  ..O.......pG....
 03a0 83b000af 064b7b60 7b681b68 43f08052  .....K{`{h.hC..R
 03b0 7b681a60 00bf0c37 bd4680bc 704700bf  {h.`...7.F..pG..
 03c0 04ed00e0 80b582b0 00af7860 4ff00100  ..........x`O...
 03d0 80f31088 104b1b78 002b15d0 0f4b1a68  .....K.x.+...K.h
 03e0 0d4b1b78 18467b68 1a440d49 03010b44  .K.x.F{h.D.I...D
 03f0 04331a60 084b1b78 094a1b01 13440833  .3.`.K.x.J...D.3
 0400 ff221a70 fff7feff 4ff00000 80f31088  ."p....O.......
 0410 00bf0837 bd4680bd 00000000 00000000  ...7.F..........
 0420 00000000 eff30980 20e9f00f 00b5fff7  ........ .......
 0430 feffffff7 feffffff7 feffb0e8 f00f80f3  ................
 0440 09885df8 04eb7047 00bf80b4 00af044b  ..]...pG.......K
 0450 1b680133 024a1360 00bfbd46 80bc7047  .h.3.J.`...F..pG
 0460 00000000 80b483b0 00af0123 7b601be0  ...........#{`..
 0470 124a7b68 1b011344 08331b78 002b10d0  .J{h...D.3.x.+..
 0480 0e4a7b68 1b011344 04331a68 0c4b1b68  .J{h...D.3.h.K.h
 0490 9a4206d1 094a7b68 1b011344 08330022  .B...J{h...D.3."
 04a0 1a707b68 01337b60 7b68042b e0dd00bf  .p{h.3{`{h.+....
 04b0 00bf0c37 bd4680bc 704700bf 00000000  ...7.F..pG......
 04c0 00000000 80b582b0 00af084b 7b60fff7  ...........K{`..
 04d0 feffffff7 feff7b68 1b6843f0 80527b68  ......{h.hC..R{h
 04e0 1a6000bf 0837bd46 80bd00bf 04ed00e0  .`...7.F........
 04f0 80b500af 0148fff7 fefffee7 84000000  .....H..........
 0500 80b500af 0148fff7 fefffee7 9c000000  .....H..........
 0510 80b500af 0148fff7 fefffee7 b4000000  .....H..........
```

Contents of section .data:

```
 0000 01                                   .
```

Contents of section .rodata:

```
 0000 64000000 64000000 64000000 496d706c  d...d...d...Impl
 0010 656d656e 74617469 6f6e206f 66207369  ementation of si
 0020 6d706c65 20746173 6b207363 68656475  mple task schedu
 0030 6c657200 5461736b 31206973 20657865  ler.Task1 is exe
 0040 63757469 6e670000 5461736b 32206973  cuting..Task2 is
 0050 20657865 63757469 6e670000 5461736b   executing..Task
 0060 33206973 20657865 63757469 6e670000  3 is executing..
 0070 5461736b 34206973 20657865 63757469  Task4 is executi
 0080 6e670000 45786365 7074696f 6e203a20  ng..Exception :
 0090 48617264 6661756c 74000000 45786365  Hardfault...Exce
 00a0 7074696f 6e203a20 4d656d4d 616e6167  ption : MemManag
 00b0 65000000 45786365 7074696f 6e203a20  e...Exception :
 00c0 42757346 61756c74 00                 BusFault.
```

Contents of section .comment:

```
 0000 00474343 3a202831 353a392d 32303139  .GCC: (15:9-2019
 0010 2d71342d 30756275 6e747531 2920392e  -q4-0ubuntu1) 9.
```

```
 0020 322e3120 32303139 31303235 20287265  2.1 20191025 (re
 0030 6c656173 6529205b 41524d2f 61726d2d  lease) [ARM/arm-
 0040 392d6272 616e6368 20726576 6973696f  9-branch revisio
 0050 6e203237 37353939 5d00               n 277599].
Contents of section .ARM.attributes:
 0000 412d0000 00616561 62690001 23000000  A-...aeabi..#...
 0010 0537452d 4d00060d 074d0902 12041401  .7E-M....M......
 0020 15011703 18011901 1a011e06 2201      ..........".
```

4.3.4 -D option

- --disassemble-all Display assembler contents of all sections

vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump -D main.o

main.o: file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:

```
  0:   b580            push   {r7, lr}
  2:   af00            add    r7, sp, #0
  4:   f7ff fffe       bl     278 <enable_processor_faults>
  8:   f7ff fffe       bl     0 <initialise_monitor_handles>
  c:   4809            ldr    r0, [pc, #36]   ; (34 <main+0x34>)
  e:   f7ff fffe       bl     154 <init_scheduler_stack>
 12:   4809            ldr    r0, [pc, #36]   ; (38 <main+0x38>)
 14:   f7ff fffe       bl     0 <puts>
 18:   f7ff fffe       bl     15e <init_tasks_stack>
 1c:   f7ff fffe       bl     0 <led_init_all>
 20:   f44f 707a       mov.w  r0, #1000       ; 0x3e8
 24:   f7ff fffe       bl     ec <init_systick_timer>
 28:   f7ff fffe       bl     384 <switch_sp_to_psp>
 2c:   f7ff fffe       bl     42 <task1_handler>
 30:   e7fe      b.n    30 <main+0x30>
 32:   bf00            nop
 34:   2001ec00        andcs  lr, r1, r0, lsl #24
 38:   0000000c        andeq  r0, r0, ip
```

.
.
.
.

Disassembly of section .data:

00000000 <current_task>:
  0: Address 0x0000000000000000 is out of bounds.

Disassembly of section .bss:

00000000 <g_tick_count>:
  0: 00000000 andeq r0, r0, r0

Disassembly of section .rodata:

00000000 <const_v_1>:
  0: 00000064 andeq r0, r0, r4, rrx

00000004 <const_v_2>:
  4: 00000064 andeq r0, r0, r4, rrx
  .
  .
  .
  .

4.4 Why is it called a relocatable object file?

From assembler contents of above command

00000000 <main>: //main function
0: b580 push {r7, lr}
//b580 is opcode placed at address 00000000 with offset = 0
2: af00 add r7, sp, #0
//af00 is opcode placed at address 00000000 with offset = 2

- 00000000 is not a absolute address, you should mention the appropriate address here , and you will do that by using linker script because in our microcontroller the code space starts from 0x08000000 not from 0x00000000, thats why it is called relocatable sections
- You have to relocate these opcodes to some other addresses as per your microcontroller address map

Now, modify the MakeFile to add targets and dependencies

vlab@HYVLAB8:~/Desktop/cross-compilation$ cat led.log

led.o: file format elf32-littlearm

Disassembly of section .text:

00000000 <delay>: //here the start address is 0x00000000

```
 0:   b480            push   {r7}
 2:   b085            sub    sp, #20
 4:   af00            add    r7, sp, #0
 6:   6078            str    r0, [r7, #4]
 8:   2300            movs   r3, #0
 a:   60fb            str    r3, [r7, #12]
 c:   e002            b.n    14 <delay+0x14>
 e:   68fb            ldr    r3, [r7, #12]
10:   3301            adds   r3, #1
12:   60fb            str    r3, [r7, #12]
14:   68fa            ldr    r2, [r7, #12]
16:   687b            ldr    r3, [r7, #4]
18:   429a            cmp    r2, r3
1a:   d3f8            bcc.n  e <delay+0xe>
1c:   bf00            nop
1e:   bf00            nop
20:   3714            adds   r7, #20
22:   46bd            mov    sp, r7
24:   bc80            pop    {r7}
26:   4770            bx     lr
```

.
.
.

- Here we can observe that the .text section of both main.o and led.o located at 0x00000000, that means there is a conflict then the opcode of both these files will go into the same address thats not possible, so we must relocate them and we will do that by using linker script and by assigning the relocatable address to these sections
- Relocatable address have to be decided based on your microcontroller or processor memory map
- In every object file we can see that the base address is selected as 0x00000000, from that the offset is counted

- Also the base address of each section is selected as zero, which is not possible so we must relocate it at some possible address, then only our program will run properly

Disassembly of section **.data:**

00000000 <current_task>:
0: Address 0x0000000000000000 is out of bounds.

Disassembly of section **.bss:**

00000000 <g_tick_count>:
0: 00000000 andeq r0, r0, r0

Disassembly of section **.rodata:**

00000000 <const_v_1>:
0: 00000064 andeq r0, r0, r4, rrx

5. Startup File

5.1 Importance of start-up file

- The start-up file i responsible for setting up the right environment for the main user code to run
- Code written in the 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 (example : vector table, the way you access the stack pointer are processor specific)
- In some cases startup code may have to turn on some coprocessors like FPU(floating point unit).For example if main code is doing some floating point operations then FPU must be turned on prior to that otherwise there could be exception in the program.So, to enable any special peripheral or target you can do it in startup code
- Startup code take care of vector table placement in code memory as required by the ARM cortex MX processor (initial code memory addresses have to be filled with vector table)
- Startup code may also take care of stack reinitialization (means you can alter the stack placement)
- Startup code is responsible of .data,.bss section initialization in main memory by properly copying data section from flash to SRAM and by properly utilizing BSS section in the SRAM

Writing a startup file

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

- Go to reference manual of the STM32F401RE and view the vector table
- The starting address of vector table is 0x00000000 is reserved for MSP
- The address after 4 bytes is reset handler address 0x00000004

Table 38. Vector table for STM32F401xB/CSTM32F401xD/E

Position	Priority	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004
	-2	fixed	NMI	Non maskable interrupt, Clock Security System	0x0000 0008
	-1	fixed	HardFault	All class of fault	0x0000 000C
	0	settable	MemManage	Memory management	0x0000 0010
	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
	-	-	-	Reserved	0x0000 001C - 0x0000 002B
	3	settable	SVCall	System Service call via SWI instruction	0x0000 002C
	4	settable	Debug Monitor	Debug Monitor	0x0000 0030
		-	-	Reserved	0x0000 0034
	5	settable	PendSV	Pendable request for system service	0x0000 0038
	6	settable	Systick	System tick timer	0x0000 003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040

1	8	settable	EXTI16 / PVD	EXTI Line 16 interrupt / PVD through EXTI line detection interrupt	0x0000 0044
2	9	settable	EXTI21 / TAMP_STAMP	EXTI Line 21 interrupt / Tamper and TimeStamp interrupts through the EXTI line	0x0000 0048
3	10	settable	EXTI22 / RTC_WKUP	EXTI Line 22 interrupt / RTC Wakeup interrupt through the EXTI line	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11	18	settable	DMA1_Stream0	DMA1 Stream0 global interrupt	0x0000 006C
12	19	settable	DMA1_Stream1	DMA1 Stream1 global interrupt	0x0000 0070
13	20	settable	DMA1_Stream2	DMA1 Stream2 global interrupt	0x0000 0074
14	21	settable	DMA1_Stream3	DMA1 Stream3 global interrupt	0x0000 0078
15	22	settable	DMA1_Stream4	DMA1 Stream4 global interrupt	0x0000 007C
16	23	settable	DMA1_Stream5	DMA1 Stream5 global interrupt	0x0000 0080
17	24	settable	DMA1_Stream6	DMA1 Stream6 global interrupt	0x0000 0084
18	25	settable	ADC	ADC1 global interrupts	0x0000 0088
23	30	settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C
24	31	settable	TIM1_BRK_TIM9	TIM1 Break interrupt and TIM9 global interrupt	0x0000 00A0
25	32	settable	TIM1_UP_TIM10	TIM1 Update interrupt and TIM10 global interrupt	0x0000 00A4
26	33	settable	TIM1_TRG_COM_TIM11	TIM1 Trigger and Commutation interrupts and TIM11 global interrupt	0x0000 00A8
27	34	settable	TIM1_CC	TIM1 Capture Compare interrupt	0x0000 00AC
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
30	37	settable	TIM4	TIM4 global interrupt	0x0000 00B8

31	38	settable	I2C1_EV	I^2C1 event interrupt	0x0000 00BC
32	39	settable	I2C1_ER	I^2C1 error interrupt	0x0000 00C0
33	40	settable	I2C2_EV	I^2C2 event interrupt	0x0000 00C4
34	41	settable	I2C2_ER	I^2C2 error interrupt	0x0000 00C8
35	42	settable	SPI1	SPI1 global interrupt	0x0000 00CC
36	43	settable	SPI2	SPI2 global interrupt	0x0000 00D0
37	44	settable	USART1	USART1 global interrupt	0x0000 00D4
38	45	settable	USART2	USART2 global interrupt	0x0000 00D8
40	47	settable	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0
41	48	settable	EXTI17 / RTC_Alarm	EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt	0x0000 00E4
42	49	settable	EXTI18 / OTG_FS_WKUP	EXTI Line 18 interrupt / USB On-The-Go FS Wakeup through EXTI line interrupt	0x0000 00E8
47	54	settable	DMA1_Stream7	DMA1 Stream7 global interrupt	0x0000 00FC
49	56	settable	SDIO	SDIO global interrupt	0x0000 0104
50	57	settable	TIM5	TIM5 global interrupt	0x0000 0108
51	58	settable	SPI3	SPI3 global interrupt	0x0000 010C
56	63	settable	DMA2_Stream0	DMA2 Stream0 global interrupt	0x0000 0120
57	64	settable	DMA2_Stream1	DMA2 Stream1 global interrupt	0x0000 0124
58	65	settable	DMA2_Stream2	DMA2 Stream2 global interrupt	0x0000 0128
59	66	settable	DMA2_Stream3	DMA2 Stream3 global interrupt	0x0000 012C
60	67	settable	DMA2_Stream4	DMA2 Stream4 global interrupt	0x0000 0130
67	74	settable	OTG_FS	USB On The Go FS global interrupt	0x0000 014C
68	75	settable	DMA2_Stream5	DMA2 Stream5 global interrupt	0x0000 0150
69	76	settable	DMA2_Stream6	DMA2 Stream6 global interrupt	0x0000 0154
70	77	settable	DMA2_Stream7	DMA2 Stream7 global interrupt	0x0000 0158
71	78	settable	USART6	USART6 global interrupt	0x0000 015C
72	79	settable	I2C3_EV	I^2C3 event interrupt	0x0000 0160
73	80	settable	I2C3_ER	I^2C3 error interrupt	0x0000 0164
81	88	Settable	FPU	FPU global interrupt	0x0000 0184
84	91	settable	SPI4	SPI 4 global interrupt	0x0000 0190

- The initial 15 memory locations should carry addresses of various exception handlers of the processor, after that the first IRQ handler i.e., WWDG (window watchdog interrupt) and it goes all the way to 84
- 0 to 84 are IRQs that means total 85 IRQs

5.2 Creating a vector table

- Creating 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

5.3 Defining reset handler in startup code (stm32_startup.c)

- Create a stm32_startup.c file and write below code

