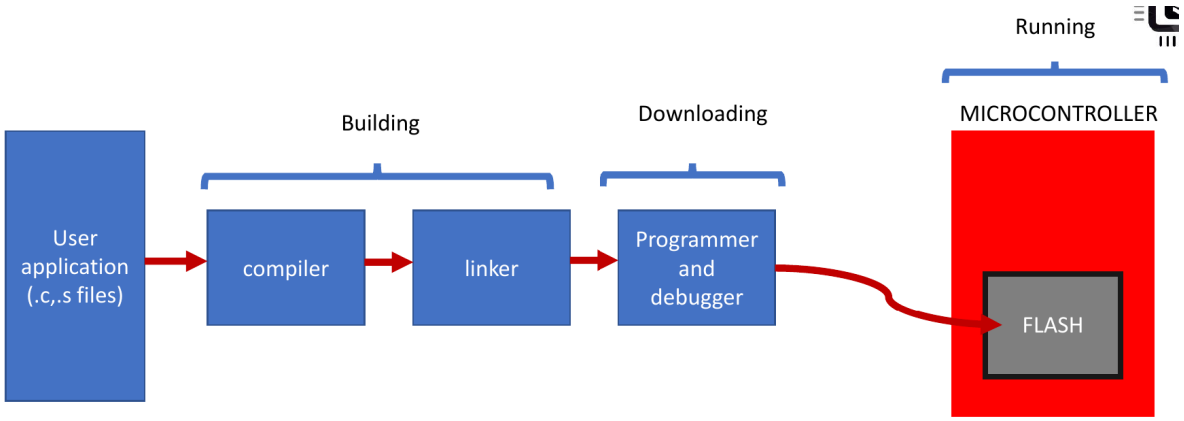
Building and Running Baremetal Executables for ARM target using GNU tools

Project Detailed overview :

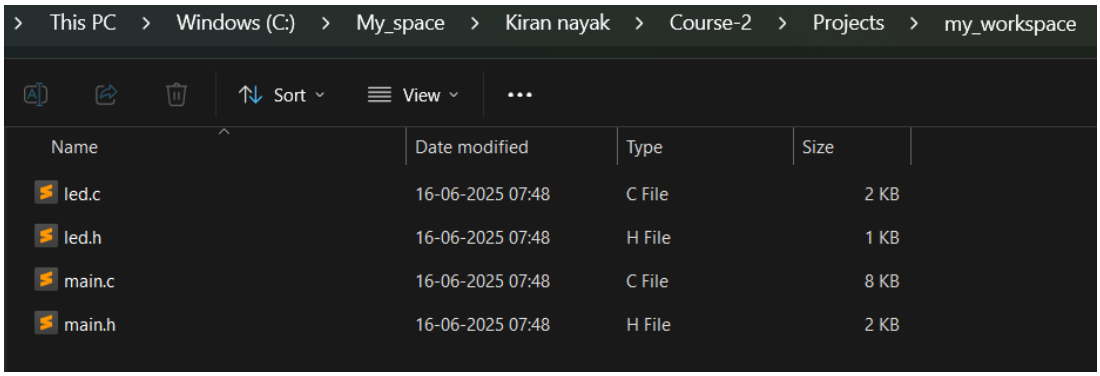
- Compiling a 'C' program for an embedded target without using an IDE
- Writing microcontroller startup file for STM32F4 MCU
- Writing own 'C' startup code(code which runs before main())
- Writing linker script file from scratch and understanding section placements
- Understanding different sections of the relocatable object file(.o files). Linking multiple .o files using linker script and generating application executable (.elf, bin, hex)
- Loading the final executable on the target using OpenOCD and GDB client

Brief overview :

- Write Linker script from scratch
- Write Startup file from scratch
- Use Command Line Tools to compile, link and generate binary
- Transfer the binary from host to target machine by using Command line tools given by OpenOCD and GDB

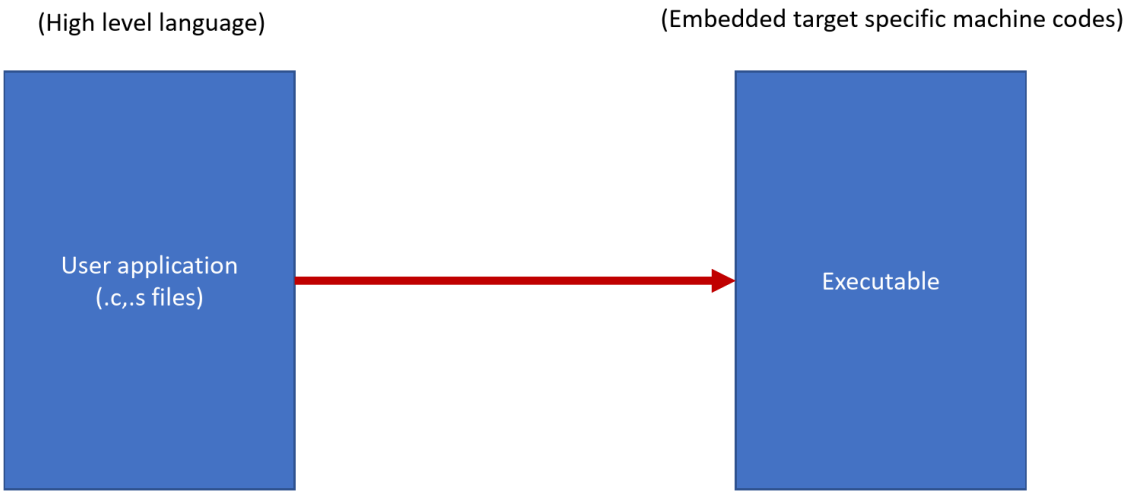


To achieve the above tasks, we need sample C program, for that we will take previously written programs from task_scheduler



Lets compile all these files using cross-compiler :

All these programs are written in high level language, our goal is to compile these high level language into embedded target specific machine codes (executable)



What is cross compiler ?

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

Cross-compiler toolchains :

- ✓ Toolchain or a cross-compilation toolchain is a collection of binaries which allows you to compile, assemble, link your applications.
- ✓ It also contains binaries to debug the application on the target

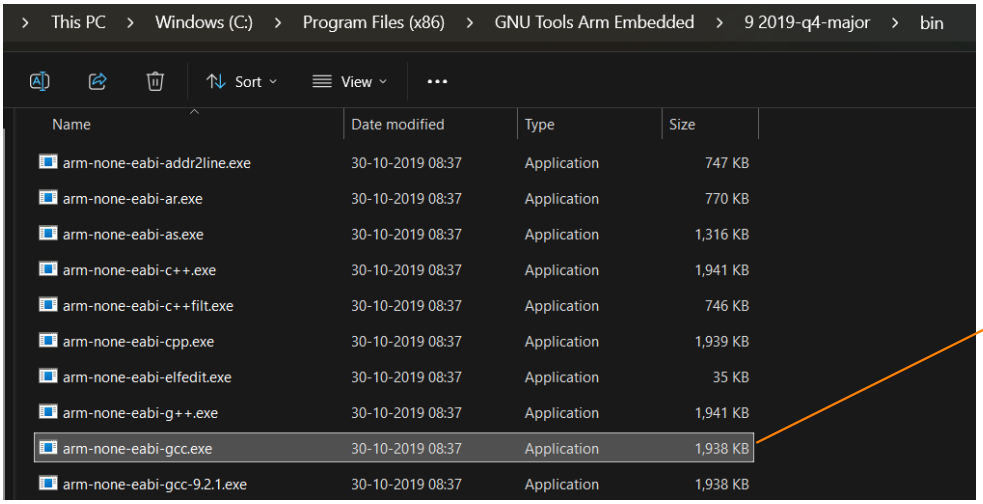
✓ Toolchain also comes with other binaries which help you to analyze the executables:

- dissect different sections of the executable
- disassemble
- Extract symbol and size information
- Convert executable to other formats such as bin, ihex
- Provides C standard libraries

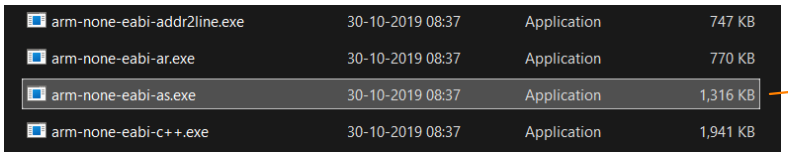
Popular toolchains :

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

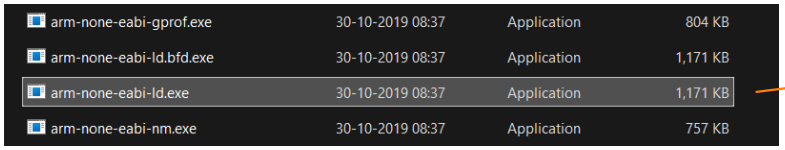
In this project, we will use GNU's Compiler Collections(GCC) Toolchain



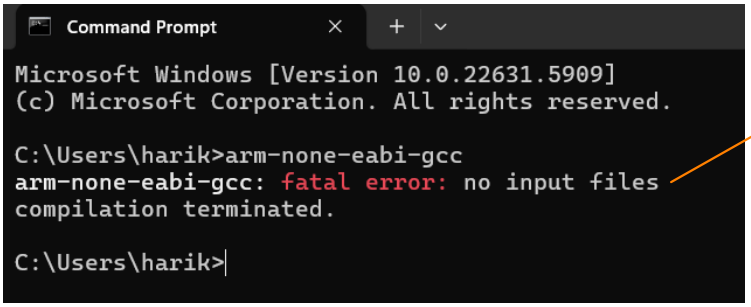
Our cross compiler



Assembler



Linker



Cross compiler is installed in our machine

Cross-toolchain important binaries :

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

linker
arm-none-eabi-ld

Assembler
arm-none-eabi-as

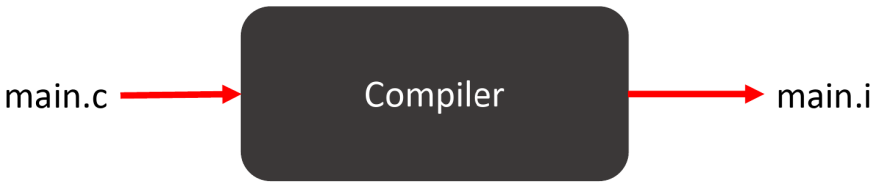
Elf file analyzer
arm-none-eabi-objdump
arm-none-eabi-readelf
arm-none-eabi-nm

Format converter
arm-none-eabi-objcopy

Note:

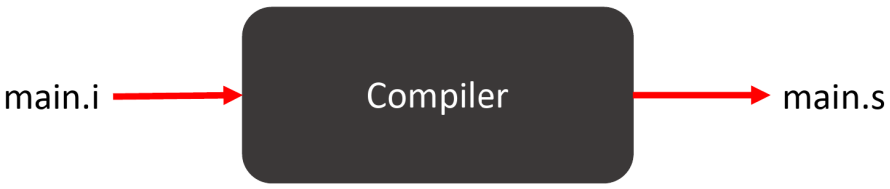
arm-none-eabi-gcc not only does compilation, it also assembles and links object files to create final executable file

Build process :

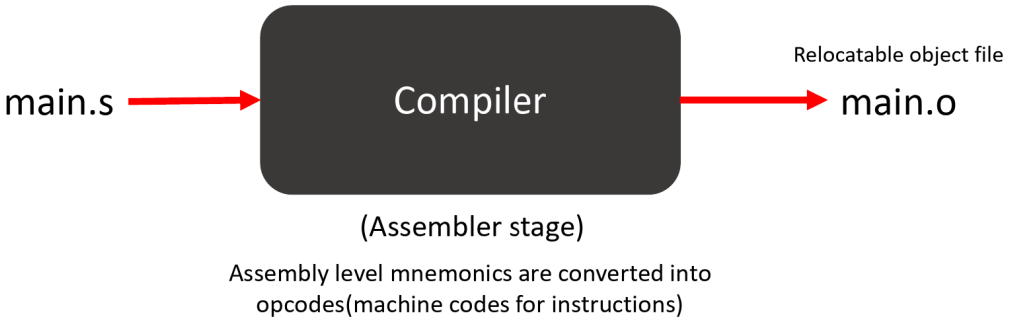


(Pre-processing stage)
All pre-processing directives will be resolved

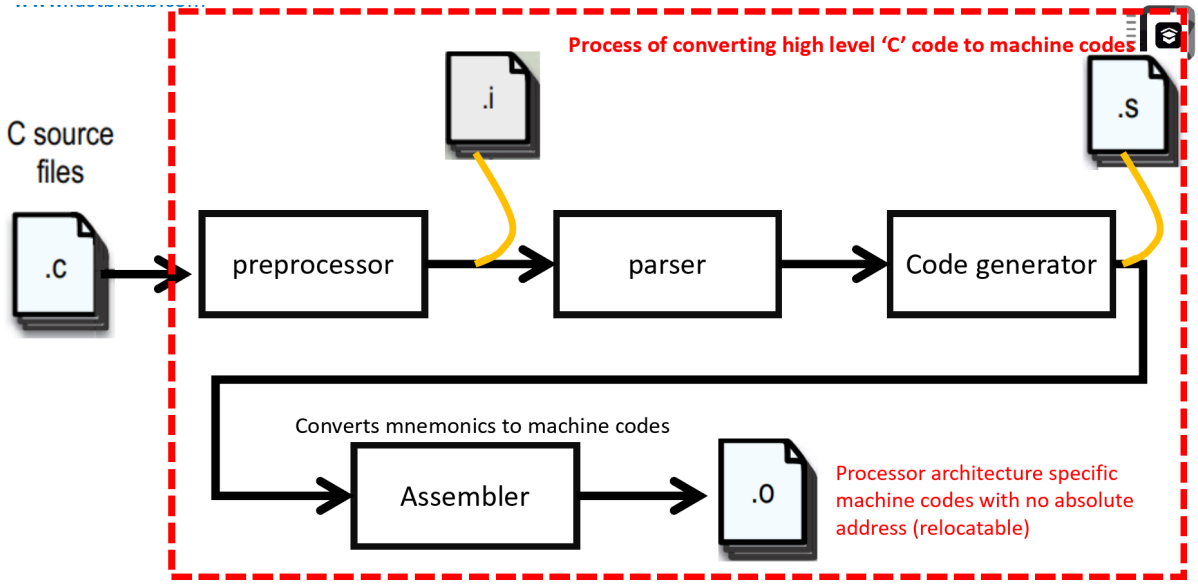
translating a source file into assembly language