```
#include<stdint.h>

#define SRAM_START  0x20000000U
#define SRAM_SIZE  (96*1024) //96KB
#define SRAM_END   ((SRAM_START)+(SRAM_SIZE))

#define STACK_START      SRAM_END

void Reset_handler(void);

uint32_t vectors[] = {
      STACK_START,
      (uint32_t)&Reset_handler,



};

void Reset_handler(void)
{

}
```

- Make changes in MakeFile to include startup file also in compilation and also to clear the generated .o and .elf files using make clean

```
CC = arm-none-eabi-gcc
MACH=cortex-m4
CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -O0

all:main.o led.o stm32_startup.o

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

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

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

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

vlab@HYVLAB8:~/Desktop/cross-compilation$ make clean
rm -rf *.o *.elf
vlab@HYVLAB8:~/Desktop/cross-compilation$ ls
led.c led.log main.h Makefile stm32_startup.c
led.h main.c main.log sections_data.txt
vlab@HYVLAB8:~/Desktop/cross-compilation$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 main.c -o main.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 led.c -o led.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 stm32_startup.c -o stm32_startup.o
vlab@HYVLAB8:~/Desktop/cross-compilation$ ls
led.c led.log main.c main.log Makefile stm32_startup.c
led.h led.o main.h main.o sections_data.txt stm32_startup.o

- All the initialized data is kept in the .data section. Likewise in code the vectors array created for vector table is stored in .data section by default but we must keep in user defined section

```
uint32_t vectors[] = {
	STACK_START,
	(uint32_t)&Reset_handler,

};
```

- To keep it in user defined section we must use the compiler attributes like
    - **section("section-name") :** used to push the data in user defined section other than the .data and .bss section
    - **Syntax:**
        - **`struct duart a __attribute__((section ("DUART_A"))) = {0};`**
        - A variable named "a" of type struct uart is placed in auser defined section called DUART_A
- Make the required changes in startup file to add new user defined section for vector table

```
uint32_t vectors[] = {
	STACK_START,
	(uint32_t)&Reset_handler,



};__attribute__((section(".isr_vector")))
```

(or)

```
uint32_t vectors[] __attribute__((section(".isr_vector")))= {
      STACK_START,
      (uint32_t)&Reset_handler,



};
```

vlab@HYVLAB8:~/Desktop/cross-compilation$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 stm32_startup.c -o stm32_startup.o
vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump -h stm32_startup.o

stm32_startup.o: file format elf32-littlearm

Sections:
Idx Name Size VMA LMA File off Algn
0 .text 0000000c 00000000 00000000 00000034 2**1
CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .data 00000000 00000000 00000000 00000040 2**0
CONTENTS, ALLOC, LOAD, DATA
2 .bss 00000000 00000000 00000000 00000040 2**0
ALLOC
3 .isr_vector 00000008 00000000 00000000 00000040 2**2
CONTENTS, ALLOC, LOAD, RELOC, DATA
4 .comment 0000005a 00000000 00000000 00000048 2**0
CONTENTS, READONLY
5 .ARM.attributes 0000002e 00000000 00000000 000000a2 2**0
CONTENTS, READONLY

- By default the Reset_Handler function is placed in .text section as we see size 0x0000000c
- But we may keep it in some other random_section

```
void Reset_handler(void)__attribute__((section(".random_section")));
```

vlab@HYVLAB8:~/Desktop/cross-compilation$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 stm32_startup.c -o stm32_startup.o
vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump -h stm32_startup.o

stm32_startup.o: file format elf32-littlearm

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 0 .text         00000000  00000000  00000000  00000034  2**1
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .data         00000000  00000000  00000000  00000034  2**0
                 CONTENTS, ALLOC, LOAD, DATA
 2 .bss          00000000  00000000  00000000  00000034  2**0
                 ALLOC
 3 .random_section 0000000c  00000000  00000000  00000034  2**1
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .isr_vector   00000008  00000000  00000000  00000040  2**2
                 CONTENTS, ALLOC, LOAD, RELOC, DATA
 5 .comment      0000005a  00000000  00000000  00000048  2**0
                 CONTENTS, READONLY
 6 .ARM.attributes 0000002e  00000000  00000000  000000a2  2**0
                 CONTENTS, READONLY
```

vlab@HYVLAB8:~/Desktop/cross-compilation$ arm-none-eabi-objdump -d stm32_startup.o

stm32_startup.o: file format elf32-littlearm

Disassembly of section .random_section:

```
00000000 <Reset_handler>:
   0:   b480        push    {r7}
   2:   af00        add     r7, sp, #0
   4:   bf00        nop
   6:   46bd        mov     sp, r7
   8:   bc80        pop     {r7}
   a:   4770        bx      lr
```

- Now we can observe that text section size is reduced to zero and .random_section is created with size of 0x0000000c

Define default interrupt handler

- We have got 85 interrupts . But there is no need to write handlers for all the exceptions.
- Lets create a single default handler for all the exceptions and allow programmer to implement required handlers as per application requirements
- Make default handler as an alias all the exception handlers for that we have to use the GCC attributes “weak” and “alias”
    - **Weak :** programmer can override already defined weak function (dummy) with same function name
    - **Alias :** programmer can give a alias name for a function
    - Syntax:
        - Extern int __attribute__((alias(“var_target”))) var_alias;

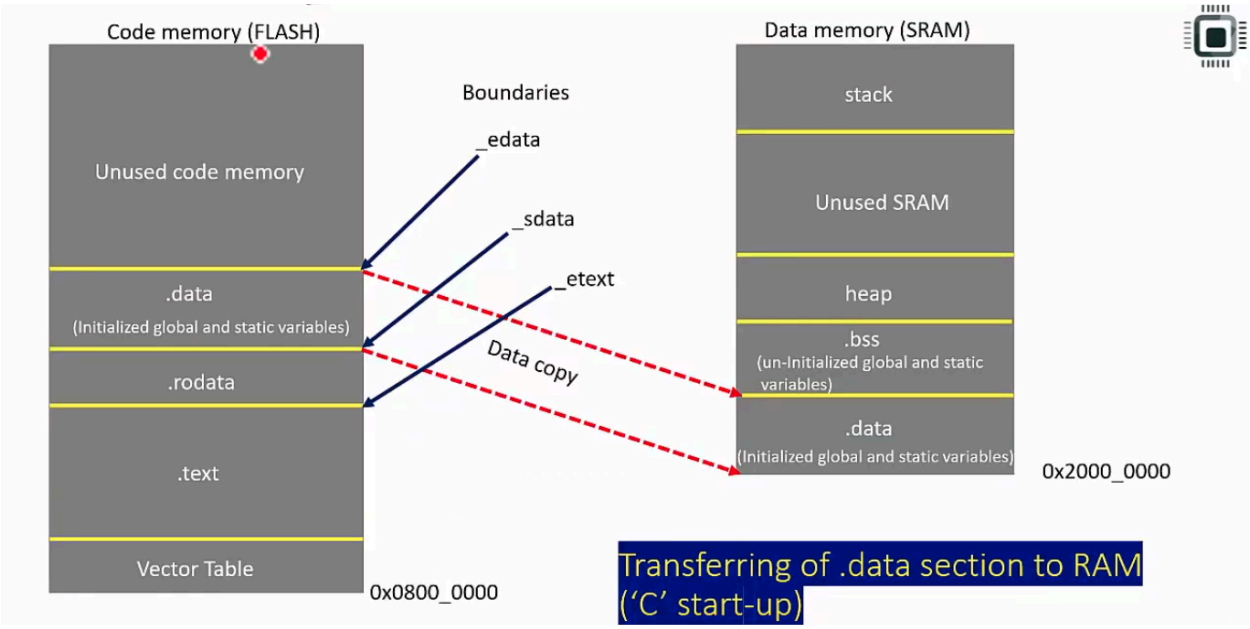
```
#include<stdint.h>
```