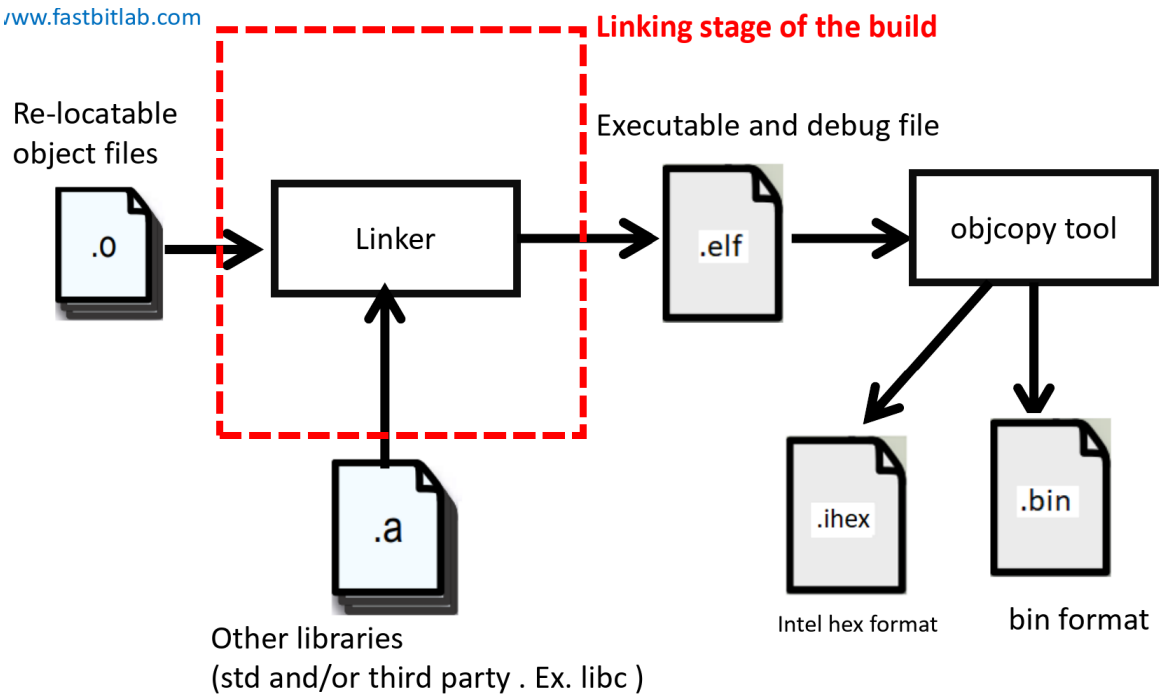


(Code generation stage)
Higher level language code statements will be converted into processor architectural level mnemonics

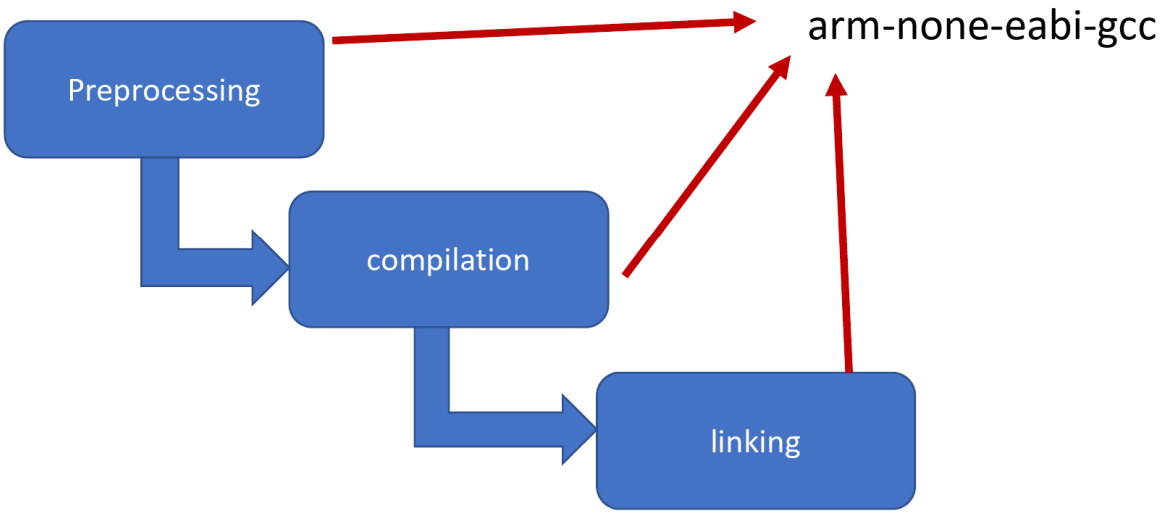


All these conversions will be done by the compiler itself, in our case compiler is arm-none-eabi-gcc





Summary of build process :




```
C:\Users\harik\project_space\Bare_metal_embedded>ls
led.c  led.h  main.c  main.h

C:\Users\harik\project_space\Bare_metal_embedded>dir
 Volume in drive C is Windows
 Volume Serial Number is 86B1-A999

 Directory of C:\Users\harik\project_space\Bare_metal_embedded

19-09-2025  20:58    <DIR>          .
19-09-2025  20:58    <DIR>          ..
16-06-2025  07:48             1,033 led.c
16-06-2025  07:48               572 led.h
16-06-2025  07:48             7,500 main.c
16-06-2025  07:48             1,172 main.h
               4 File(s)         10,277 bytes
               2 Dir(s)  65,507,500,032 bytes free
```

How do you just compile a C program and generate the relocatable file ?

```
C:\Users\harik\project_space\Bare_metal_embedded>arm-none-eabi-gcc main.c -o main.o
```

This command will compile, assemble and link

```
C:\Users\harik\project_space\Bare_metal_embedded>arm-none-eabi-gcc -c main.c -o main.o
```

This will do just compile and assemble

Linking will not be done

Compiler options :

`-c`

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

By default, the object file name for a source file is made by replacing the suffix ‘`.c`’, ‘`.i`’, ‘`.s`’, etc., with ‘`.o`’.

Unrecognized input files, not requiring compilation or assembly, are ignored.

`-S`

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix ‘`.c`’, ‘`.i`’, etc., with ‘`.s`’.

Input files that don’t require compilation are ignored.

`-E`

Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files that don’t require preprocessing are ignored.

```
C:\Users\harik\project_space\Bare_metal_embedded>arm-none-eabi-gcc -c main.c -o main.o
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s: Assembler messages:
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s:256: Error: selected processor does not support requested special purpose regist
er -- `msr MSP,r3'
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s:608: Error: selected processor does not support requested special purpose regist
er -- `msr PSP,R0'
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s:617: Error: selected processor does not support requested special purpose regist
er -- `msr CONTROL,R0'
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s:676: Error: selected processor does not support requested special purpose regist
er -- `msr PRIMASK,R0'
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s:709: Error: selected processor does not support requested special purpose regist
er -- `msr PRIMASK,R0'
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s:738: Error: selected processor does not support requested special purpose regist
er -- `mrs R0,PSP'
C:\Users\harik\AppData\Local\Temp\ccXij2Je.s:759: Error: selected processor does not support requested special purpose regist
er -- `msr PSP,R0'

C:\Users\harik\project_space\Bare_metal_embedded>
```

The compilation didn't go well, this is the problem with the assembler.

```
C:\Users\harik\AppData\Local
er -- `msr MSP,r3'
C:\Users\harik\AppData\Local
er -- `msr PSP,R0'
C:\Users\harik\AppData\Local
er -- `msr CONTROL,R0'
C:\Users\harik\AppData\Local
er -- `msr PRIMASK,R0'
C:\Users\harik\AppData\Local
er -- `msr PRIMASK,R0'
C:\Users\harik\AppData\Local
er -- `mrs R0,PSP'
C:\Users\harik\AppData\Local
er -- `msr PSP,R0'
```

These codes are there in our C program.

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

3.20 Machine-Dependent Options

Each target machine supported by GCC can have its own options—for example, to allow you to compile for a particular processor variant or ABI, or to control optimizations specific to that machine. By convention, the names of machine-specific options start with ‘`-m`’.

Some configurations of the compiler also support additional target-specific options, usually for compatibility with other compilers on the same platform.

- AArch64 Options
- Adapteva Epiphany Options
- AMD GCN Options
- ARC Options
- ARM Options
- AVR Options
- Blackfin Options

If you know the architecture, you can use the beow option :

`-march=`*name*`[+extension…]`

This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. This option can be used in conjunction with or instead of the `-mcpu=` option.

Permissible names are: ‘`armv4t`’, ‘`armv5t`’, ‘`armv5te`’, ‘`armv6`’, ‘`armv6j`’, ‘`armv6k`’, ‘`armv6kz`’, ‘`armv6t2`’, ‘`armv6z`’, ‘`armv6zk`’, ‘`armv7`’, ‘`armv7-a`’, ‘`armv7ve`’, ‘`armv8-a`’, ‘`armv8.1-a`’, ‘`armv8.2-a`’, ‘`armv8.3-a`’, ‘`armv8.4-a`’, ‘`armv8.5-a`’, ‘`armv8.6-a`’, ‘`armv9-a`’, ‘`armv7-r`’, ‘`armv8-r`’, ‘`armv6-m`’, ‘`armv6s-m`’, ‘`armv7-m`’, ‘`armv7e-m`’, ‘`armv8-m.base`’, ‘`armv8-m.main`’, ‘`armv8.1-m.main`’, ‘`iwmmxt`’ and ‘`iwmmxt2`’.

Additionally, the following architectures, which lack support for the Thumb execution state, are recognized but support is deprecated: ‘`armv4`’.

If you don't know the architecture, you can use the following :

`-mcpu=`*name*`[+extension…]`

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

Many of the supported CPUs implement optional architectural extensions. Where this is so the architectural extensions are normally enabled by default. If implementations that lack the extension exist, then the extension syntax can be used to disable those extensions that have been omitted. For floating-point and Advanced SIMD (Neon) instructions, the settings of the options `-mfloat-abi` and `-mfpu` must also be considered: floating-point and Advanced SIMD instructions will only be used if `-mfloat-abi` is not set to ‘`soft`’; and any setting of `-mfpu` other than ‘`auto`’ will override the available floating-point and SIMD extension instructions.

For example, ‘`cortex-a9`’ can be found in three major configurations: integer only, with just a floating-point unit or with floating-point and Advanced SIMD. The default is to enable all the instructions, but the extensions ‘`+nosimd`’ and ‘`+nofp`’ can be used to disable just the SIMD or both the SIMD and floating-point instructions respectively.

Permissible names for this option are the same as those for `-mtune`.

can be obtained by using this option. Permissible names are: ‘`arm7tdmi`’, ‘`arm7tdmi-s`’, ‘`arm710t`’, ‘`arm720t`’, ‘`arm740t`’, ‘`strongarm`’, ‘`strongarm110`’, ‘`strongarm1100`’, ‘`strongarm1110`’, ‘`arm8`’, ‘`arm810`’, ‘`arm9`’, ‘`arm9e`’, ‘`arm920`’, ‘`arm920t`’, ‘`arm922t`’, ‘`arm946e-s`’, ‘`arm966e-s`’, ‘`arm968e-s`’, ‘`arm926ej-s`’, ‘`arm940t`’, ‘`arm9tdmi`’, ‘`arm10tdmi`’, ‘`arm1020t`’, ‘`arm1026ej-s`’, ‘`arm10e`’, ‘`arm1020e`’, ‘`arm1022e`’, ‘`arm1136j-s`’, ‘`arm1136jf-s`’, ‘`mpcore`’, ‘`mpcorenovfp`’, ‘`arm1156t2-s`’, ‘`arm1156t2f-s`’, ‘`arm1176jz-s`’, ‘`arm1176jzf-s`’, ‘`generic-armv7-a`’, ‘`cortex-a5`’, ‘`cortex-a7`’, ‘`cortex-a8`’, ‘`cortex-a9`’, ‘`cortex-a12`’, ‘`cortex-a15`’, ‘`cortex-a17`’, ‘`cortex-a32`’, ‘`cortex-a35`’, ‘`cortex-a53`’, ‘`cortex-a55`’, ‘`cortex-a57`’, ‘`cortex-a72`’, ‘`cortex-a73`’, ‘`cortex-a75`’, ‘`cortex-a76`’, ‘`cortex-a76ae`’, ‘`cortex-a77`’, ‘`cortex-a78`’, ‘`cortex-a78ae`’, ‘`cortex-a78c`’, ‘`cortex-a710`’, ‘`ares`’, ‘`cortex-r4`’, ‘`cortex-r4f`’, ‘`cortex-r5`’, ‘`cortex-r7`’, ‘`cortex-r8`’, ‘`cortex-r52`’, ‘`cortex-r52plus`’, ‘`cortex-m0`’, ‘`cortex-m0plus`’, ‘`cortex-m1`’, ‘`cortex-m3`’, ‘`cortex-m4`’, ‘`cortex-m7`’, ‘`cortex-m23`’, ‘`cortex-m33`’, ‘`cortex-m35p`’, ‘`cortex-m52`’, ‘`cortex-m55`’, ‘`cortex-m85`’, ‘`cortex-x1`’, ‘`cortex-x1c`’, ‘`cortex-m1.small-multiply`’, ‘`cortex-m0.small-multiply`’, ‘`cortex-m0plus.small-multiply`’, ‘`exynos-m1`’, ‘`marvell-pj4`’, ‘`neoverse-n1`’, ‘`neoverse-n2`’, ‘`neoverse-v1`’, ‘`xscale`’, ‘`iwmmxt`’, ‘`iwmmxt2`’, ‘`ep9312`’, ‘`fa526`’, ‘`fa626`’, ‘`fa606te`’, ‘`fa626te`’, ‘`fmp626`’, ‘`fa726te`’, ‘`star-mc1`’, ‘`xgene1`’.

`-mthumb`
`-marm`

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

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

```
C:\Users\harik\project_space\Bare_metal_embedded>arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb main.c -o main.o

C:\Users\harik\project_space\Bare_metal_embedded>dir
 Volume in drive C is Windows
 Volume Serial Number is 86B1-A999

 Directory of C:\Users\harik\project_space\Bare_metal_embedded

19-09-2025  21:20    <DIR>          .
19-09-2025  20:58    <DIR>          ..
16-06-2025  07:48             1,033 led.c
16-06-2025  07:48               572 led.h
16-06-2025  07:48             7,500 main.c
16-06-2025  07:48             1,172 main.h
19-09-2025  21:20             4,228 main.o
               5 File(s)         14,505 bytes
               2 Dir(s)  65,495,158,784 bytes free

C:\Users\harik\project_space\Bare_metal_embedded>
```

Compilation went well and main.o is created

If you want to create just assembly file :

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

```
C:\Users\harik\project_space\Bare_metal_embedded>arm-none-eabi-gcc -S -mcpu=cortex-m4 -mthumb main.c -o main.S

C:\Users\harik\project_space\Bare_metal_embedded>dir
 Volume in drive C is Windows
 Volume Serial Number is 86B1-A999

 Directory of C:\Users\harik\project_space\Bare_metal_embedded

19-09-2025  21:23    <DIR>          .
19-09-2025  20:58    <DIR>          ..
16-06-2025  07:48             1,033 led.c
16-06-2025  07:48               572 led.h
16-06-2025  07:48             7,500 main.c
16-06-2025  07:48             1,172 main.h
19-09-2025  21:20             4,228 main.o
19-09-2025  21:23            17,824 main.S
               6 File(s)         32,329 bytes
               2 Dir(s)  65,493,495,808 bytes free

C:\Users\harik\project_space\Bare_metal_embedded>
```

main.S

```
1       .cpu cortex-m4
2       .eabi_attribute 20, 1
3       .eabi_attribute 21, 1
4       .eabi_attribute 23, 3
5       .eabi_attribute 24, 1
6       .eabi_attribute 25, 1
7       .eabi_attribute 26, 1
8       .eabi_attribute 30, 6
9       .eabi_attribute 34, 1
10      .eabi_attribute 18, 4
11      .file   "main.c"
12      .text
13      .global current_task
14      .data
15      .type   current_task, %object
16      .size   current_task, 1
17  current_task:
18      .byte   1
19      .global g_tick_count
20      .bss
21      .align  2
22      .type   g_tick_count, %object
23      .size   g_tick_count, 4
```

```
188 init_systick_timer:
189     @ args = 0, pretend = 0, frame = 24
190     @ frame_needed = 1, uses_anonymous_args = 0
191     @ link register save eliminated.
192     push    {r7}
193     sub sp, sp, #28
194     add r7, sp, #0
195     str r0, [r7, #4]
196     ldr r3, .L24
197     str r3, [r7, #20]
198     ldr r3, .L24+4
199     str r3, [r7, #16]
200     ldr r2, .L24+8
201     ldr r3, [r7, #4]
202     udiv    r3, r2, r3
203     subs    r3, r3, #1
204     str r3, [r7, #12]
205     ldr r3, [r7, #20]
206     movs    r2, #0
```

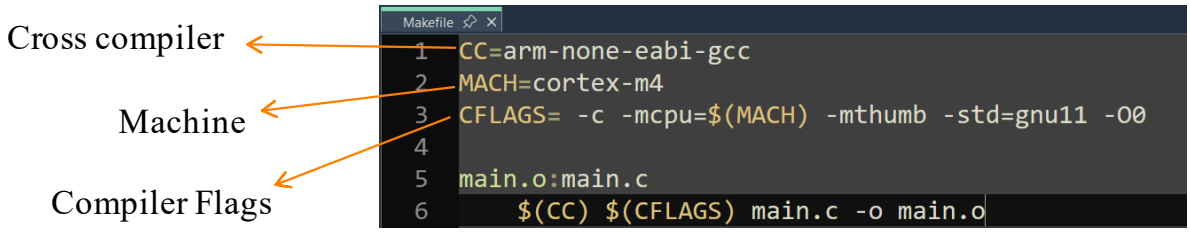
→ Thumb instructions

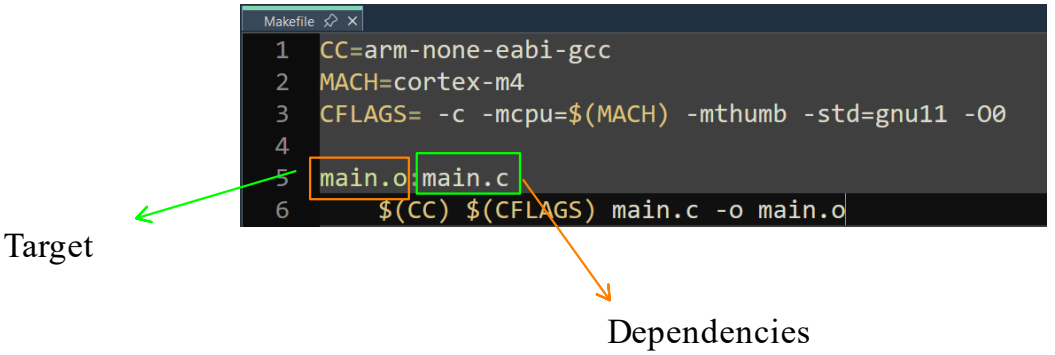
To automate the build process, we can create one makefile.

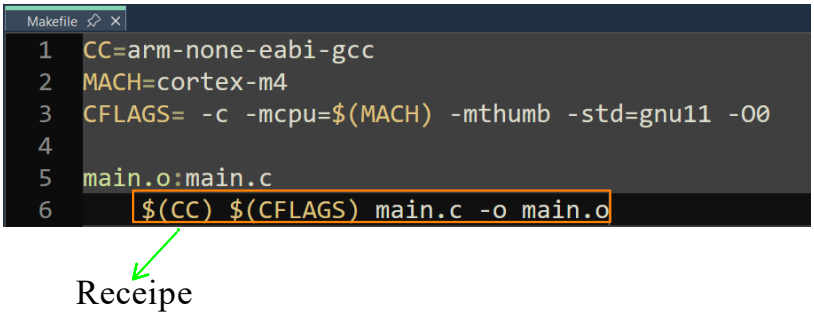
```
Makefile
1  CC=arm-none-eabi-gcc
2  MACH=cortex-m4
3  CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -O0
4
5  main.o:main.c
6      $(CC) $(CFLAGS) main.c -o main.o
```

```
C:\Users\harik\project_space\Bare_metal_embedded>make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 main.c -o main.o

C:\Users\harik\project_space\Bare_metal_embedded>ls
led.c  led.h  main.c  main.h  main.o  main.S  Makefile

C:\Users\harik\project_space\Bare_metal_embedded>
```

Lets deconstruct the nature of makefile :







A receipt is command to create output file from input files

You can do like this :

```
Makefile
1  CC=arm-none-eabi-gcc
2  MACH=cortex-m4
3  CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -O0
4
5  main.o:main.c
6      $(CC) $(CFLAGS) $^ -o $@
```

$^ - denotes dependency
$@ - denotes target

Also this

```
Makefile
1  CC=arm-none-eabi-gcc
2  MACH=cortex-m4
3  CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -O0
4
5  main.o:main.c
6      $(CC) $(CFLAGS) -o $@ $^
```

Analyzing .o file (Relocatable object files):

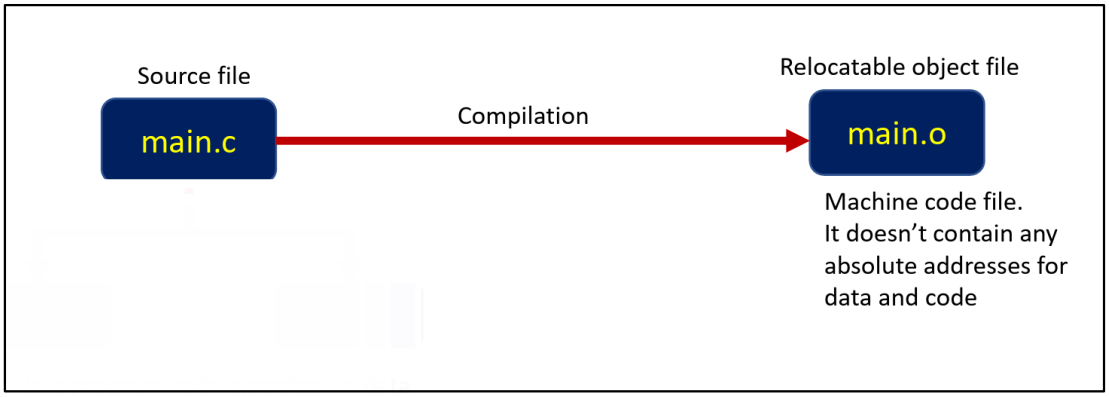
- main.o is in elf format(Executable and linkable format)
- ELF is a standard file format for object files and executable files when you use GCC
- A file format standard describes a way of organizing various elements(data, read-only data, code, uninitialized data, etc.) of a program in different sections.

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

What main.o contains ?

Machine specific code and the data of the program.



Lets analyze the .o file using command line tools :

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-obj
arm-none-eabi-objcopy.exe   arm-none-eabi-objdump.exe
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe
Usage: C:\Program Files (x86)\GNU Tools Arm Embedded\9 2019-q4-major\bin\arm-none-eabi-objdump.exe <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
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000004f4  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000001  00000000  00000000  00000528  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  0000052c  2**2
                  ALLOC
  3 .rodata       00000045  00000000  00000000  0000052c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000007a  00000000  00000000  00000571  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002e  00000000  00000000  000005eb  2**0
                  CONTENTS, READONLY
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000004f4  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000001  00000000  00000000  00000528  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  0000052c  2**2
                  ALLOC
  3 .rodata       00000045  00000000  00000000  0000052c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000007a  00000000  00000000  00000571  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002e  00000000  00000000  000005eb  2**0
                  CONTENTS, READONLY
```

File format is elf

Little endian

For ARM architecture

Holds code or instructions of the program

Initialised data of the program

All uninitialised data

Readonly data

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000004f4  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000001  00000000  00000000  00000528  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  0000052c  2**2
                  ALLOC
  3 .rodata       00000045  00000000  00000000  0000052c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000007a  00000000  00000000  00000571  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002e  00000000  00000000  000005eb  2**0
                  CONTENTS, READONLY
```

Apart from these you can also add user defined sections

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -d main.o > main_log
```

We have disassembled the main.o and redirect the output to main_log file

main_log

```
1
2   main.o:     file format elf32-littlearm
3
4
5   Disassembly of section .text:
6
7   00000000 <main>:
8      0:   b580        push    {r7, lr}
9      2:   af00        add r7, sp, #0
10     4:   f7ff fffe   bl  24c <enable_processor_faults>
11     8:   4807        ldr r0, [pc, #28]   ; (148 <init_tasks_stack+0x16>)
12     a:   f7ff fffe   bl  128 <init_scheduler_stack>
13     e:   f7ff fffe   bl  132 <init_tasks_stack>
14    12:   f7ff fffe   bl  0 <led_init_all>
15    16:   f44f 707a   mov.w   r0, #1000   ; 0x3e8
16    1a:   f7ff fffe   bl  c0 <init_systick_timer>
17    1e:   f7ff fffe   bl  358 <switch_sp_to_psp>
18    22:   f7ff fffe   bl  32 <task1_handler>
19    26:   e7fe        b.n 26 <main+0x26>
20    28:   2001ec00    .word   0x2001ec00
21
22  0000002c <idle_task>:
23    2c:   b480        push    {r7}
24    2e:   af00        add r7, sp, #0
25    30:   e7fe        b.n 30 <idle_task+0x4>
26
```

This command helps us to understand various assembly level instructions generated for different functions of our program

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -s main.o

main.o:     file format elf32-littlearm

Contents of section .text:
 0000 80b500af fff7feff 0748fff7 fefffff7  .........H......
 0010 fefffff7 feff4ff4 7a70fff7 fefffff7  ......O.zp......
 0020 fefffff7 fefffee7 00ec0120 80b400af  ........... ....
 0030 fee780b5 00af0c20 fff7feff 0448fff7  ....... .....H..
 0040 feff0c20 fff7feff 0148fff7 fefff2e7  ... .....H......
 0050 d0121300 80b500af 0d20fff7 feff0548  ......... .....H
 0060 fff7feff 0d20fff7 feff0248 fff7feff  ..... .....H....
 0070 f2e700bf 68890900 80b500af 0f20fff7  ....h........ ..
 0080 feff0548 fff7feff 0f20fff7 feff0248  ...H..... .....H
 0090 fff7feff f2e700bf b4c40400 80b500af  ................
 00a0 0e20fff7 feff0548 fff7feff 0e20fff7  . .....H..... ..
 00b0 feff0248 fff7feff f2e700bf 5a620200  ...H........Zb..
 00c0 80b487b0 00af7860 144b7b61 144b3b61  ......x`.K{a.K;a
 00d0 144a7b68 b2fbf3f3 013bfb60 7b690022  .J{h.....;.`{i."
 00e0 1a607b69 1a68fb68 1a437b69 1a603b69  .`{i.h.h.C{i.`;i
```

Contents of different sections

Collection of opcodes or machine instructions

```
Contents of section .data:
 0000 01                                    .
Contents of section .rodata:
 0000 45786365 7074696f 6e203a20 48617264  Exception : Hard
 0010 6661756c 74000000 45786365 7074696f  fault...Exceptio
 0020 6e203a20 4d656d4d 616e6167 65000000  n : MemManage...
 0030 45786365 7074696f 6e203a20 42757346  Exception : BusF
 0040 61756c74 00                          ault.
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -D main.o > main_log
```

This command helps you to get disassembly of all the sections

```
Disassembly of section .data:

00000000 <current_task>:
   0:   Address 0x00000000 is out of bounds.


Disassembly of section .bss:

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

```
Disassembly of section .rodata:

00000000 <.rodata>:
   0:   65637845    strbvs  r7, [r3, #-2117]!   ; 0xfffff7bb
   4:   6f697470    svcvs   0x00697470
   8:   203a206e    eorscs  r2, sl, lr, rrx
   c:   64726148    ldrbtvs r6, [r2], #-328 ; 0xfffffeb8
  10:   6c756166    ldfvse  f6, [r5], #-408 ; 0xfffffe68
  14:   00000074    andeq   r0, r0, r4, ror r0
  18:   65637845    strbvs  r7, [r3, #-2117]!   ; 0xfffff7bb
  1c:   6f697470    svcvs   0x00697470
  20:   203a206e    eorscs  r2, sl, lr, rrx
  24:   4d6d654d    cfstr64mi   mvdx6, [sp, #-308]! ; 0xfffffecc
  28:   67616e61    strbvs  r6, [r1, -r1, ror #28]!
  2c:   00000065    andeq   r0, r0, r5, rrx
  30:   65637845    strbvs  r7, [r3, #-2117]!   ; 0xfffff7bb
  34:   6f697470    svcvs   0x00697470
  38:   203a206e    eorscs  r2, sl, lr, rrx
  3c:   46737542    ldrbtmi r7, [r3], -r2, asr #10
  40:   746c7561    strbtvc r7, [ip], #-1377    ; 0xfffffa9f
    ...
```

When you are looking at disassembly of data sections, don't give importance to this assembly instructions, these are compiler made

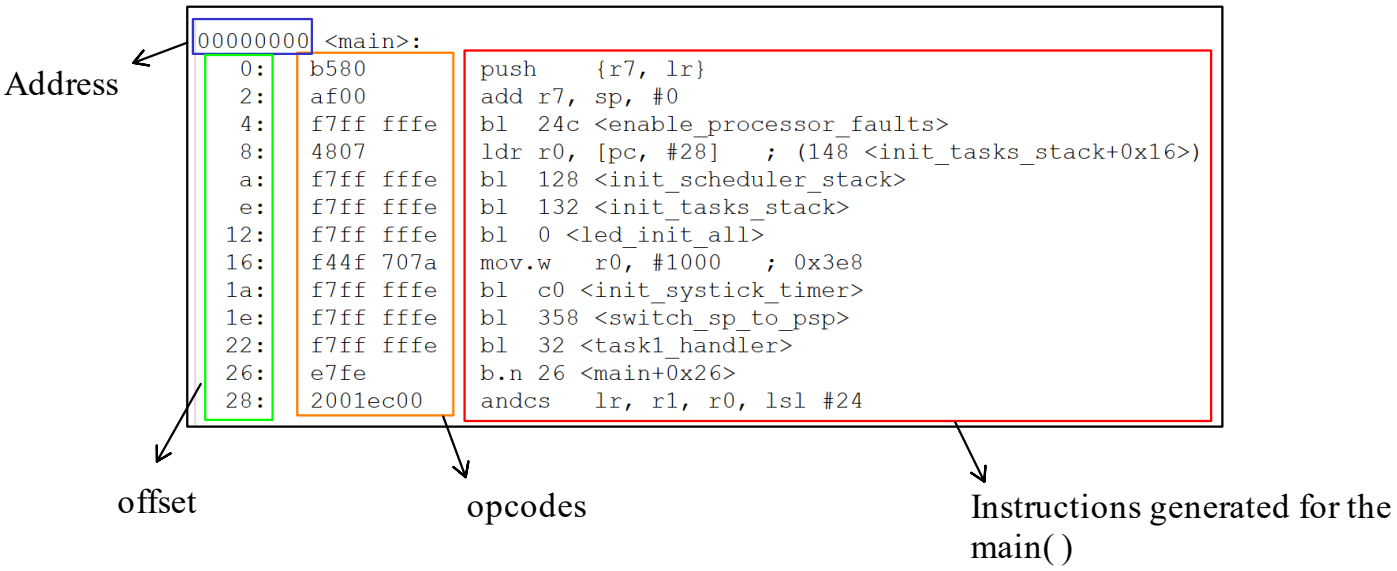
These are not true instructions

Remember data sections contain data, not instructions.

How instructions are generated ?

We ran disassembler. It sees every number(data) as an opcode.

Why .o file is called as relocatable object file ?



Each opcode will be placed at the address differentiated by the offset

This address is not an absolute address, you should mention the address using LinkerScript

We can relocate as per our need, and it will be automatically aligned by its appropriate offset.

Thats why its called as relocatable object file.

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

all:main.o led.o

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

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

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 -o led.o led.c
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -D led.o > led_log
```