```
#define SRAM_START  0x20000000U
#define SRAM_SIZE   (96*1024) //96KB
#define SRAM_END    ((SRAM_START)+(SRAM_SIZE))

#define STACK_START SRAM_END

void Reset_handler(void)__attribute__((section(".random_section")));
void NMI_Handler(void) __attribute__((weak,alias("default_Handler")));
Void HardFault_Handler(void) __attribute__((weak,alias("default_Handler")));


uint32_t vectors[] __attribute__((section(".isr_vector")))= {
       STACK_START,
       (uint32_t)&Reset_handler,
       (uint32_t)&NMI_Handler,
       (uint32_t)&HardFault_Handler,




};

void default_Handler(void)
{
       while(1);
}

void Reset_handler(void)
{

}
```

- Here “default_Handler” is an alias function name for “NMI_Handler”
- In the vector table array, address of “default_Handler” function will be stored
- So, when NMI exception triggers, “default_Handler” will be executed
- Similarly define HardFault_handler by aliasing it with the “default_Handler”
- Now, we used weak attribute,it allow programmer to override this function with same function name in main application.There programmer can implement real implementation of handling that exception
- If we use the NMI_Handler function in main() then it will not call aliased function named “default_Handler” as it is marked as weak, it will use the override function that is defined in the main()
- Similarly define all the interrupt handlers aliasing with “Default_handler”

6. Linker script

- It 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 linker script
- Linker script also includes the code and data memory address and size information
- Linker scripts are written using GNU linker command language
- GNU linker script has the file extension of .ld
- We must supply linker script at the linking phase to the linker using -T option



6.1 Linker script commands

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

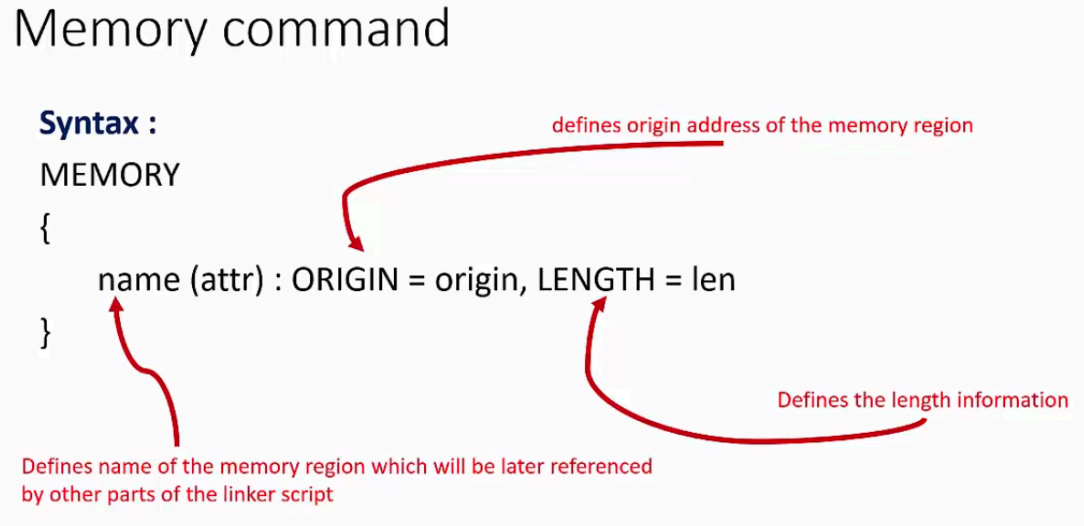
6.1.1 ENTRY command

- This command is used to set the “Entry point address” information 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)

6.1.2 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 the body of the MEMORY command we can write many statements having three parts
  - Label : name(attr) ,Name with attribute (optional)
  - ORIGIN : start address of the memory region
  - LENGTH :length of memory address

```
MEMORY
{
    FLASH(rx):ORIGIN =0x08000000,LENGTH =512K
    SRAM(rwx):ORIGIN =0x20000000,LENGTH =96K
}
```



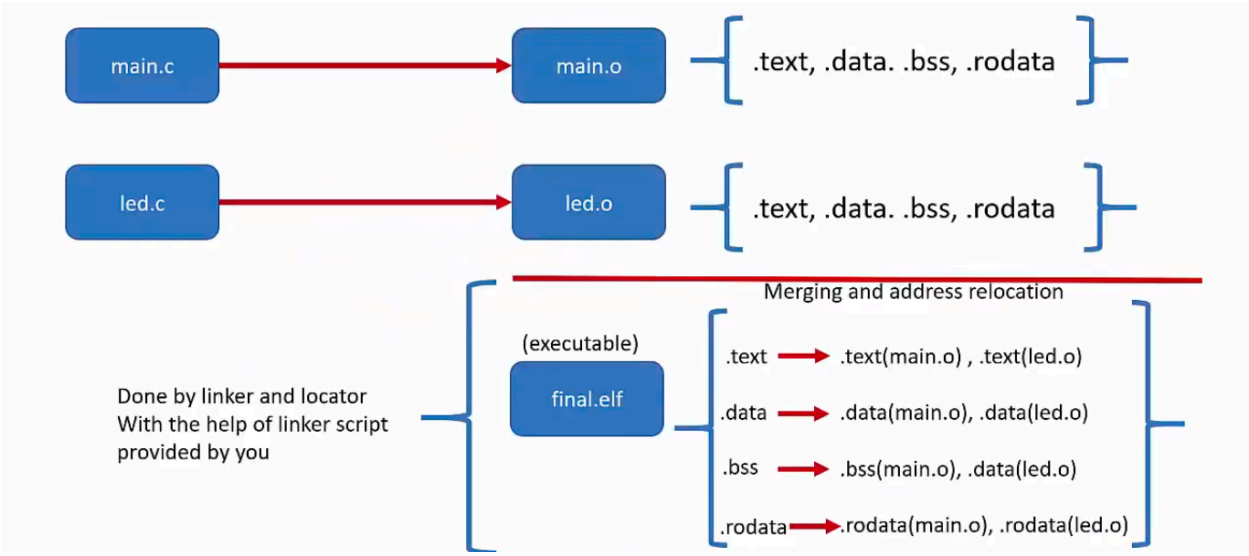
- **attr :** defines the attributes list of the memory region , valid attribute lists must be made up of the characters "ALIRWX" that match section attributes. We can also combine two or more attributes
    - **R :** read only section
    - **W :** read and write section
    - **X :** section containing executable code
    - **A**: Accessed (indicates if the memory region has been accessed)
    - **L**: Locked (indicates if the memory region is locked)
    - **I**: Instruction Cacheable (indicates if the memory region is cacheable for instructions)

6.1.3 SECTIONS command

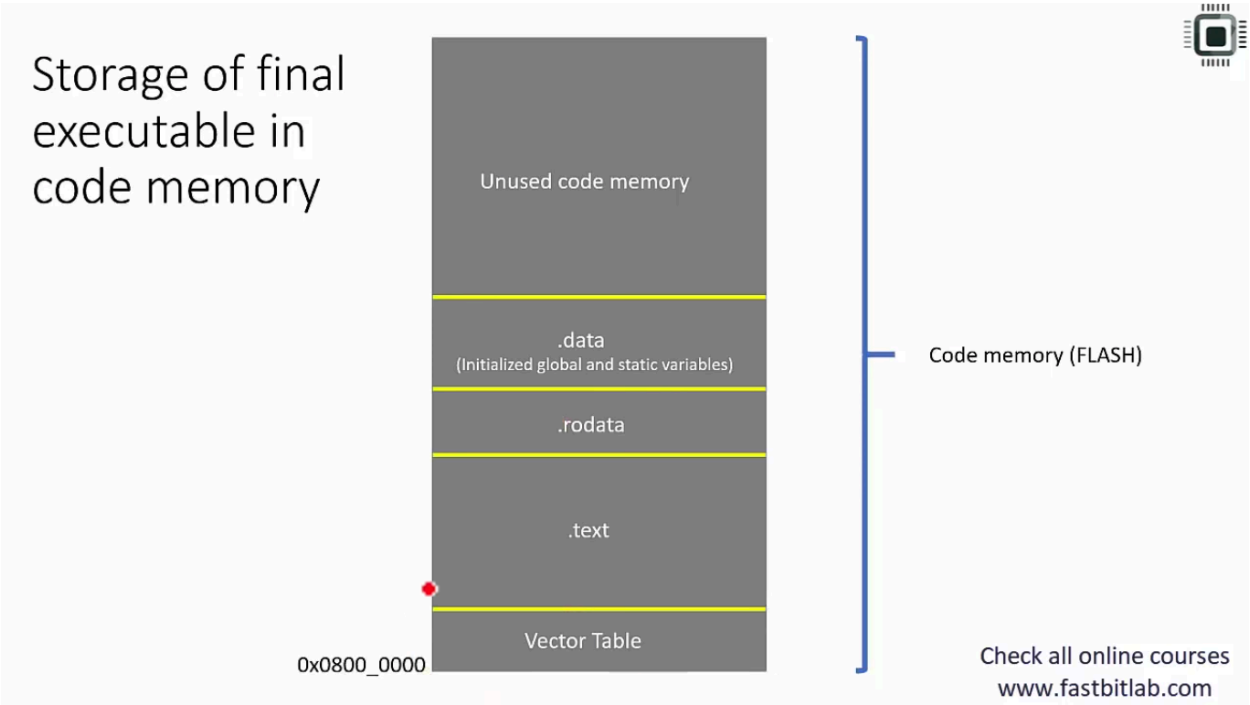
- SECTIONS command is used to create different output sections in the final elf executable generated
- Important command by which you can instruct the linker how to merge the input sections to yield an output section
- This command also controls the order in which different output sections appear in 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)

}
```

- .text , .data : it is a label of the output section, this label can be anything it depends on you can give any name you want but it must follow some standard naming conventions
- This body of the output section is directed to some address that tells locator where to place this section in the memory.This is a combination of VMA(virtual memory address) and LMA(Load memory address)
- Once the linker sees .text section it generates the absolute addresses for the .text sectionand that address falls in vma i.e., FLASH
- Linker also generates the load addresses for .text section that addresses fall in lma that is mentioned after AT command
- If vma and lma are same no need to mention AT>lma, mention only vma which denotes vma and lma are same