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


Disassembly of section .text:

00000000 <delay>:
   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
```

```
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  24c <enable_processor_faults>
   8:   4807        ldr r0, [pc, #28]   ; (148 <init_tasks_stack+0x16>)
   a:   f7ff fffe   bl  128 <init_scheduler_stack>
   e:   f7ff fffe   bl  132 <init_tasks_stack>
  12:   f7ff fffe   bl  0 <led_init_all>
  16:   f44f 707a   mov.w   r0, #1000   ; 0x3e8
  1a:   f7ff fffe   bl  c0 <init_systick_timer>
  1e:   f7ff fffe   bl  358 <switch_sp_to_psp>
  22:   f7ff fffe   bl  32 <task1_handler>
  26:   e7fe        b.n 26 <main+0x26>
  28:   2001ec00    andcs   lr, r1, r0, lsl #24
```

If you notice both .text section of main.o and led.o as base address starts at 0

When you merge the opcodes, there will be conflict, so thats only you will relocate these sections using linkerScript.

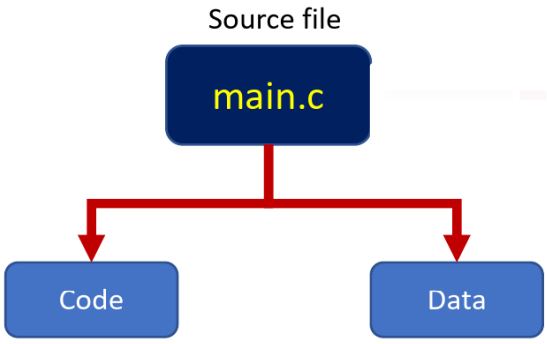
Relocatable address you have to decide based on the microcontroller or processor memory map.

Every section of .o file starts with address 0, you have to provide proper absolute address using LinkerScript.

Code and Data of a program :

What is a program ?

A program is a collection of code and data.



In a program code operates on data

Where do you store the code ?

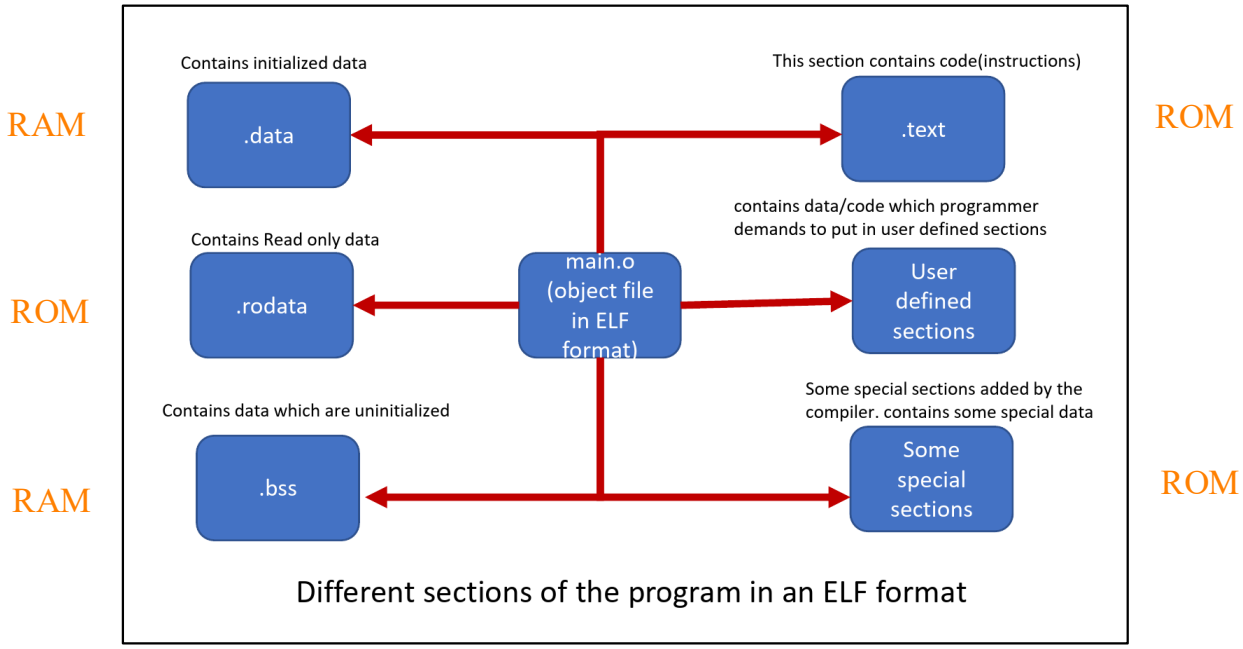
Instructions (codes) are stored in FLASH memory of the microcontroller

Where do you store data ?

Data (variables) can be stored in main memory (RAM) or in FLASH depends upon its nature.

Why data is stored in RAM but not in FLASH ?

Because data are nothing but variables, they may change anytime during runtime of the program. So it makes sense to store read only data in flash but not variable data.

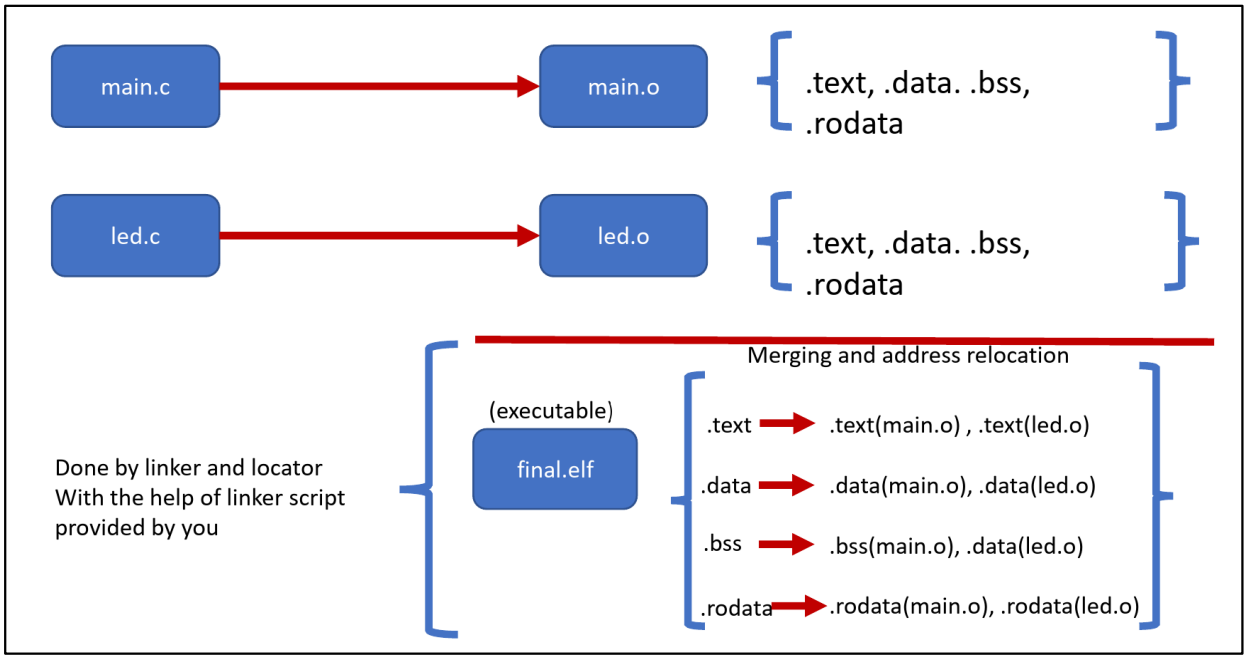


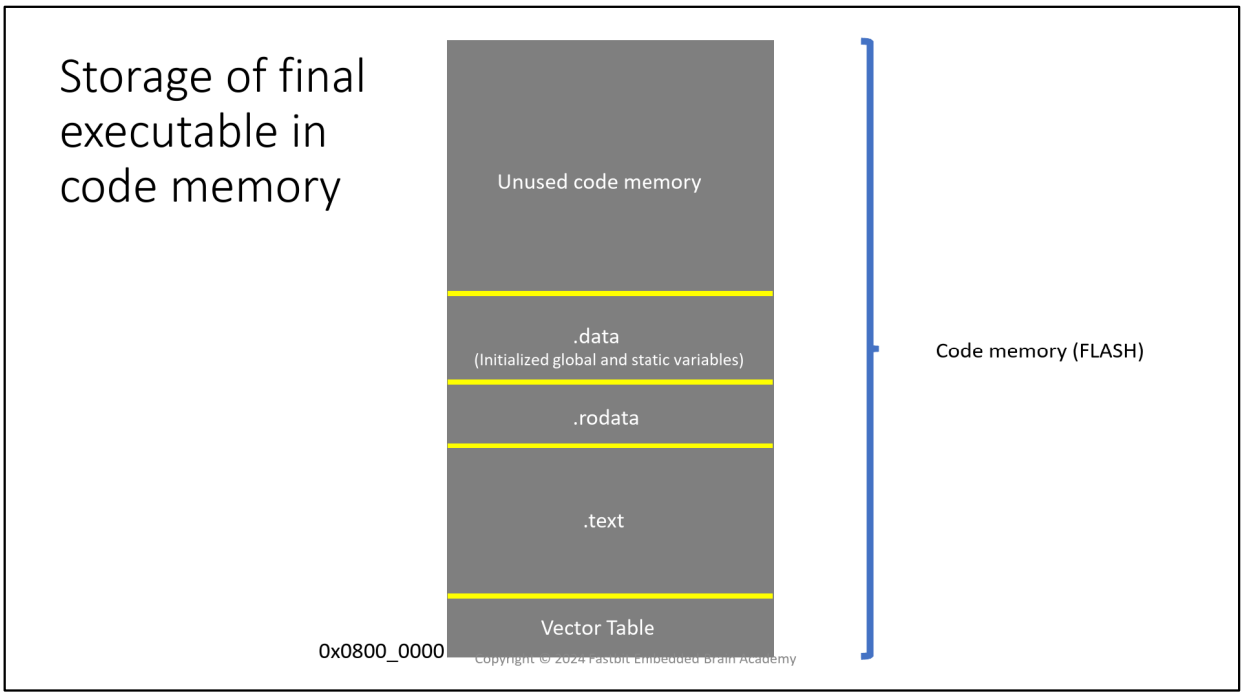
Different sections of the program in an ELF format

All these relocations have to be done using LinkerScript

Linker and Locator :

- ✓ Use the linker to merge similar sections of different object files and to resolve all undefined symbols of different object files.