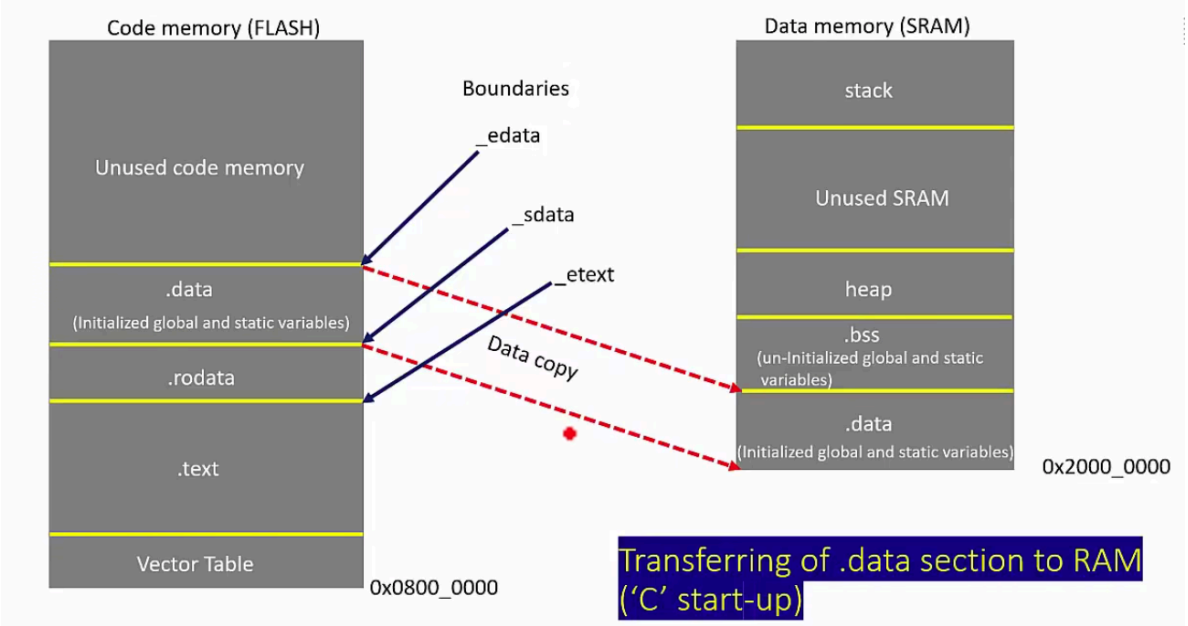
- The output section merges all other sections of individual input files like .text section merges all .isr_vector,.text and .rodata of all input files
- Attribute list of FLASH is 'rx' (readable and executable),because flash contains code that is not writable by user
- * is wild card character.It just says merge .text section of all input files



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

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

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


}
```

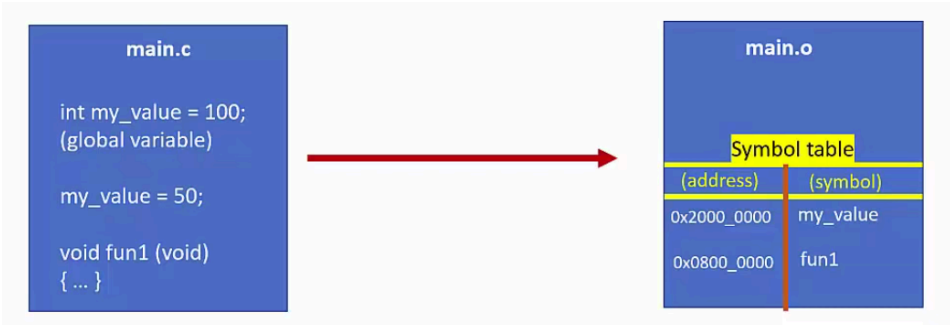
6.2 Location counter (.)

- 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 the 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
- Location counter incremented by the size of the output section



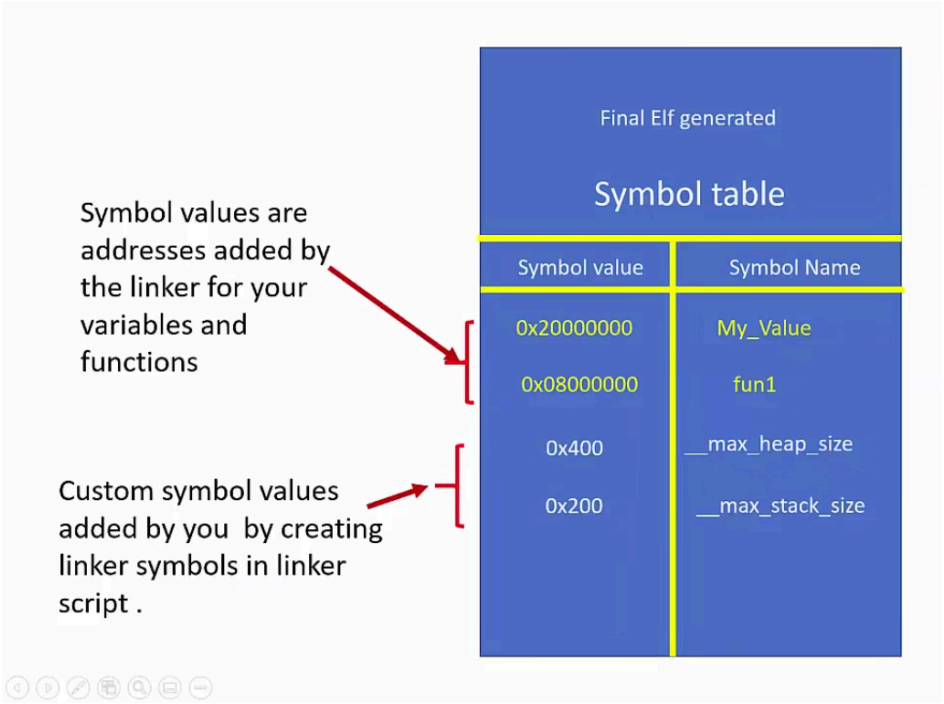
6.3 Linker Script Symbol

- 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



- Initially location counter is equal to the vma (that is assumed by the linker)
- Linker sees the symbols entry , it add these entries to the symbol table of final executable

NOTE: location counter always track VMA of the section in which it is being used not LMA



6.3.1 Example of writing symbols in linker script:

```
ENTRY (Reset_Handler)

MEMORY
{
        FLASH(rx):ORIGIN=0x08000000,LENGTH=512K
        SRAM(rwx):ORIGIN=0x20000000,LENGTH=96K
}


__max_heap_size = 0x400;
__max_stack_size = 0x200;

SECTIONS
{
        .text:
        {
                *(.isr_vector)
                *(.text)
                *(.rodata)
                end_of_text = .; /*store updated counter value in this symbol*/
```

```
        }> FLASH

        .data
        {
                start_of_data= 0x20000000;   /*assign a value to a symbol*/
                *(.data)
        }> SRAM AT>FLASH

        .bss
        {
                *(.bss)
        }> SRAM


}
```

6.3.2 Adding linker symbols to existing script

```
ENTRY(Reset_Handler)

MEMORY
{
   FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
   SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 96K
}

SECTIONS
{
   .isr_vector :
   {
      KEEP(*(.isr_vector))  /* Keep the interrupt vector table */
   } > FLASH

   .text :
   {
      *(.text)             /* All text sections from .o files */
      *(.rodata)            /* Read-only data */
      _etext = .;           /* End of text */
   } > FLASH

   .data :
   {
      _sdata = .;           /* Start of .data section */
      *(.data)             /* All data sections */
      _edata = .;           /* End of .data section */
   } > SRAM AT> FLASH
```

```
    .bss :
    {
        _sbss = .;          /* Start of .bss section */
        *(.bss)             /* All bss sections */
        _ebss = .;          /* End of .bss section */
    } > SRAM
}
```

7. Linking and analyzing memory map file

```
vlab@HYVLAB7:~/lochu$ make clean
rm -rf *.o *.elf
vlab@HYVLAB7:~/lochu$ ls
led.c  led.h  main.c  main.h  Makefile  stm32_ls.ld  stm32_startup.c
vlab@HYVLAB7:~/lochu$ make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 main.c -o main.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 led.c -o led.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 stm32_startup.c -o
stm32_startup.o
vlab@HYVLAB7:~/lochu$ arm-none-eabi-gcc -nostdlib -T stm32_ls.ld *.o -o final.elf
vlab@HYVLAB7:~/lochu$ ls
final.elf  led.h  main.c  main.o    stm32_ls.ld      stm32_startup.o
led.c      led.o  main.h  Makefile  stm32_startup.c
vlab@HYVLAB7:~/lochu$ arm-none-eabi-objdump -h final.elf

final.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .isr_vector   00000010  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .text         000005ed  08000010  08000010  00010010  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .random_section 00000006  080005fe  080005fe  000105fe  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

```
 3 .data         00000001  20000000  08000604  00020000  2**0
                 CONTENTS, ALLOC, LOAD, DATA
 4 .bss          00000054  20000004  08000605  00020004  2**2
                 ALLOC
 5 .comment      00000059  00000000  00000000  00020001  2**0
                 CONTENTS, READONLY
 6 .ARM.attributes 0000002e  00000000  00000000  0002005a  2**0
                 CONTENTS, READONLY
```

7.1 Automating the creation of .elf file

You can now modify the Makefile to automate the creation of final .elf file

```
CC = arm-none-eabi-gcc
MACH=cortex-m4
CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -Wall -O0
LDFLAGS = -nostdlib -T stm32_ls.ld
all:main.o led.o stm32_startup.o final.elf
main.o:main.c
	$(CC) $(CFLAGS) $^ -o $@
led.o:led.c
	$(CC) $(CFLAGS) $^ -o $@
stm32_startup.o:stm32_startup.c
	$(CC) $(CFLAGS) $^ -o $@
final.elf: main.o led.o stm32_startup.o
	$(CC) $(LDFLAGS)  $^ -o $@
clean:
	rm -rf *.o *.elf
```

```
vlab@HYVLAB7:~/lochu$ make clean
rm -rf *.o *.elf
vlab@HYVLAB7:~/lochu$ make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 main.c -o main.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 led.c -o led.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 stm32_startup.c -o
stm32_startup.o
arm-none-eabi-gcc -nostdlib -T stm32_ls.ld  main.o led.o stm32_startup.o -o final.elf
vlab@HYVLAB7:~/lochu$ ls
final.elf  led.h  main.c  main.o    stm32_ls.ld      stm32_startup.o
led.c      led.o  main.h  Makefile  stm32_startup.c
```

7.2 Memory Map file

- We can instruct the linker to create a special file called map file
- Through Map file we can analyze the various resource allocation and placement in the memory
- There is a command to instruct the linker to create the memory map file or simply we called it as map file
- We use the below linker argument in the used with linker command
    - -Map=final.map
- Sometimes this argument may not be recognized by the compiler used , thats why we must explicitly mention that this is a linker argument by using argument -Wl and you have to give comma
    - -Wl,-Map=final.map
- arm-none-eabi-gcc executable may also contain a linker driver
- The changed makefile is:

```
CC = arm-none-eabi-gcc
MACH=cortex-m4
CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -Wall -O0
LDFLAGS = -nostdlib -T stm32_ls.ld -Wl,-Map=final.map


all:main.o led.o stm32_startup.o final.elf

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

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

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

final.elf: main.o led.o stm32_startup.o
	$(CC) $(LDFLAGS)  $^ -o $@

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

vlab@HYVLAB7:~/lochu$ make clean
rm -rf *.o *.elf
vlab@HYVLAB7:~/lochu$ make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 main.c -o main.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 led.c -o led.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 stm32_startup.c -o stm32_startup.o
arm-none-eabi-gcc -nostdlib -T stm32_ls.ld -Wl,-Map=final.map main.o led.o stm32_startup.o -o final.elf

vlab@HYVLAB7:~/lochu$ ls
final.elf led.c led.o main.h Makefile stm32_startup.c
final.map led.h main.c main.o stm32_ls.ld stm32_startup.o

7.3 How to get the symbols of the application

- To get all the symbols of the application you are running use **"arm-none-eabi-nm"**
  vlab@HYVLAB7:~/lochu$ arm-none-eabi-nm final.elf
  080004dc T BusFault_Handler
  080005f4 T const_v_1
  080005f8 T const_v_2
  080005fc T const_V_3
  20000000 D current_task
  080005ec T default_Handler
  080004f0 T delay
  20000058 B _ebss
  20000004 D _edata
  08000258 T enable_processor_faults
  08000600 T _etext
  08000294 T get_psp_value
  20000004 B g_tick_count
  080004d0 T HardFault_Handler
  08000044 T idle_task
  080004e2 T initialise_monitor_handles
  08000134 T init_scheduler_stack
  080000ca T init_systick_timer
  0800013e T init_tasks_stack
  08000518 T led_init_all
  080005bc T led_off
  0800058c T led_on
  08000010 T main
  080004d6 T MemManage_Handler
  080005ec W NMI_Handler
  08000404 T PendSV_Handler
  08000600 T Reset_Handler
  080002b4 T save_psp_value
  20000004 B _sbss
  0800037e T schedule
  20000000 D _sdata
  08000364 T switch_sp_to_psp
  080004a4 T SysTick_Handler
  0800004a T task1_handler