- ✓ Locator (part of linker) takes the help of a linker script written by you to understand how you wish to merge different sections and assigns mentioned addresses to different sections.
- ✓ Locator is the one which provides absolute addresses to all the sections which have got merged.





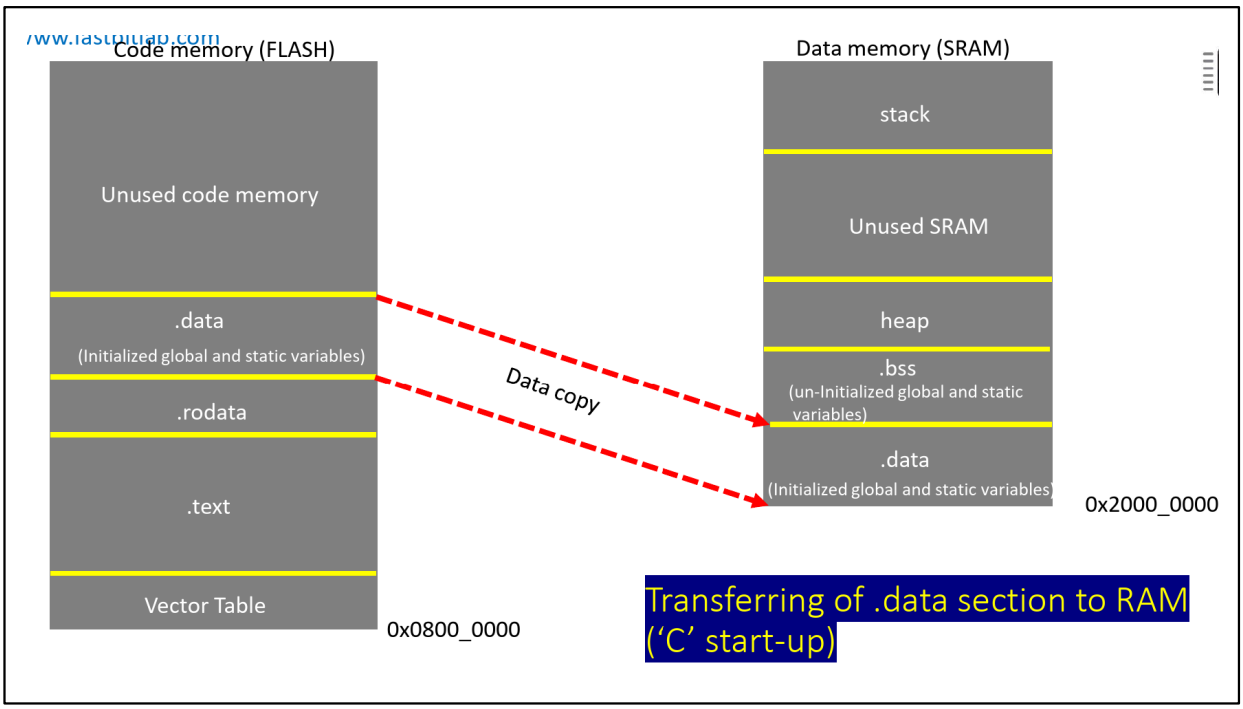
Vector table will be included in our startup file.

We know .data section is read and write section (i.e) it may change during runtime of the program

We should relocate or copy this section to RAM

This is done by the startup code.

Before calling main( ), startup code will run, it will copy data section contents from FLASH to RAM



This copying of .data section from FLASH to SRAM is called as C start-up.

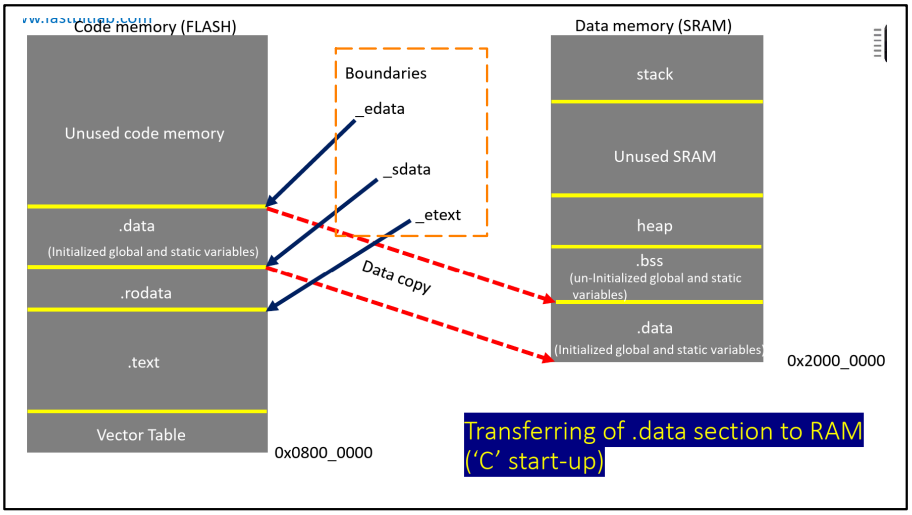
It will also initialize all .bss sections data with zero.

Startup code can also re initialize the stack pointer

After all these C startup code is going to call main( ) routine of our application.

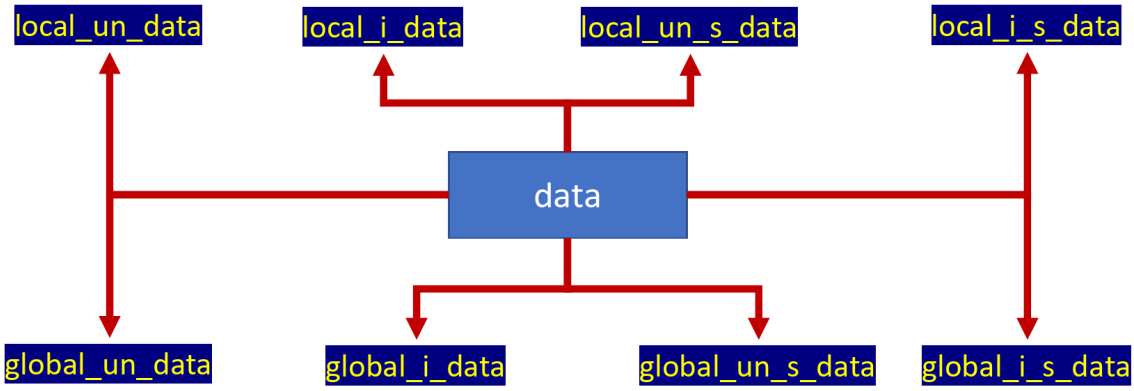
Copying .data section and zeroing .bss section is important functions of the startup code.

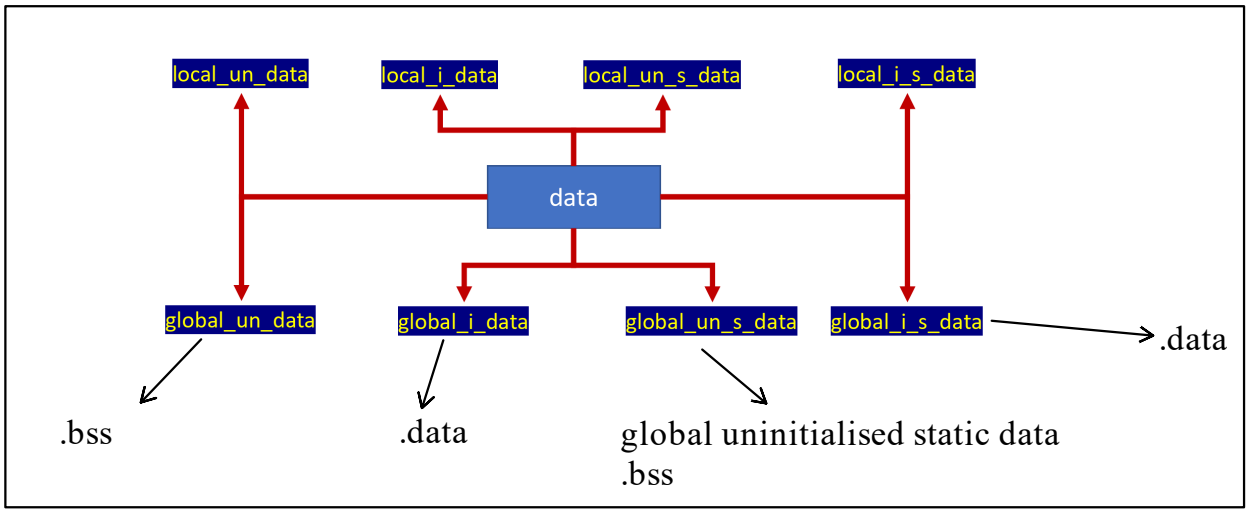
Inorder to do the data copying, we need to know the boundary addresses

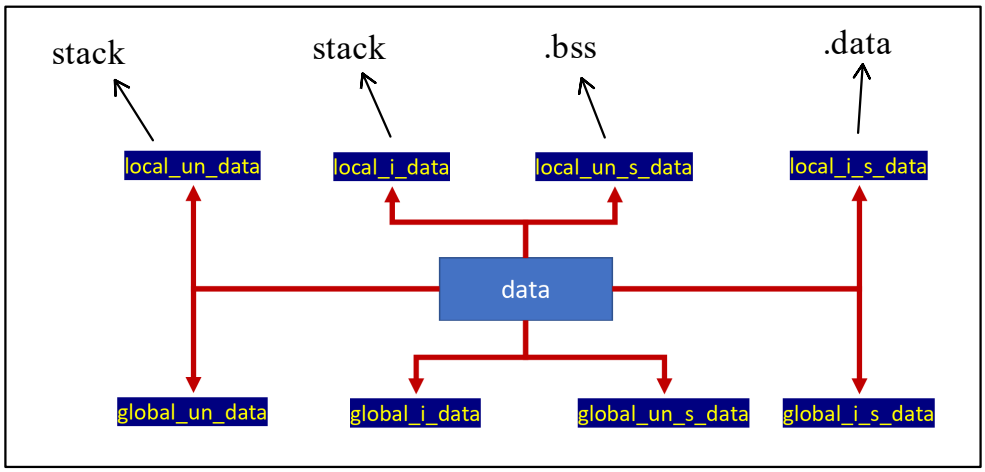


Atleast the initial address stored in a symbol _sdata and size of the data section

Different data of a program and related sections :







Initialised Global static -> If these words come together then its in .data section

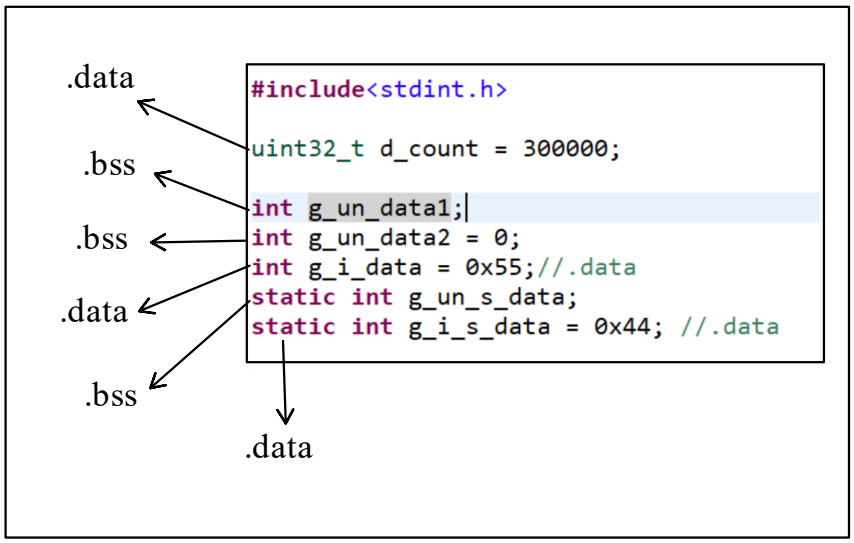
Variable (Data)	LOAD time LMA	RUN time VMA	Section	Note
Global initialized	FLASH	RAM	.data	Should be copied from flash to ram by startup code
Global uninitialized	----	RAM	.bss	startup code reserves space for this data in RAM and initializes to zero
Global static initialized	FLASH	RAM	.data	Should be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

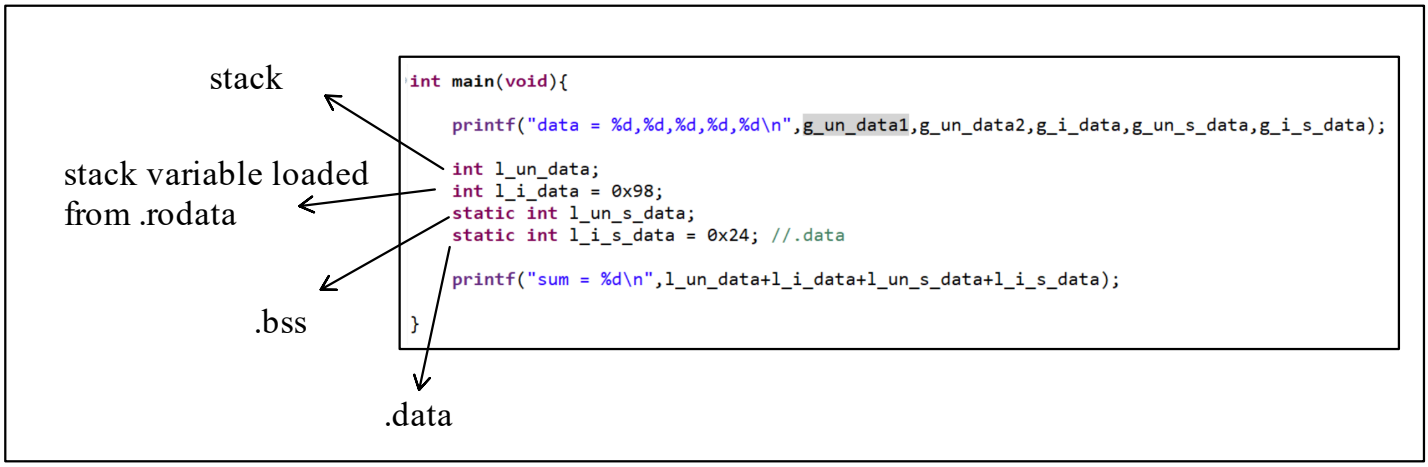
LMA - Load Memory Address
VMA - Virtual Memory Address

bss and data section :

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

```c
#include<stdint.h>

uint32_t d_count = 300000;

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


int main(void){

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

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

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

}
```





Writing startup file from scratch

Startup file of a microcontroller :

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

Note:

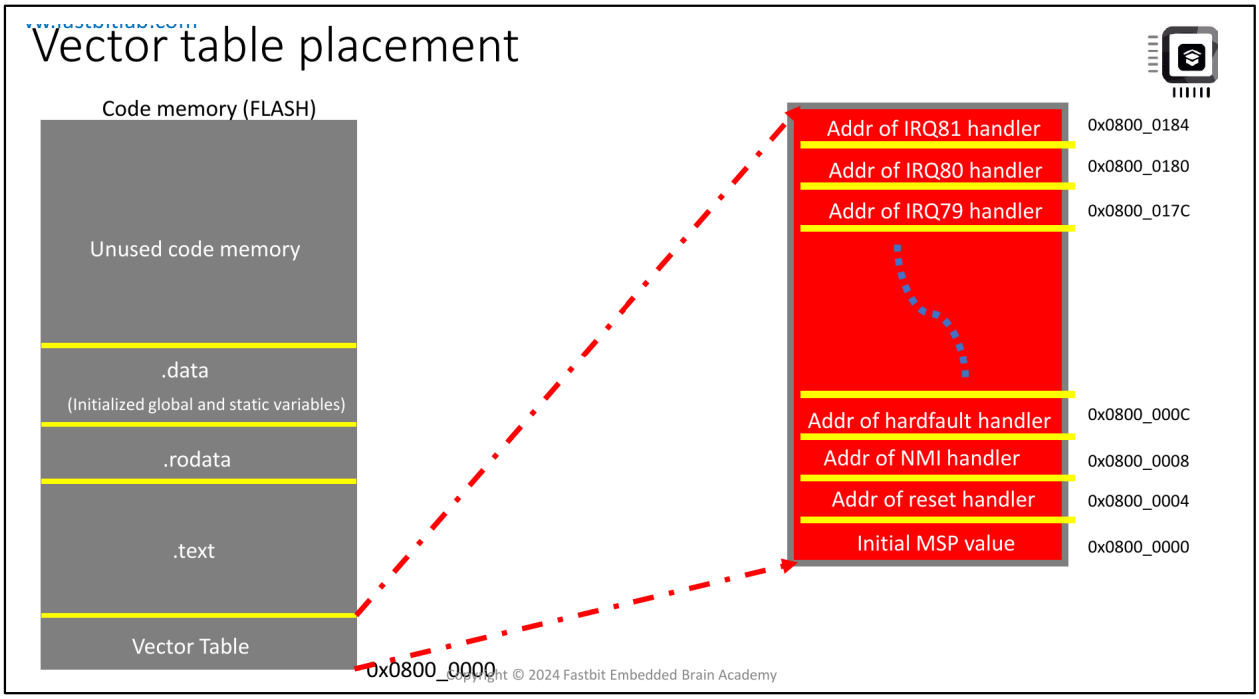
In some cases startup code may have to turn on some co-processors like FPU. E.g if main( ) app is using floating point operation, it has to be turned on prior to that otherwise there will be a exception

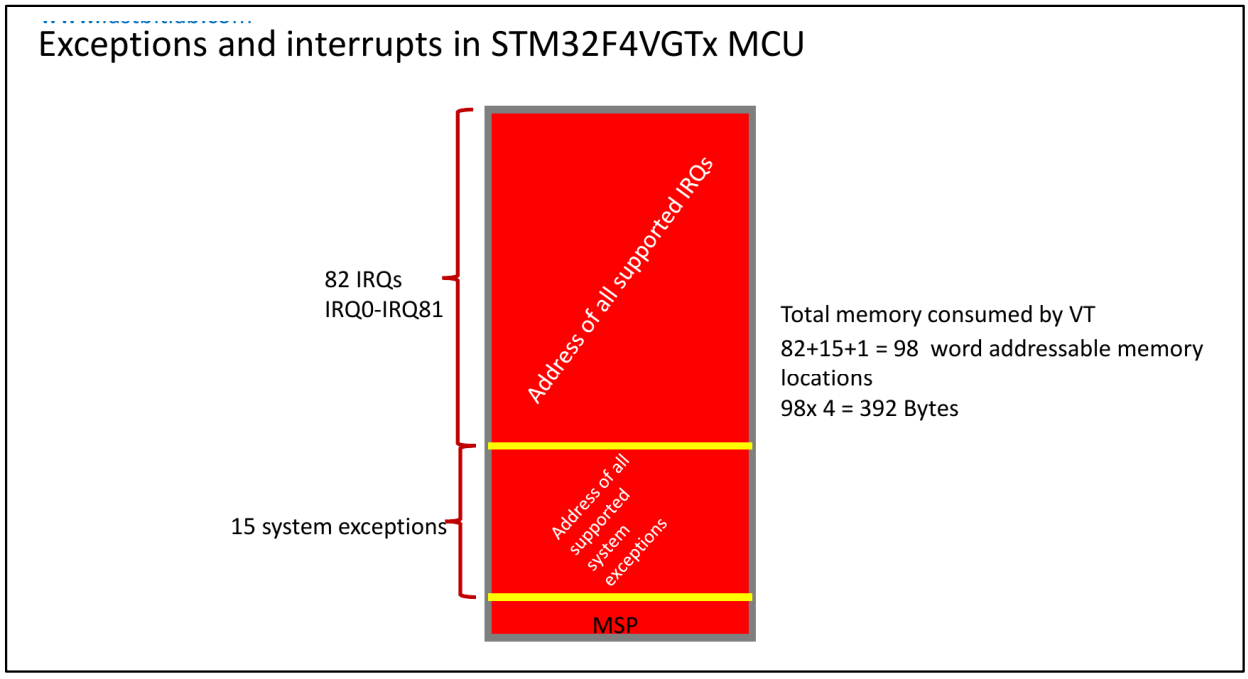
If you want to enable any special peripheral of the processor or target, you can do that in the startup code.

Overview :

1. Create 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( )

Task-1: Create a vector table :





You can extract number of interrupts and exception in Reference manual of your controller.

Table 62. Vector table for STM32F405xx/07xx and STM32F415xx/17xx

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	fixed	Reset	Reset	0x0000 0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	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	PVD	PVD through EXTI line detection interrupt	0x0000 0044
2	9	settable	TAMP_STAMP	Tamper and TimeStamp interrupts through the EXTI line	0x0000 0048
3	10	settable	RTC_WKUP	RTC Wake-up interrupt through the EXTI line	0x0000 004C

MSP

15 system exceptions, 82 IRQs

How to create a vector table ?

Vector table is a collection of addresses.

So create an array to store MSP and handler addresses

Inorder to hold the vector table, we are going to create an array of uint32_t

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

What is this array ?

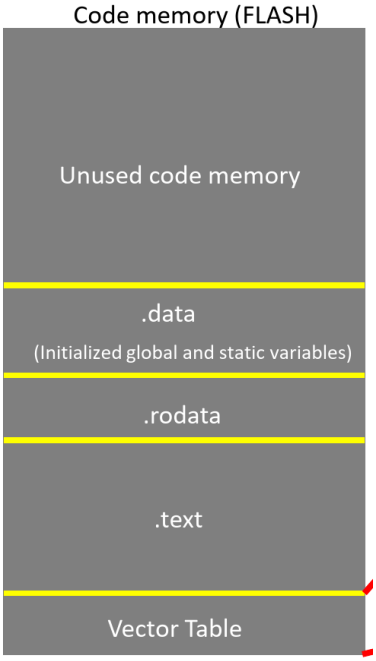
This array is a data.

Data of the program

By default compiler will store initialised data in data section

This should not happen, vector table should not go to .data section.

It should be in the initial addresses.



For that we have to Instruct the compiler not to include the above array in .data section but in a different user defined section

```c
#include <stdint.h>

uint32_t vectors[] = {


};
```

First entry should be stack pointer (MSP value)

We will place the stack at the end of SRAM

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE

#define STACK_START SRAM_END

uint32_t vectors[] = {
    STACK_START,

};
```

As per our vector table, second element is reset handler.

This is not a function pointer

This is a function pointer

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE

#define STACK_START SRAM_END

void Reset_handler(void);

uint32_t vectors[] = {
    STACK_START,
    &Reset_handler,

};

void Reset_handler(void)
{

}
```

You have to typecast.

stm32_startup.c | Makefile

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#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)
{

}
```

stm32_startup.c | Makefile

```
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 $@ $^
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -O0 -o stm32_startup.o stm32_startup.c

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

Our startup file compiled successfully

stm32_startup.c Makefile

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

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make clean
rm -rf *.o *.elf

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ ls
Makefile  led.c  led.h  led_log  main.s  main.c  main.h  main_log  stm32_startup.c

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o main.o main.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o led.o led.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o stm32_startup.o stm32_startup.c

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ ls
Makefile  led.c  led.h  led.o  led_log  main.s  main.c  main.h  main.o  main_log  stm32_startup.c  stm32_startup.o

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

Now this initialized vector array is in .data section,

how do you to tell the compiler to put this in different section ? (user defined section)

That you can do by using the compiler attribute

`section ("section-name")`

Normally, the compiler places the objects it generates in sections like `data` and `bss`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `section` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA")));

main()
{
  /* Initialize stack pointer */
  init_sp (stack + sizeof (stack));

  /* Initialize initialized data */
  memcpy (&init_data, &data, &edata - &data);

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

stm32_startup.c | Makefile

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

void Reset_handler(void);

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

};

void Reset_handler(void)
{

}
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o stm32_startup.o stm32_startup.c

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -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      0000007a  00000000  00000000  00000048  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002e  00000000  00000000  000000c2  2**0
                  CONTENTS, READONLY
```

Our initialized array placed in user defined section

We can also place these attribute in functions also.

stm32_startup.c Makefile

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

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

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

};

void Reset_handler(void)
{

}
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -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      0000007a  00000000  00000000  00000048  2**0
                  CONTENTS, READONLY
  6 .ARM.attributes 0000002e  00000000  00000000  000000c2  2**0
                  CONTENTS, READONLY
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -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
```

The function code is moved to random section

From our vector table, next exception is NMI.

stm32_startup.c Makefile

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

void Reset_Handler(void);

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

};

void NMI_Handler(void)
{

}

void Reset_Handler(void)
{

}
```

Like this we have to write handlers for all 97 exceptions,

instead of that tedious work, we can create a single default handler (common handler) for all the exceptions and allow programmer to implement required handlers as per application requirements.

For that we have to use GCC attribute called Weak and alias.

Weak :

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

alias:

Lets programmer to give alias name for a function

`alias ("target")`

The `alias` variable attribute causes the declaration to be emitted as an alias for another symbol known as an *alias target*. Except for top-level qualifiers the alias target must have the same type as the alias. For instance, the following

```
int var_target;
extern int __attribute__ ((alias ("var_target"))) var_alias;
```

defines `var_alias` to be an alias for the `var_target` variable.

It is an error if the alias target is not defined in the same translation unit as the alias.

Note that in the absence of the attribute GCC assumes that distinct declarations with external linkage denote distinct objects. Using both the alias and the alias target to access the same object is undefined in a translation unit without a declaration of the alias with the attribute.

This attribute requires assembler and object file support, and may not be available on all targets.

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

void Reset_Handler(void);
void NMI_Handler(void) __attribute__((alias("Default_Handler")));

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

};

void Default_Handler(void)
{

}

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.

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

void Reset_Handler(void);
void NMI_Handler(void)                  __attribute__((alias("Default_Handler")));
void HardFault_Handler(void)            __attribute__((alias("Default_Handler")));

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

};

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

void Reset_Handler(void)
{

}
```

Default handler is a dummy function for NMI and HardFault exception

How to implement real functions which handle these exceptions ?

That is done by the programmer

Programmer can override these exception handler in the source file and real implementation can be done for these exceptions in the source file.

For that we have to make those exception handler symbols as weak.

```
void Reset_Handler(void);
void NMI_Handler(void)                  __attribute__((weak, alias("Default_Handler")));
void HardFault_Handler(void)            __attribute__((weak, alias("Default_Handler")));
```

By making these exceptions as a weak symbol, gives control to the programmer to override these functions, programmer can create function with the same name (symbol) without giving any weak attribute. In that case, its overriding these exception handlers.

stm32_startup.c Makefile

```
1
2   #include <stdint.h>
3
4   /*Calculating end of SRAM*/
5   #define SRAM_START 0x20000000U
6   #define SRAM_SIZE  (128 * 1024)  // 128KB
7   #define SRAM_END   ((SRAM_START) + (SRAM_SIZE))
8
9   #define STACK_START SRAM_END
10
11  /* function prototypes of STM32F407x system exception and IRQ handlers */
12
13  void Reset_Handler(void);
14
15  void NMI_Handler                    (void) __attribute__ ((weak, alias("Default_Handler")));
16  void HardFault_Handler              (void) __attribute__ ((weak, alias("Default_Handler")));
17  void MemManage_Handler              (void) __attribute__ ((weak, alias("Default_Handler")));
18  void BusFault_Handler               (void) __attribute__ ((weak, alias("Default_Handler")));
19  void UsageFault_Handler             (void) __attribute__ ((weak, alias("Default_Handler")));
20  void SVC_Handler                    (void) __attribute__ ((weak, alias("Default_Handler")));
21  void DebugMon_Handler               (void) __attribute__ ((weak, alias("Default_Handler")));
```

stm32_startup.c Makefile

```
105
106 uint32_t vectors[] __attribute__((section(".isr_vector"))) = {
107     STACK_START,                        /* Initial stack pointer(Reset) */
108     (uint32_t)&Reset_Handler,           /* Reset Handler */
109     (uint32_t)&NMI_Handler,             /* NMI Handler */
110     (uint32_t)&HardFault_Handler,       /* Hard Fault Handler */
111     (uint32_t)&MemManage_Handler,       /* MPU Fault Handler */
112     (uint32_t)&BusFault_Handler,        /* Bus Fault Handler */
113     (uint32_t)&UsageFault_Handler,      /* Usage Fault Handler */
114     0,                                  /* Reserved */
115     0,                                  /* Reserved */
116     0,                                  /* Reserved */
117     0,                                  /* Reserved */
118     (uint32_t)&SVC_Handler,             /* SVCall Handler */
119     (uint32_t)&DebugMon_Handler,        /* Debug Monitor Handler */
120     0,                                  /* Reserved */
121     (uint32_t)&PendSV_Handler,          /* PendSV Handler */
122     (uint32_t)&SysTick_Handler,         /* SysTick Handler */
123
124     /* Device specific Interrupts start here */
125     (uint32_t)&WWDG_IRQHandler,                 /* Window WatchDog */
126     (uint32_t)&PVD_IRQHandler,                  /* PVD through EXTI Line detection */
127     (uint32_t)&TAMP_STAMP_IRQHandler,            /* Tamper and TimeStamps through the EXTI line */
128     (uint32_t)&RTC_WKUP_IRQHandler,              /* RTC Wakeup through EXTI line */
```

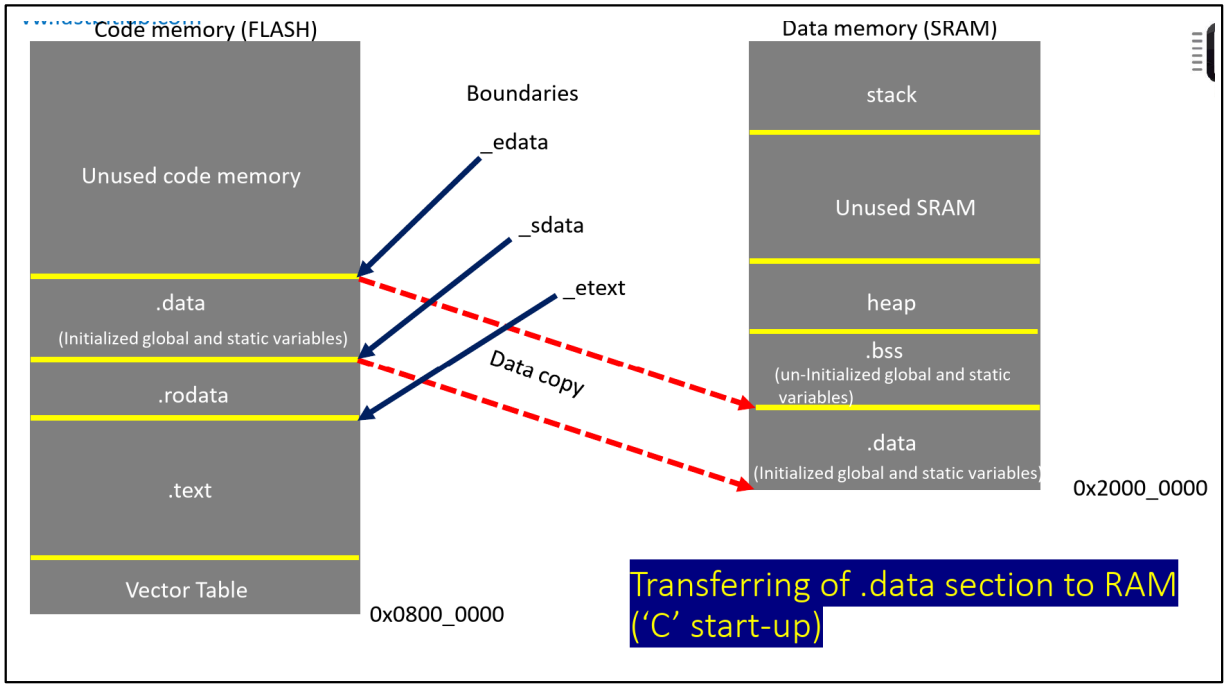
```
203     (uint32_t)&CRYP_IRQHandler,                 /* CRYP crypto global interrupt */
204     (uint32_t)&HASH_RNG_IRQHandler,              /* HASH / RNG global interrupt */
205     (uint32_t)&FPU_IRQHandler                   /* FPU global interrupt */
206 };
207
208 void Default_Handler(void)
209 {
210     while(1);
211 }
212
213 void Reset_Handler(void)
214 {
215
216 }
217
218
```

Task-2 : Write a startup code which initializes .data and .bss section in SRAM:

This job will be implement in the reset handler,

because reset handler is the first function which gets executed when the processor undergoes reset