```
0800006c T task2_handler
0800008e T task3_handler
080000ac T task4_handler
080003a4 T task_delay
08000444 T unblock_tasks
0800042a T update_global_tick_count
080002dc T update_next_task
20000008 B user_tasks
08000000 T vectors
```

- In a c program,if youjust write '_edata' then it menas you want to access the value from memory location (0x20000004) associated with that variable . These has no value stored in any memory location for this symbol '_edata'
- So you must use '&_edata',this will give you the corresponding symbol value (an address in this case) from symbol table that is 0x20000004

7.4 Definition of the Reset_Handler function

- Make the below changes in the startup file to add the definition of Reset_Handler

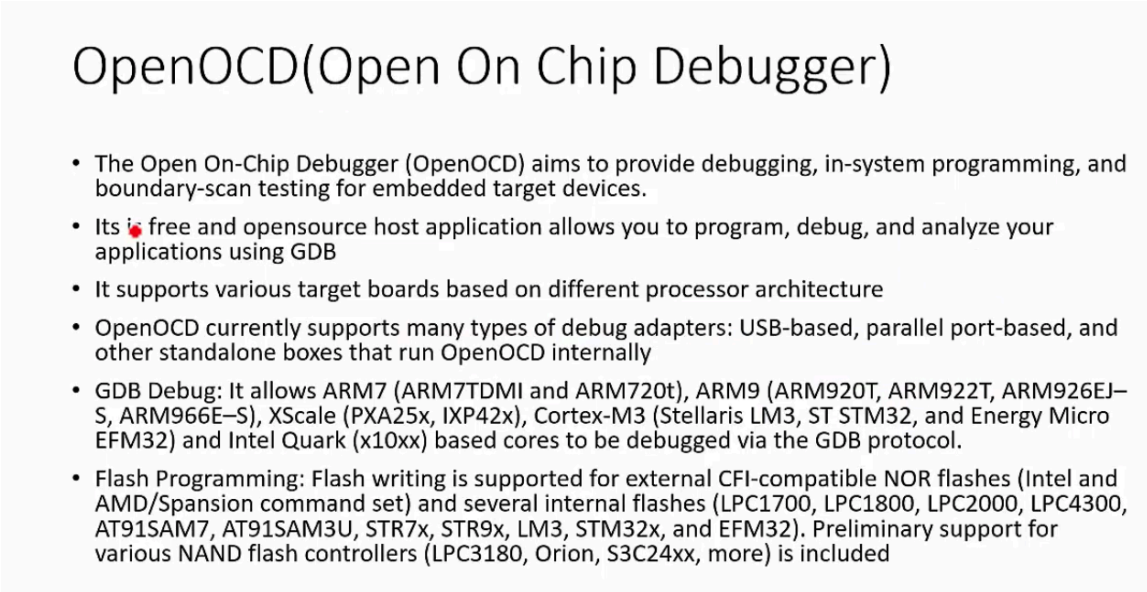
```
#include <stdint.h>

#define SRAM_START          0x20000000U
#define SRAM_SIZE  (96*1024) //96KB
#define SRAM_END   ((SRAM_START)+(SRAM_SIZE))

#define STACK_START        SRAM_END

extern uint32_t _etext;
extern uint32_t _sdata;
extern uint32_t _edata;

extern uint32_t _ebss;
extern uint32_t _sbss;


//prototype of main
int main(void);

void Reset_Handler(void)      __attribute__((section(".random_section")));
void NMI_Handler(void)              __attribute__((weak,alias("default_Handler")));
void HardFault_Handler(void)        __attribute__((weak,alias("default_Handler")));
```

```
uint32_t vectors[] __attribute__((section(".isr_vector"))) = {
        STACK_START,
        (uint32_t)&Reset_Handler,
        (uint32_t)&NMI_Handler,
        (uint32_t)&HardFault_Handler,


};

void default_Handler(void)
{
        while(1);
}

void Reset_Handler(void)
{
        //copy .data section to SRAM
        uint32_t size = (uint32_t)&_edata - (uint32_t)&_sdata;

        uint8_t *pDst = (uint8_t*)&_sdata;  //sram
        uint8_t *pSrc = (uint8_t*)&_etext;   //flash
        for(uint32_t i=0;i<size;i++)
        {
                *pDst++ = *pSrc++;
        }
        //init. the .bss section to zero in SRAM
        size = &_ebss - &_sbss;
        pDst = (uint8_t*)&_sbss;
        for(uint32_t i=0;i<size;i++)
        {
                *pDst++ = 0;
        }

        main();
}
```

8. Downloading and Debugging executable

- We have to download and debug the executable file (.elf) into the internal flash of the microcontroller and after that we can run our program on the development board
- How to download the executable , There are two important methods for it
    - One is you have connect the target board to PC via **“debug adapter(In-circuit programmer/debugger)”**
        - Programming through a debug adapter is also called as In-circuit programming and debugging
        - The debugger actually does the protocol conversion , it converts the host protocol to native target protocol
        - Example : SWD or JTAG debugger has JTAG based debug protocol that we call as target interface , because your target understands the debug protocol like SWD or JTAG and host understands interface such as USB / serial port
        - On the host side you have to run some application to talk to debug adapter , there by you can talk to target and send commands/.executables to target
        - One such application that we run on host machine is OpenOCD

8.1 OpenOCD



- We use OpenOCD mainly to achieve to goals : to download the executable into internal flash of microcontroller and debug the code using gdb

8.2 Programming Adapters

Programming adapters

- Programming adapters are used to get access to the debug interface of the target with native protocol signaling such as SWD or JTAG since HOST doesn't support such interfaces.
- It does protocol conversion. For example, commands and messages coming from host application in the form of USB packets will be converted to equivalent debug interface signaling (SWD or JTAG) and vice versa
- Mainly debug adapter helps you to download and debug the code
- Some advanced debug adapters will also help you to capture trace events such as on the fly instruction trace and profiling information

8.3 Some of the popular programming adapters

1. Segger JLINK EDU

SEGGER J-Link EDU - JTAG/SWD Debugger
Multiple CPUs supported—8051, PIC32, RX, ARM7/9/11, Cortex-M/R/A, RISC-V
Download speed up to 1 MByte/s
Debug Protocol : JTAG/SWD
Target Interface : 20-pin

2. KEIL ULINK-Pro

provides extended on-the-fly debug capabilities for Cortex-M devices. You can control the processor, set breakpoints, and read/write memory contents, all while the processor is running at full speed.
High-Speed data and instruction trace are streamed directly to your PC enabling you to analyze detailed program behavior.

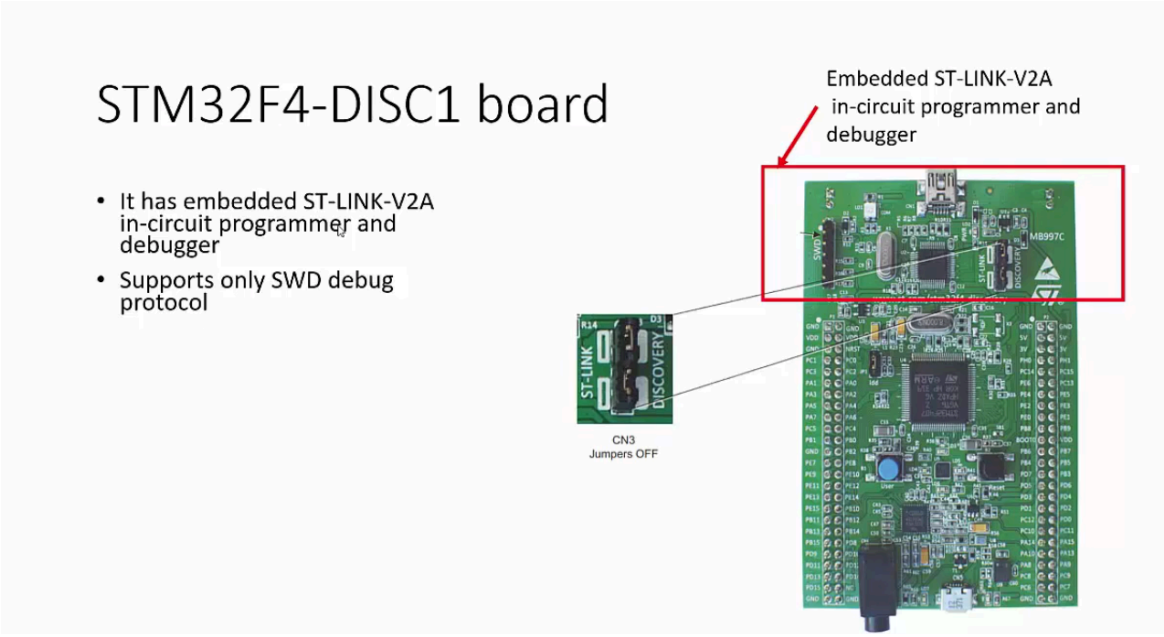
•Target Connectors10-pin (0.05") - Cortex Debug Connector
•20-pin (0.10") - ARM Standard JTAG Connector
•20-pin (0.05") - Cortex Debug+ETM Connector

3. ST-LINK/V2

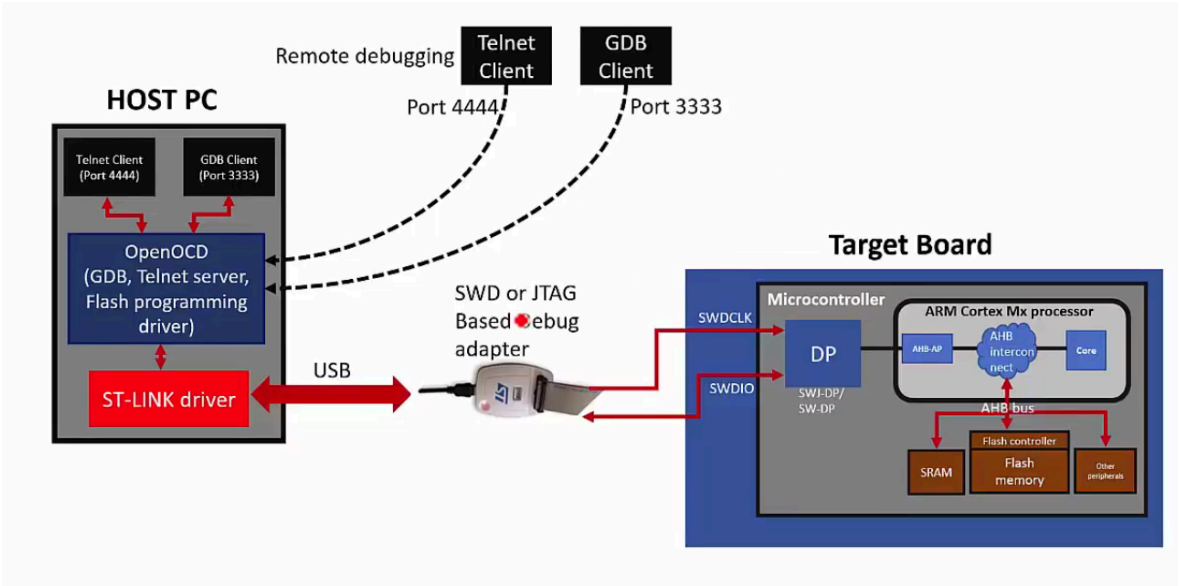
The ST-LINK/V2 is an in-circuit debugger and programmer for the STM8 and STM32 microcontroller families.

The single wire interface module (SWIM) and JTAG/serial wire debugging (SWD) interfaces are used to communicate with any STM8 or STM32 microcontroller located on an application board.

4. Inbuilt ST-LINK/V2 in disco boards



8.4 How your code gets downloaded into the internal flash?

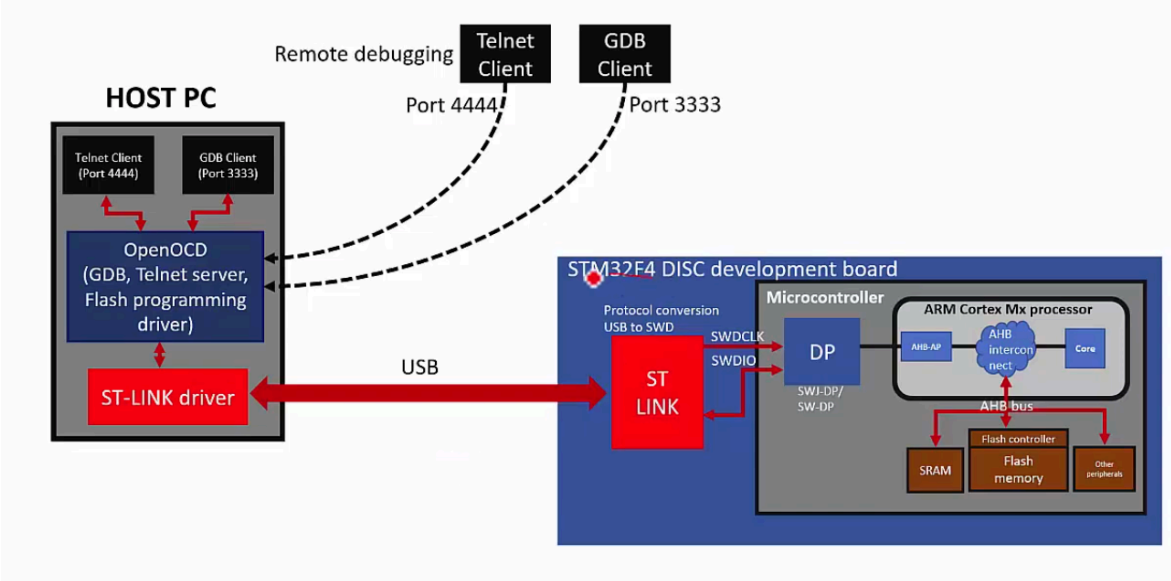


- First you will be using some host application like OpenOCD

- OpenOCD connected to the debug adapter over a driver like ST-LINK
- OpenOCD send USB packets to adapter
- At target interface you can use JTAG pins or SWD pins
- Suppose we use SWD protocol to program and debug your target board , then you would need only two pins SWDCLK and SWDIO(data line)
- SWDCLK is controlled by debug adapter that's why we call it as a master
- SWDIO is a bidirectional data line where commands are sent and responses are received by host application
- Microcontroller provides you two pins to connect the SWDCLK and SWDIO , as it require sonly two pins it is quite famous than JTAG which needs a four pins
- In microcontroller we have something called debug port or debug access port DP , this block knows how to talk to the processor
- By using DP we can access various bus interfaces of the processor like you can access AHB / APB bus interface
- You must have an access point in processor to talk to memory and to the core
- We have AHB-AP access point which helps you to get access to the AHB interconnect where it is connected to SRAM , flash memory and core
- AHB interconnect can control the core means you can reset the core
- If any binary is sent over OpenOCD , it would send the USB packets that will be converted SWD packets that carry your data , it comes all the way to the AHB interconnect to flash memory

8.5 Using STM32F4 disco board

- It has inbuilt ST link driver



9. Installation of OpenOCD

- Download the OpenOCD config files for different boards using below link:
  https://github.com/xpack-dev-tools/openocd-xpack/releases

- Follow below command for installation in ubuntu

```
$ git clone https://git.code.sf.net/p/openocd/code openocd-code
$ sudo apt-get install build-essential git libtool libudev-dev
pkg-config libftdi-dev libusb-1.0-0-dev autoconf automake
libncurses5-dev libreadline-dev python3 python3-pytest
$ cd openocd-code
$ ./bootstrap
$ ./configure --enable-stlink --enable-ftdi --enable-usb-blaster
$ make
$ sudo make install
$ openocd --version
```

Open On-Chip Debugger 0.12.0+dev-00730-gad2161361 (2024-09-26-17:32)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html

- Add the openocd command in the Makefile atmost below after clean section for the stm32f401re board

```
load:
	sudo openocd -f
	~/Desktop/xpack-openocd-0.12.0-4-linux-x64/xpack-openocd-0.12.0-4/openocd/scripts/interface/stlink.cfg -f
	~/Desktop/xpack-openocd-0.12.0-4-linux-x64/xpack-openocd-0.12.0-4/openocd/scripts/target/stm32f4x.cfg -c
	"adapter speed 1000"
```

- Again clean the make , remake and make the load also

```
$ make clean
$ make
$ make load
```

sudo openocd -f
~/Desktop/xpack-openocd-0.12.0-4-linux-x64/xpack-openocd-0.12.0-4/openocd/scripts/interface/stlink.cfg -f
~/Desktop/xpack-openocd-0.12.0-4-linux-x64/xpack-openocd-0.12.0-4/openocd/scripts/target/stm32f4x.cfg -c "adapter speed 1000"
Open On-Chip Debugger 0.12.0+dev-00730-gad2161361 (2024-09-26-17:32)
Licensed under GNU GPL v2

For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use 'transport select <transport>'.
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 1000 kHz
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : STLINK V2J45M30 (API v2) VID:PID 0483:374B
Info : Target voltage: 3.238345
Info : [stm32f4x.cpu] Cortex-M4 r0p1 processor detected
Info : [stm32f4x.cpu] target has 6 breakpoints, 4 watchpoints
Info : [stm32f4x.cpu] Examination succeed
Info : [stm32f4x.cpu] starting gdb server on 3333
Info : Listening on port 3333 for gdb connections

9.1 Opening OpenOCD using GDB client

- Enter **“arm-none-eabi-gdb”** in CLI and press enter

```
vlab@HYVLAB7:~/lochu$ arm-none-eabi-gdb
GNU gdb (Arm GNU Toolchain 13.3.Rel1 (Build arm-13.24))
14.2.90.20240526-git
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu
--target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
```

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

9.2 Gdb commands

9.2.1 target remote localhost:3333

- Enter command : **target remote localhost:3333** in gdb command line

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
warning: No executable has been specified and target does not
support
determining executable automatically.  Try using the "file"
command.
0x080004d4 in ?? ()
```

- After running that command in the info terminal you will find “accepting gdb connection” message

```
Open On-Chip Debugger 0.12.0+dev-00730-gad2161361
(2024-09-26-17:32)
Licensed under GNU GPL v2
For bug reports, read
     http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport
"hla_swd". To override use 'transport select <transport>'.
Info : The selected transport took over low-level target
control. The results might differ compared to plain JTAG/SWD
adapter speed: 1000 kHz
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : STLINK V2J45M30 (API v2) VID:PID 0483:374B
Info : Target voltage: 3.246224
Info : [stm32f4x.cpu] Cortex-M4 r0p1 processor detected
Info : [stm32f4x.cpu] target has 6 breakpoints, 4 watchpoints
Info : [stm32f4x.cpu] Examination succeed
Info : [stm32f4x.cpu] starting gdb server on 3333
```

```
Info : Listening on port 3333 for gdb connections
Info : accepting 'gdb' connection on tcp/3333
[stm32f4x.cpu] halted due to debug-request, current mode: Handler
HardFault
xPSR: 0x01000003 pc: 0x080004d4 msp: 0x2001eba4
Info : device id = 0x10016433
Info : flash size = 512 KiB
Info : flash size = 512 bytes
Warn : Prefer GDB command "target extended-remote :3333"
instead of "target remote :3333"
```

9.2.2 Monitor reset command

- Here reset init is OpemOCD command and you have to use “monitor” word here because this keyword is used to differentiate between a gdb command and the native OpenOCD command
- If you specifying the OpenOCD specific command then always the command must preceded by monitor
- This monitor keyword is not required if you are issuing this command through telnet

```
(gdb) monitor reset init
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
[stm32f4x.cpu] halted due to debug-request, current mode:
Thread
xPSR: 0x01000000 pc: 0x08000600 msp: 0x20018000
Unable to match requested speed 8000 kHz, using 4000 kHz
Unable to match requested speed 8000 kHz, using 4000 kHz
```

9.2.3 monitor flash write_image erase anyfile.elf

- Command to flash the .elf file

```
(gdb) monitor flash write_image erase final.elf
auto erase enabled
wrote 16384 bytes from file final.elf in 0.588179s (27.203
KiB/s)
```

9.2.4 monitor reset halt

- Command to halt the execution due to the debug request

```
(gdb) monitor reset halt
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
[stm32f4x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000654 msp: 0x20018000
```

9.2.5 monitor resume

- Command to resume the execution

```
(gdb) monitor resume
```

9.2.6 mdw

- Memory access command to read a word

```
(gdb) monitor mdw 0x20000000
0x20000000: b086b580
(gdb) monitor mdw 0x20000000 4
0x20000000: b086b580 00000000 00000000 00000000
```

- According to this program there is only one global initialized data that is “current task”
- The variable “current_task” value will vary between 0 to 4 during run time of the program as per the program logic.So,at any point in time , the value of the variable “current_task” must be between 0 to 4
- But when we read the memory location of this variable , we found an entirely different value. That means our data copy from flash to ram could have went wrong

```
Reset the Reset_Handler code
{
	//copy .data section to SRAM
	uint32_t size = (uint32_t)&_edata - (uint32_t)&_sdata;

	uint8_t *pDst = (uint8_t*)&_sdata;	//sram
	uint8_t *pSrc = (uint8_t*)&_etext;	//flash
	for(uint32_t i=0;i<size;i++)
	{
		*pDst++ = *pSrc++;
	}
```

```
        //init. the .bss section to zero in SRAM
        size = (uint32_t)&_ebss - (uint32_t)&_sbss;
        pDst = (uint8_t*)&_sbss;
        for(uint32_t i=0;i<size;i++)
        {
              *pDst++ = 0;
        }

        main();
}
```