```
void Reset_Handler(void)
{
    // copy .data section to SRAM

    // Initialize the .bss section to zero in SRAM

    // Call init function of std library (printf)

    // call main()
}
```



Now we have to do the .data copy,

how do you know the boundaries ?

boundary info and size info is required to make data transfer possible

these boundary information have to be exported from linker script to your startup file, then you can able to implement data copy in the startup file

Writing Linker Script from scratch

Linker Scripts :

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

Linker scripts commands :

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

ENTRY command :

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

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ touch stm32_ls.ld

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ ls
Makefile  led.h  led_log  main.c  main.o    stm32_ls.ld     stm32_startup.o
led.c     led.o  main.S   main.h  main_log  stm32_startup.c

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

stm32_ls.ld | stm32_startup.c | Makefile | Rough.txt

```
1  ENTRY(Reset_Handler)
2
3
4
5
6
7
```

This will set the entry point address in the final elf which we are going to generate

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 (relocatable 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

Syntax :

```
MEMORY
{
    name (attr) : ORIGIN = origin, LENGTH = len
}
```

label

defines origin address of the memory region

Defines the length information

Defines name of the memory region which will be later referenced by other parts of the linker script

Syntax :

```
MEMORY
{
name (attr) : ORIGIN = origin, LENGTH = len
}
```

Defines the attribute list of the memory region
Valid attribute lists must be made up of the characters "ALIRWX" that match section attributes

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

Syntax :

```
MEMORY
{
name (attr) : ORIGIN = origin, LENGTH = len
}
```

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

stm32_ls.ld | stm32_startup.c | Makefile | Rough.txt

```
ENTRY(Reset_Handler)

MEMORY
{
    FLASH(rx) :ORIGIN=0x08000000, LENGTH=1024K
    SRAM1(rwx) :ORIGIN=0x20000000, LENGTH=112K
    SRAM2(rwx) :ORIGIN=0x20000000+112K-4, LENGTH=16K
}
```

stm32_ls.ld

```
ENTRY(Reset_Handler)

MEMORY
{
    FLASH(rx) :ORIGIN=0x08000000, LENGTH=1024K
    SRAM(rwx) :ORIGIN=0x20000000, LENGTH=128K
}
```

Sections command :

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

```
/* Sections */
SECTIONS
{

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

         } >(vma) AT>(lma)

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

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

         } >(vma) AT>(lma)

}
```

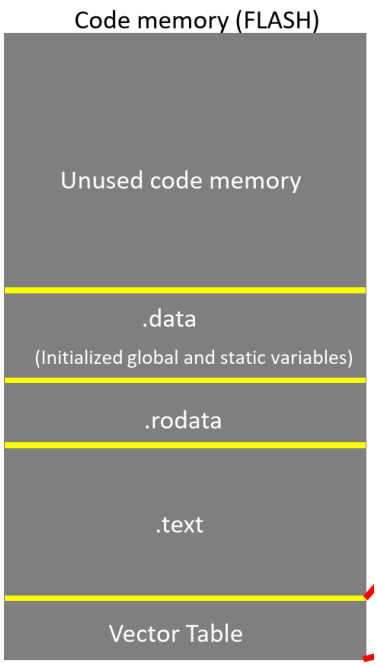
* is wild card character. It just says merge .text section of all input files

stm32_ls.ld

```
1  ENTRY(Reset_Handler)
2
3  MEMORY
4  {
5      FLASH(rx) :ORIGIN=0x08000000, LENGTH=1024K
6      SRAM(rwx) :ORIGIN=0x20000000, LENGTH=128K
7  }
8
9  SECTIONS
10 {
11   .text
12   {
13     // .text of main.o led.o stm32_startup.o
14     *(.text)
15   }
16 }
```

We know our initial code space must contain vector table.

```
stm32_ls.ld
1  ENTRY(Reset_Handler)
2
3  MEMORY
4  {
5      FLASH(rx) :ORIGIN=0x08000000, LENGTH=1024K
6      SRAM(rwx) :ORIGIN=0x20000000, LENGTH=128K
7  }
8
9  SECTIONS
10 {
11   .text
12   {
13     *(.isr_vector)
14     *(.text)
15     *(.rodata)
16   }
17 }
```



This output section comprises of all these individual file sections,

we just created one output section .text,

Now we have to decide, where we wish to store these section in the memory,

that means we have to mention two memory regions for this section,

one is vma (virtual memory address) and another one is lma (load memory address)

Did we relocate this section in our startup code ?

we don't relocate these,

this is going to be stored in the FLASH, it doesn't have any relocatable address

which vma and lma are same for this section

```
SECTIONS
{
  .text
  {
    *(.isr_vector)
    *(.text)
    *(.rodata)
  }> vma AT> lma
}
```

Once the linker sees this,

it generates absolute address for this .text section,

that addresses will fall in the memory region what you mentioned in the vma

After that linker also generates load addresses for this .text section,

that address will fall in the memory region what you mentioned in the AT> command

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

In the above case, both vma and lma are same, so you can write something like this:

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

This means that vma and lma are same

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

  .data :
  {
    *(.data)
  }
}
```

.data section is relocatable isn't it,

this must be stored in FLASH, but the startup code has to copy this section from FLASH to SRAM,

thats why load address is fixed

In FLASH, first .text section will appear, then .data section will appear

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

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

When linker sees this, it will generate load address for this .data section which falls in FLASH

absolute address of this .data section, will fall in the memory region SRAM

this .data section is for initialized

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

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

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

This will not get stored in the FLASH,

thats why it has only VMA,

it doesn't have any LMA

stm32_ls.ld

```
1  ENTRY(Reset_Handler)
2
3  MEMORY
4  {
5      FLASH(rx) :ORIGIN=0x08000000, LENGTH=1024K
6      SRAM(rwx) :ORIGIN=0x20000000, LENGTH=128K
7  }
8
9  SECTIONS
10 {
11   .text :
12   {
13     *(.isr_vector)
14     *(.text)
15     *(.rodata)
16   }> FLASH
17
18   .data :
19   {
20     *(.data)
21   }>SRAM  AT> FLASH
22
23   .bss :
24   {
25     *(.bss)
26   }> SRAM
27 }
```

This linkerScript can create 3 output sections for the final executable

Location counter :

How do you trace boundary information in your C program ?

The boundary information can be tracked using LinkerScript, and then this boundaries can be send to the C program

For that Linker gives you a feature called Location Counter (.)

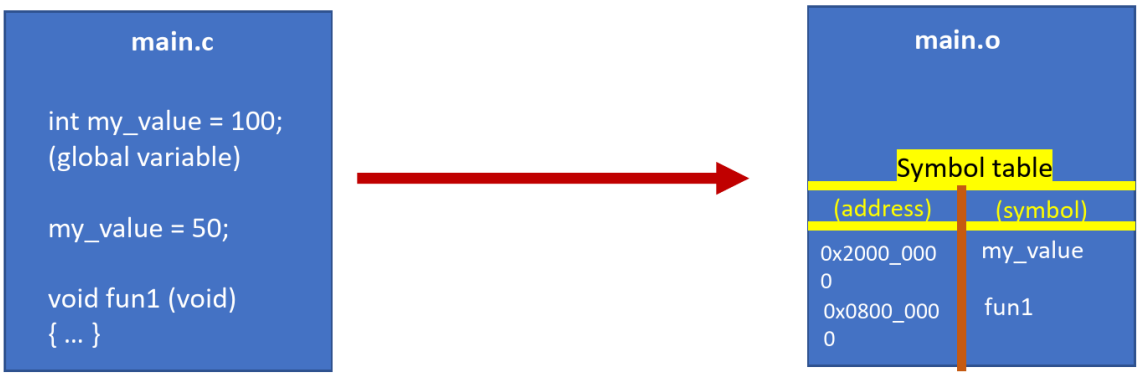
Location counter:

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

We should use Location counter along with Linker Script symbol.

Linker Script Symbols :

- A symbol is the name given for an address
- A symbol declaration is not equivalent to a variable declaration what you do in your ‘C’ application



Now you know we can track the boundary information using Location Counter(.)

How do you store that boundary information in LinkerScript ?

We have to create symbols to store these boundary informations by using Linker Script Symbols.

```
ENTRY(Reset_Handler)

MEMORY
{
  FLASH(rx):ORIGIN=0x08000000,LENGTH=1024K
  SRAM(rwx):ORIGIN=0x20000000,LENGTH=128K
}

__max_heap_size = 0x400;  /* A symbol declaration . Not a variable !!! */
__max_stack_size = 0x200; /* A symbol declaration . Not a variable !!! */

SECTIONS
{
  .text :
  {
    *(.isr_vector)
    *(.text)
    *(.rodata)
    end_of_text = .; /* Store the updated location counter value in to a symbol 'end_of_test' */
  }> FLASH

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

```

This piece of code will be seen by the linker, not by the compiler,

so that 2 size declaration will be seen as symbol declaration, not variable declaration.

When linker sees this entry, its going to add to the symbol table of the final executable

```
ENTRY(Reset_Handler)

MEMORY
{
  FLASH(rx):ORIGIN=0x08000000,LENGTH=1024K
  SRAM(rwx):ORIGIN=0x20000000,LENGTH=128K
}

__max_heap_size = 0x400;  /* A symbol declaration . Not a variable !!! */
__max_stack_size = 0x200; /* A symbol declaration . Not a variable !!! */

SECTIONS
{
  .text :
  {
    *(.isr_vector)
    *(.text)
    *(.rodata)
    end_of_text = .; /* Store the updated location counter value in to a symbol 'end_of_test' */
  }> FLASH

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

```

Here end of text section is stored by using the Location Counter

Note:

- The location counter is incremented by the size of the output section, this will happen automatically by the linker.

At the beginning Location Counter (.) = VMA will be assumed by the Linker.

```
SECTIONS
{
  .text :
  {  . = VMA
    *(.isr_vector)
    *(.text)
    *(.rodata)
    end_of_text = .; /* Store the updated location counter value in to a symbol 'end_of_test' */
  }> FLASH

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

That means . = 0x08000000 (FLASH)

```
SECTIONS
{
  .text :
  {  . = VMA
    *(.isr_vector)
    *(.text)
    *(.rodata)
    end_of_text = .; /* Store the updated location counter value in to a symbol 'end_of_test' */
  }> FLASH

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

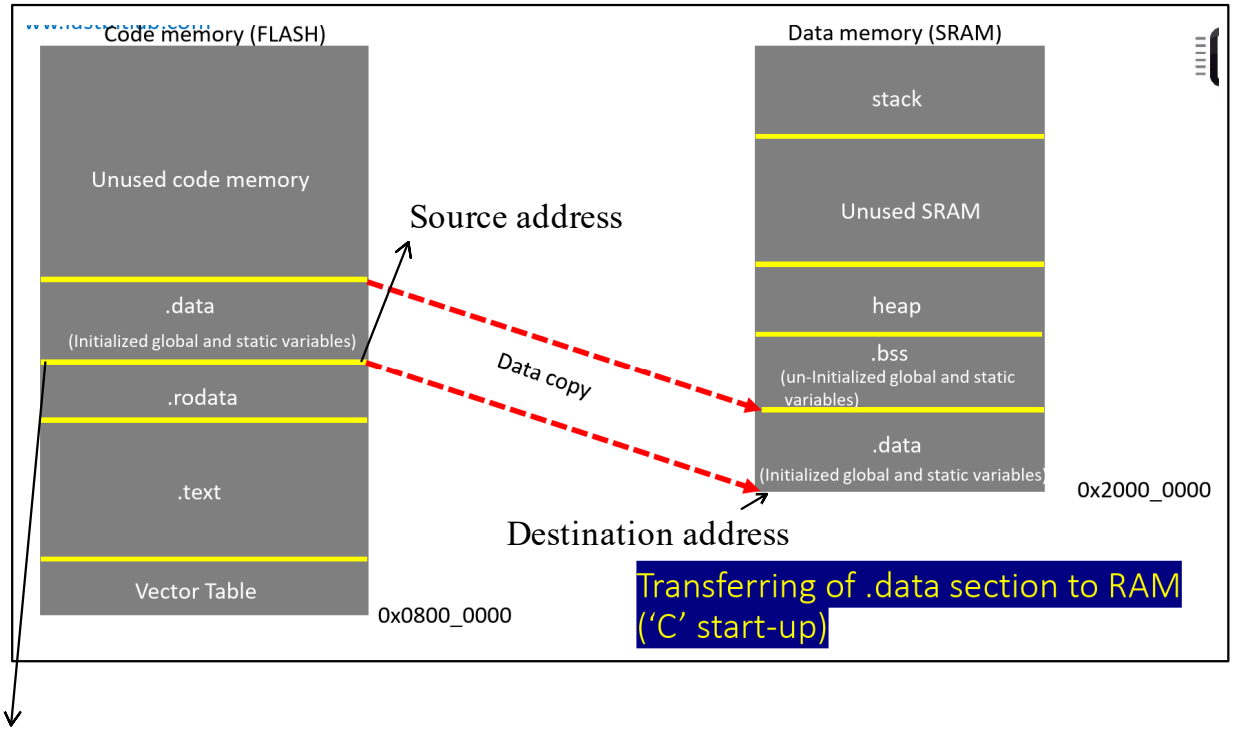
Here location counter value will be incremented by the size of the text section,

thats why end_of_text symbol will now hold end of text section (the address of end of text section),

you get the boundary,

now you can export this symbol to the C program and access the exact address where the text section ends

Similarly you can create any number of linker script symbols and access them in the C program



End of text section

Start of data section

So to make the data copy possible,

we need atleast 3 infomations possible,

Start of data section, size of data section, start of destination address

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

Location counter value is stored in _etext symbol

This location counter currently holds the value of end of text section, so our intention of having text section boundard is achieved

Note :