- Make all again and launch gdb again with all above used commands

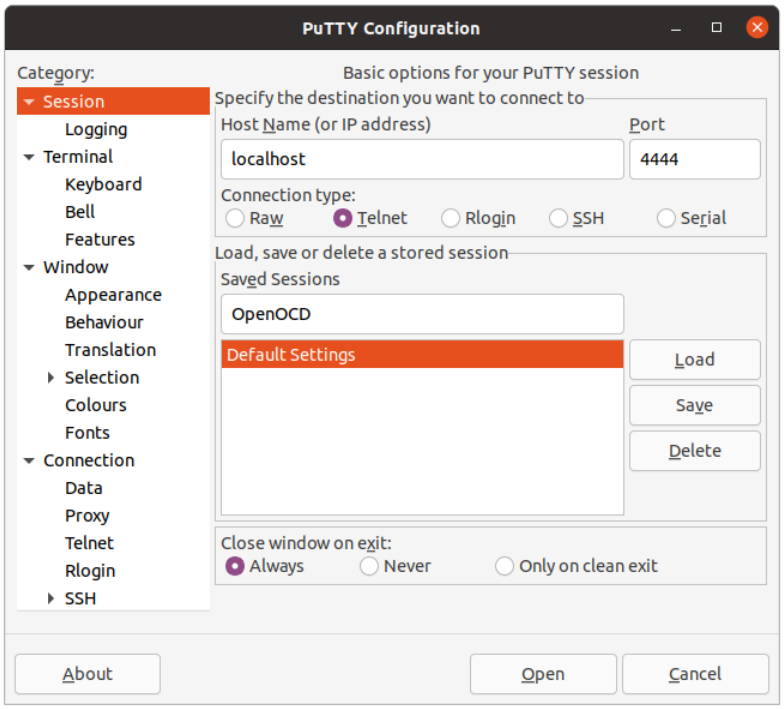
9.2.7 monitor bp [address len [hw]]

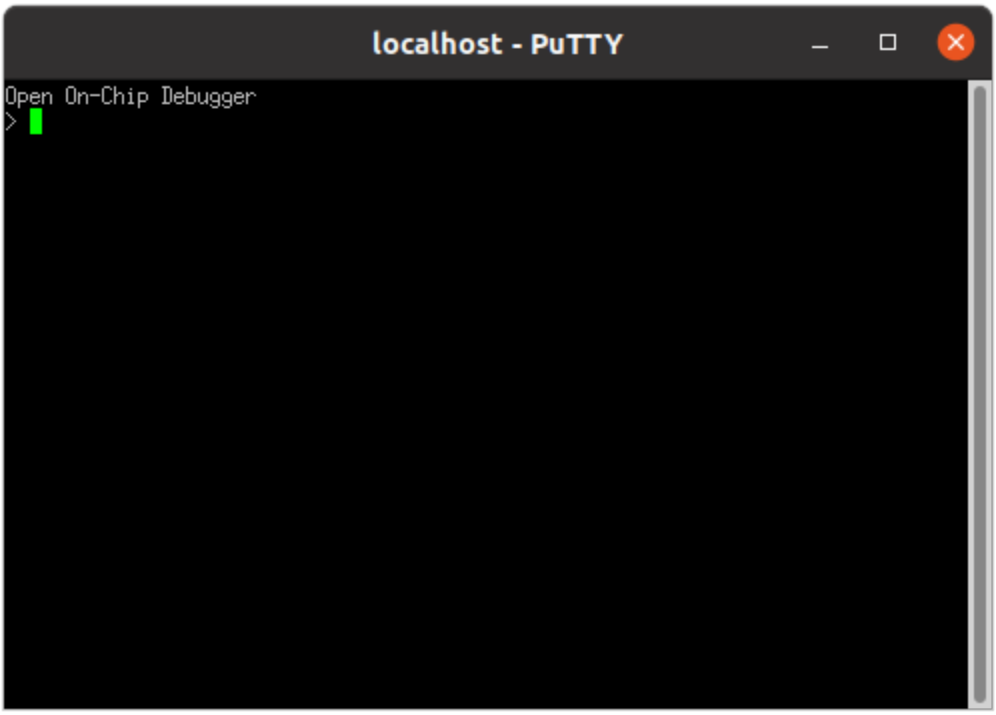
- The command used to set the breakpoint

```
(gdb) monitor bp 0x0000000008000052 2 hw
breakpoint set at 0x08000052
```

9.3 Connecting to telnet using OpenOCD through puTTy

- Open putty > enter host name as localhost > port number : 4444 > Connection type : Telnet > click open

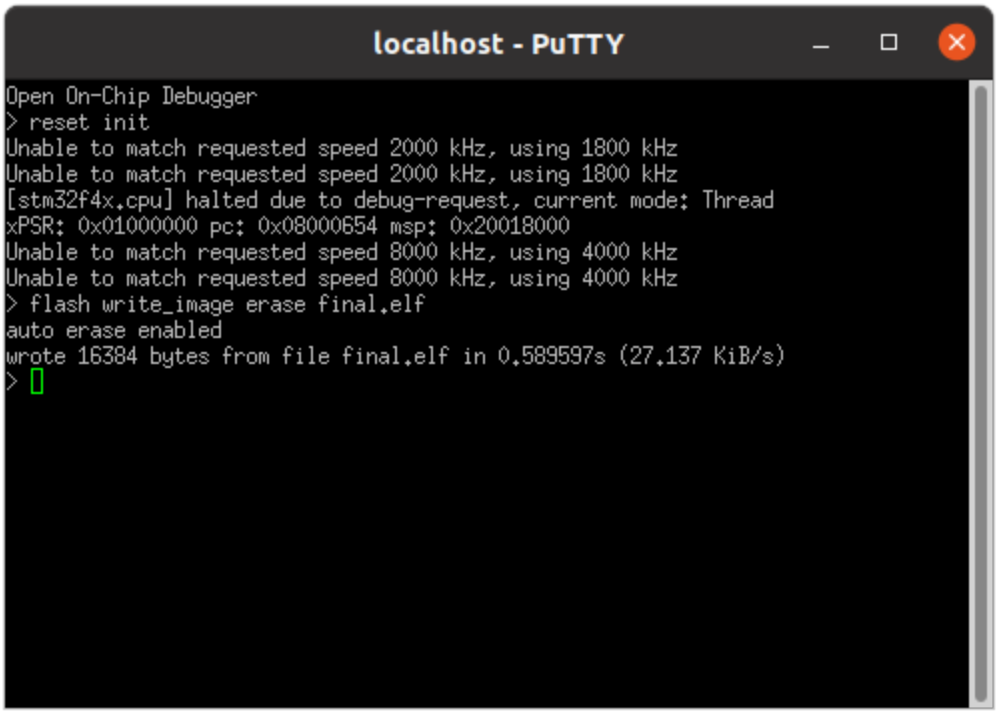




- You will observe the telnet connection in make load

```
Info : accepting 'telnet' connection on tcp/4444
Info : dropped 'telnet' connection
```

- We can use telnet server also the same but no need to use the word **“monitor”**



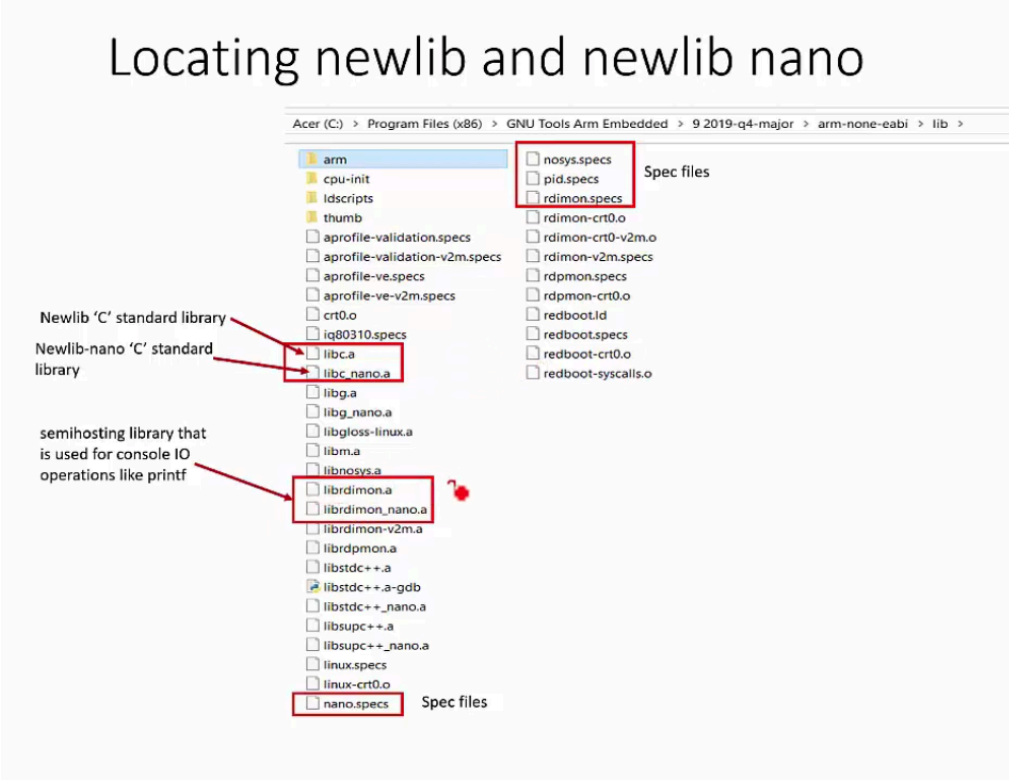
10. C standard library newlib and newlib-nano

10.1 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

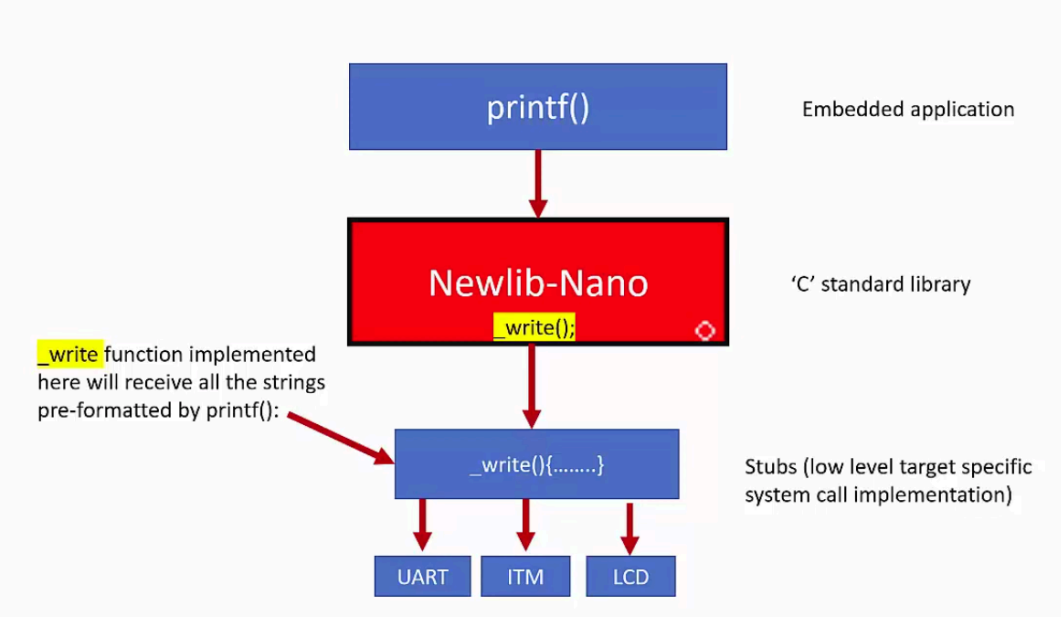
10.2 Newlib-nano

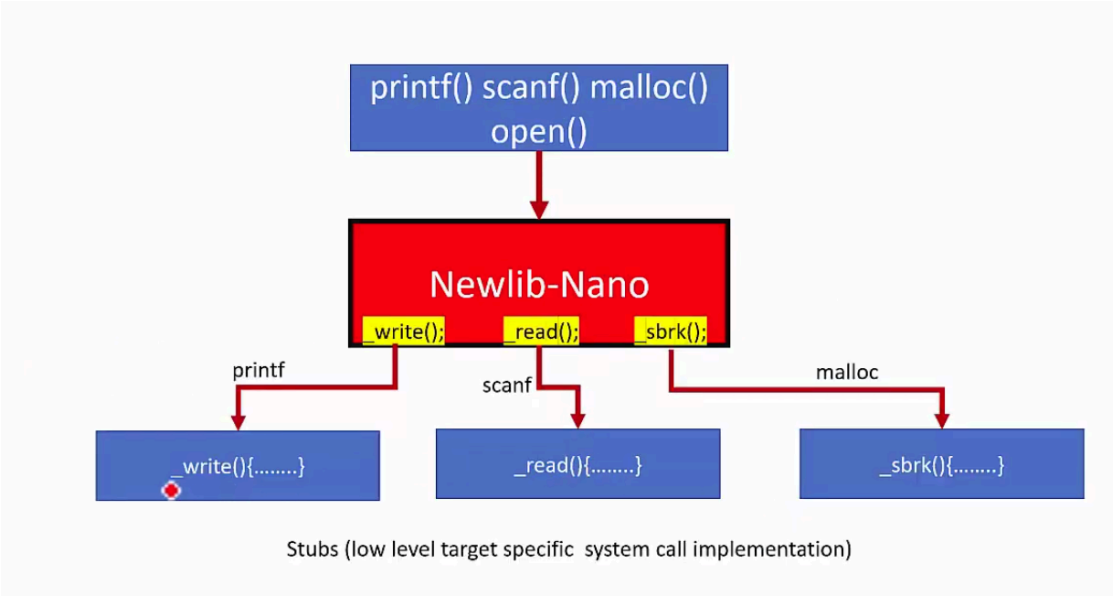
- Due to the increased feature set in newlib , it has become too bloated to use on the systems where the amount of memory is very much limited
- To provide a C-library with a minimal memory footprint suited for use with micro-controllers ,ARM introduced newlib-nano based on newlib



10.3 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





10.4 syscalls.c

- This file contains some important system calls such as write,read,close,open etc
- This file does not contain complete implementation of all those system calls
- This file just contains a function definition just to build our project without any errors

- Now modify the Makefile adding contents syscalls.c
- Now we want to link new lib nano so we need to modify the linker flags
- So we need to mention the nano lib specific spec file

Makefile

```
CC = arm-none-eabi-gcc
MACH=cortex-m4
CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -Wall -O0
LDFLAGS = -mcpu=$(MACH) -mthumb -mfloat-abi=soft --specs=nano.specs
-T stm32_ls.ld -Wl,-Map=final.map


all:main.o led.o stm32_startup.o final.elf

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

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

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

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

final.elf: main.o led.o stm32_startup.o syscalls.o
	$(CC) $(LDFLAGS)  $^ -o $@

clean:
	rm -rf *.o *.elf

load:
	sudo openocd -f
~/Desktop/xpack-openocd-0.12.0-4-linux-x64/xpack-openocd-0.12.0-4/ope
nocd/scripts/interface/stlink.cfg -f
~/Desktop/xpack-openocd-0.12.0-4-linux-x64/xpack-openocd-0.12.0-4/ope
nocd/scripts/target/stm32f4x.cfg -c "adapter speed 1000"
```

10.5 Nosys.spec

- It will be used when you dont want to use any system calls

10.6 Nano.specs

- Since we want to use newlib nano you need to use nano.specs file

```
Make some changes in the linker script also
ENTRY(Reset_Handler)

MEMORY
{
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 96K
}

SECTIONS
{
    .text :
    {
     *(.isr_vector)
        *(.text)                /* All text sections from .o files */
        *(.rodata)          /* Read-only data */
     . = ALIGN(4);
        _etext = .;             /* End of text */
    } > FLASH

    .data :
    {
        _sdata = .;             /* Start of .data section */
        *(.data)            /* All data sections */
     . = ALIGN(4);
        _edata = .;             /* End of .data section */
    } > SRAM AT> FLASH

    .bss :
    {
        _sbss = .;          /* Start of .bss section */
     __bss_start__ = _sbss;
          *(.bss)
     . = ALIGN(4);
     *(COMMON)
        _ebss = .;    /* End of .bss section */
     __bss_end__ = _ebss;
     . = ALIGN(4);
     end = .;
    } > SRAM
```

```
}
```

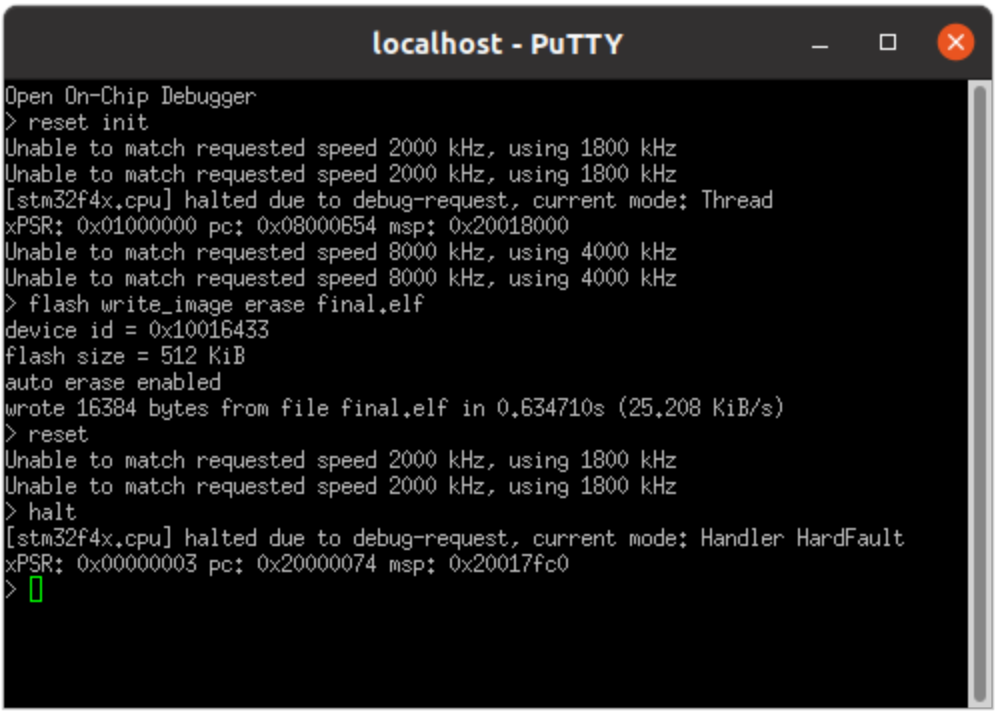
Remake the build

```
vlab@HYVLAB7:~/lochu$ make clean
rm -rf *.o *.elf
vlab@HYVLAB7:~/lochu$ make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0
main.c -o main.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0
led.c -o led.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0
stm32_startup.c -o stm32_startup.o
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0
syscalls.c -o syscalls.o
arm-none-eabi-gcc --specs=nano.specs -T stm32_ls.ld
-Wl,-Map=final.map  main.o led.o stm32_startup.o syscalls.o -o
final.elf
vlab@HYVLAB7:~/lochu$ ls
final.elf  led.h   main.h    stm32_ls.ld      syscalls.c
final.map  led.o   main.o    stm32_startup.c  syscalls.o
led.c      main.c  Makefile  stm32_startup.o
```

- By adding the new standard C library “newlib nano”, it introduced lots of sections
- Every function is considered as a sub text section
- We must resolve all sub sections yo single text section

```
vlab@HYVLAB7:~/lochu$ arm-none-eabi-objdump -h final.elf

final.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000024a4  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .init         00000018  080024a4  080024a4  000124a4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

```
 2 .fini         00000018  080024bc  080024bc  000124bc  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .random_section 00000084  080024d4  080024d4  000124d4  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .eh_frame     00000004  08002558  08002558  00012558  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 5 .ARM.exidx    00000008  0800255c  0800255c  0001255c  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .data         00000074  20000000  08002564  00020000  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 7 .init_array   00000004  20000074  080025d8  00020074  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 8 .fini_array   00000004  20000078  080025dc  00020078  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 9 .bss          00000114  2000007c  080025e0  0002007c  2**2
                 ALLOC
10 .ARM.attributes 0000002c  00000000  00000000  0002007c  2**0
                 CONTENTS, READONLY
11 .comment      000000a7  00000000  00000000  000200a8  2**0
                 CONTENTS, READONLY
12 .debug_line   00003bf7  00000000  00000000  0002014f  2**0
                 CONTENTS, READONLY, DEBUGGING, OCTETS
13 .debug_info   00012f9f  00000000  00000000  00023d46  2**0
                 CONTENTS, READONLY, DEBUGGING, OCTETS
14 .debug_abbrev 00003bf7  00000000  00000000  00036ce5  2**0
                 CONTENTS, READONLY, DEBUGGING, OCTETS
15 .debug_aranges 00000398  00000000  00000000  0003a8e0  2**3
                 CONTENTS, READONLY, DEBUGGING, OCTETS
16 .debug_str    0000157f  00000000  00000000  0003ac78  2**0
                 CONTENTS, READONLY, DEBUGGING, OCTETS
17 .debug_loc    00002923  00000000  00000000  0003c1f7  2**0
                 CONTENTS, READONLY, DEBUGGING, OCTETS
18 .debug_frame  000008d8  00000000  00000000  0003eb1c  2**2
                 CONTENTS, READONLY, DEBUGGING, OCTETS
19 .debug_ranges 000002a0  00000000  00000000  0003f3f4  2**0
                 CONTENTS, READONLY, DEBUGGING, OCTETS
```


localhost - PuTTY
Open On-Chip Debugger
> reset init
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
[stm32f4x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000654 msp: 0x20018000
Unable to match requested speed 8000 kHz, using 4000 kHz
Unable to match requested speed 8000 kHz, using 4000 kHz
> flash write_image erase final.elf
device id = 0x10016433
flash size = 512 KiB
auto erase enabled
wrote 16384 bytes from file final.elf in 0.634710s (25.208 KiB/s)
> reset
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
> halt
[stm32f4x.cpu] halted due to debug-request, current mode: Handler HardFault
xPSR: 0x00000003 pc: 0x20000074 msp: 0x20017fc0
>