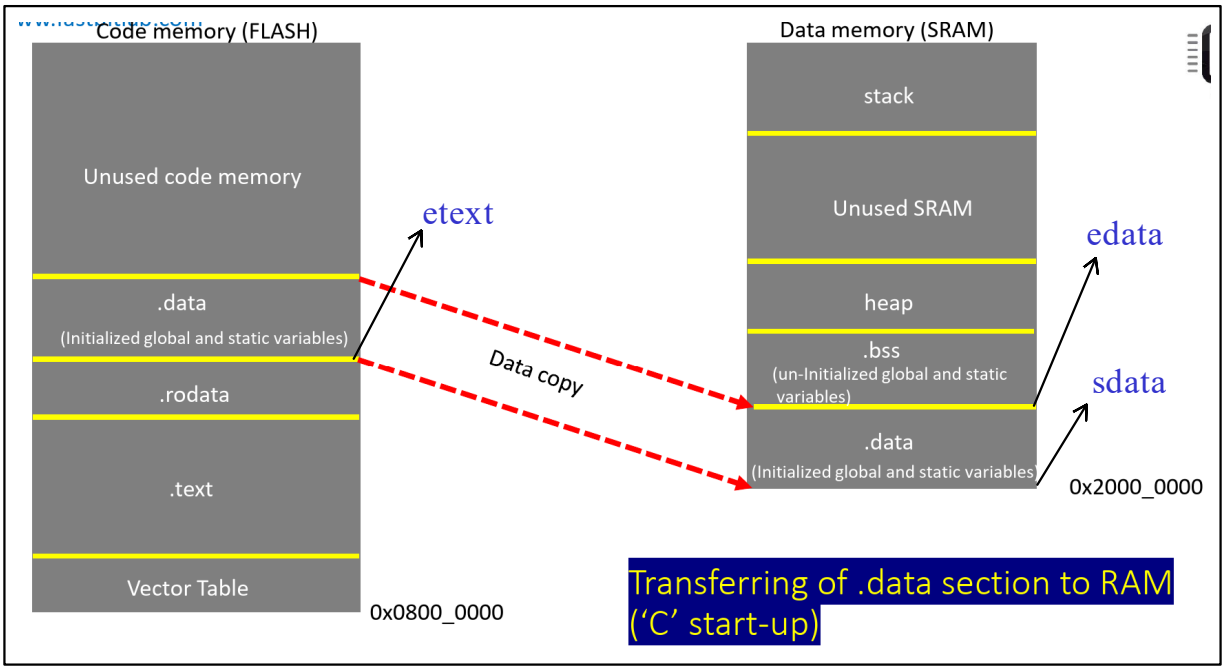
Location counter always tracks VMA of the section in which it is being used

```
  .data :
  {
    _sdata = .;
    *(.data)
  }>SRAM  AT> FLASH
```

Here, the location counter resets to start of VMA of this section

That is SRAM address = 0x20000000

```
.data :
{
  _sdata = .;
  *(.data)
  _edata = .;
}>SRAM  AT> FLASH
```



If we do edata - sdata, we will get the size

Now we know source boundary, source size, destination start,

we can do the data copy now.

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

Linking and Linker Flags :

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-gcc -nostdlib -T stm32_ls.ld *.o -o final.elf
```

This will invoke the linker which will compile this file and link with no standard libraries and also final.elf file will be generated

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-gcc -nostdlib -T stm32_ls.ld *.o -o final.elf
c:/program files (x86)/gnu tools arm embedded/9 2019-q4-major/bin/../lib/gcc/arm-none-eabi/9.2.1/../../../../arm-none-eabi/bin
/ld.exe:stm32_ls.ld:5: syntax error
collect2.exe: error: ld returned 1 exit status
```

stm32_ls.ld

```
1  ENTRY(Reset_Handler)
2
3  MEMORY
4  {
5      FLASH(rx) :ORIGIN=0x08000000, LENGTH=1024K
6      SRAM(rwx) :ORIGIN=0x20000000, LENGTH=128K
7  }
```

Space is needed

```
MEMORY
{
    FLASH(rx) :ORIGIN =0x08000000, LENGTH =1024K
    SRAM(rwx) :ORIGIN =0x20000000, LENGTH =128K
}
```

Compilation successful :

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-gcc -nostdlib -T stm32_ls.ld *.o -o final.elf

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-objdump.exe -h final.elf

final.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000076a  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000001  20000000  0800076a  00020000  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000054  20000004  0800076b  00020004  2**2
                  ALLOC
  3 .comment      00000079  00000000  00000000  00020001  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 0000002e  00000000  00000000  0002007a  2**0
                  CONTENTS, READONLY
```

We can notice our 3 sections, mentioned in the LinkerScript

stm32_ls.ld Makefile

```
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
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o main.o main.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o led.o led.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o stm32_startup.o stm32_startup.c
arm-none-eabi-gcc -nostdlib -T stm32_ls.ld -o final.elf main.o led.o stm32_startup.o

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

Analyzing elf file :

When you generate .elf file, you have to understand various resource allocation,

.elf is the collection of different sections like .text, .data, ..bss etc.. by looking at these sections, you have to make sure that all those sections are indeed placed at the appropriate absolute addresses

How do you analyze .elf file ?

There are tools such as object dump, read elf etc... you can use these or

you can instruct a linker to create a special file called map file.

By using that map file we can analyze various resource alloaction and placement in the memory

Makefile

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

This is a linker argument,

this is supposed to be used with linker command (i.e)
arm-none-eabi ld command

Here we are using compiler command itself to link as well (-gcc)

Linker binary will be present in gcc, but sometimes our map linker argument will not be recongnised by linker driver in gcc, so we have to explicitly mention.

Makefile

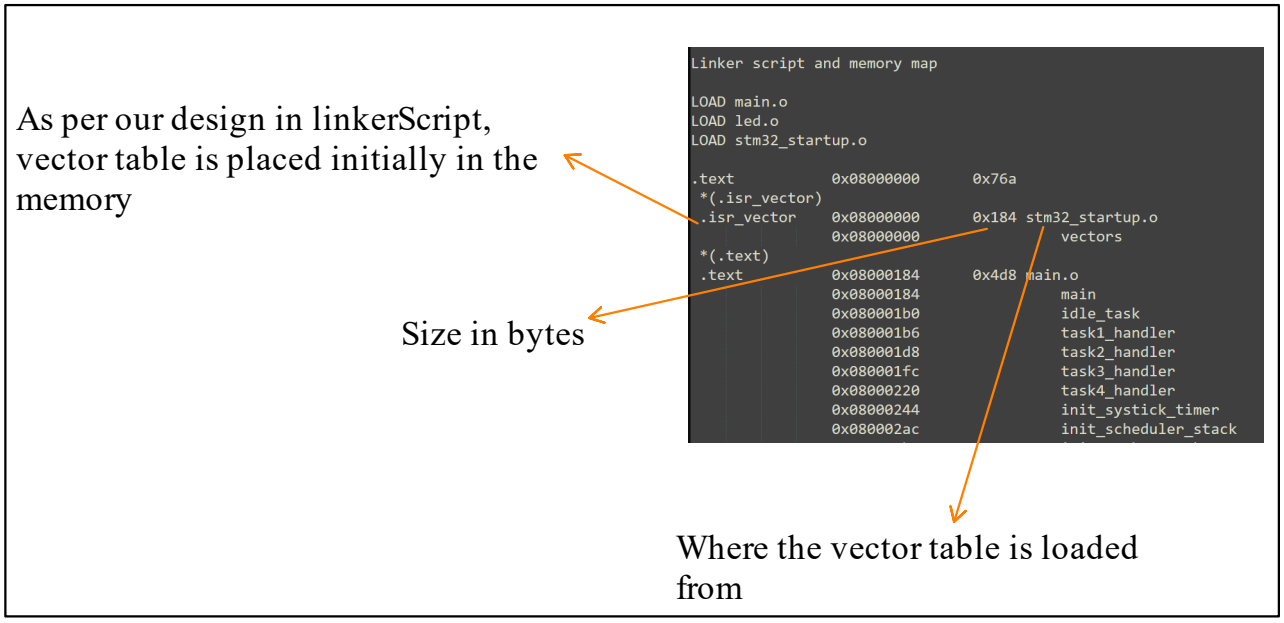
```
CC=arm-none-eabi-gcc
MACH=cortex-m4
CFLAGS= -c -mcpu=$(MACH) -mthumb -std=gnu11 -Wall -O0
LDFLAGS= -nostdlib -T stm32_ls.ld -Wl,-Map=final.map

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
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make clean
rm -rf *.o *.elf

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o main.o main.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o led.o led.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o stm32_startup.o stm32_startup.c
arm-none-eabi-gcc -nostdlib -T stm32_ls.ld -Wl,-Map=final.map -o final.elf main.o led.o stm32_startup.o

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ ls
Makefile   final.map  led.h  led_log  main.c  main.o    stm32_ls.ld      stm32_startup.o
final.elf  led.c      led.o  main.s   main.h  main_log  stm32_startup.c

harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$
```

This is our map file :

final.map

```

Allocating common symbols
Common symbol       size              file

user_tasks          0x50              main.o

Memory Configuration

Name             Origin             Length             Attributes
FLASH            0x08000000         0x00100000         xr
SRAM             0x20000000         0x00020000         xrw
*default*        0x00000000         0xffffffff

Linker script and memory map

LOAD main.o
LOAD led.o
LOAD stm32_startup.o

.text           0x08000000      0x76a
 *(.isr_vector)
 .isr_vector    0x08000000      0x184 stm32_startup.o
                0x08000000                vectors
 *(.text)
 .text          0x08000184      0x4d8 main.o
                0x08000184                main
                0x080001b0                idle_task
                0x080001b6                task1_handler
                0x080001d8                task2_handler
                0x080001fc                task3_handler
                0x08000220                task4_handler
                0x08000244                init_systick_timer
                0x080002ac                init_scheduler_stack
                0x080002b6                init_tasks_stack
```



By looking at this file, we can quickly identify addresses of various functions.

```
*(.rodata)
                0x0800076a                _etext = .
```

There is no constant data in our program.

```
Makefile  main.c
37
38  /* This variable gets updated from systick handler for every systick interrupt */
39  uint32_t g_tick_count = 0;
40
41  const uint32_t v_1 = 100;
42  const uint32_t v_2 = 200;
```

We have added a const data in our main.c

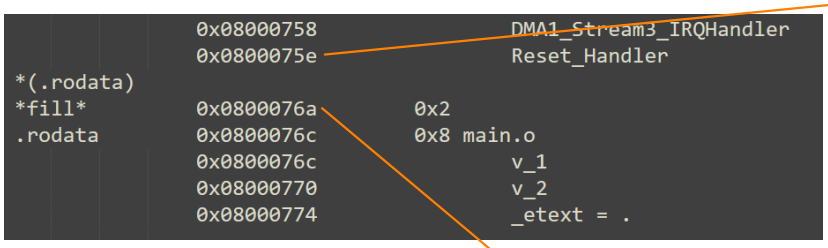
```
*(.rodata)
*fill*          0x0800076a        0x2
.rodata         0x0800076c        0x8 main.o
                0x0800076c                v_1
                0x08000770                v_2
                0x08000774                _etext = .
```

rodata consumes 8 bytes as expected

Our placement of data is successful, which we can see that in a .rodata section of final.map file.

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

In our LinkerScript after .text, .rodata comes



.text section ends here

To make a alignment, there is a padding of 2 bytes

So that rodata can start from aligned address w.r.t word alignment

This alignment is done by the linker automatically

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

  .data :
  {
    _sdata = .;
    *(.data)
    _edata = .;
  }>SRAM  AT> FLASH
```

By our linkerScript design, after .text section, .data section follows

```
*(.rodata)
*fill*          0x0800076a      0x2
.rodata         0x0800076c      0x8 main.o
                0x0800076c              v_1
                0x08000770              v_2
                0x08000774              _etext = .
```

The .data section starts from this address in a FLASH,

which is a load address of the .data section

```
*(.rodata)
*fill*          0x0800076a      0x2
.rodata         0x0800076c      0x8 main.o
                0x0800076c              v_1
                0x08000770              v_2
                0x08000774              _etext = .
```

This also means _etext is the beginning of data section in the code memory.

In this case, .rodata section is word aligned, but its not going to be true for all the time,

you have to maintain the alignment explicitly.

.rodata section is not word aligned

```
*(.rodata)
*fill*          0x0800076a      0x2
.rodata         0x0800076c      0x9 main.o
                0x0800076c              v_1
                0x08000770              v_2
                0x08000774              v_3
                0x08000775              _etext = .
```

Why linker is not padding these spaces ?

Because its end of the .text section or beginning of the new section.

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

Align the location counter

this means you are assigning aligned value of location counter

Whats happening here ?

First the previous value of location counter will be taken,

and that will be aligned to next boundary

The old value of location counter is taken, it will be aligned to the word boundary and the new value will be stored back into the location counter

```
*(.rodata)
*fill*          0x0800076a        0x2
.rodata         0x0800076c        0x9 main.o
                0x0800076c                v_1
                0x08000770                v_2
                0x08000774                v_3
                0x08000778                . = ALIGN (0x4)
*fill*          0x08000775        0x3
                0x08000778                _etext = .
```

Because of the ALIGN command,

padding is added by the linker

now its word aligned

.data section starts with address

```
.data           0x20000000        0x1 load address 0x08000778
                0x20000000                _sdata = .
*(.data)
.data           0x20000000        0x1 main.o
                0x20000000                current_task
.data           0x20000001        0x0 led.o
.data           0x20000001        0x0 stm32_startup.o
                0x20000001                _edata = .
```

It consumes 1 byte

```
/* This variable tracks the current_task being executed on the CPU */
uint8_t current_task = 1; //task1 is running

/* This variable gets updated from systick handler for every systick interrupt */
uint32_t g_tick_count = 0;

const uint32_t v_1 = 100;
const uint32_t v_2 = 200;
const uint8_t v_3 = 200;
```

As per our program,

only one data is going into the data section,

it makes sense it consumes 1 byte

```
.data           0x20000000        0x1 load address 0x08000778
                0x20000000                _sdata = .
*(.data)
.data           0x20000000        0x1 main.o
                0x20000000                current_task
.data           0x20000001        0x0 led.o
.data           0x20000001        0x0 stm32_startup.o
                0x20000001                _edata = .
```

Which is contributed by main.o,

and the exact variable is current_task

```
.data           0x20000000        0x4 load address 0x08000778
                0x20000000                _sdata = .
 *(.data)
 .data          0x20000000        0x1 main.o
                0x20000000                current_task
 .data          0x20000001        0x0 led.o
 .data          0x20000001        0x0 stm32_startup.o
                0x20000004                . = ALIGN (0x4)
 *fill*         0x20000001        0x3
                0x20000004                _edata = .

.igot.plt       0x20000004        0x0 load address 0x0800077c
 .igot.plt      0x20000004        0x0 main.o

.bss            0x20000004       0x54 load address 0x0800077c
                0x20000004                _sbss = .
 *(.bss)
 .bss           0x20000004        0x4 main.o
                0x20000004                g_tick_count
 .bss           0x20000008        0x0 led.o
 .bss           0x20000008        0x0 stm32_startup.o
                0x20000008                _ebss = .
 COMMON         0x20000008       0x50 main.o
                0x20000008                user_tasks
```

```c
/* This variable tracks the current_task being executed on the CPU */
uint8_t current_task = 1; //task1 is running

/* This variable gets updated from systick handler for every systick interrupt */
uint32_t g_tick_count = 0;

const uint32_t v_1 = 100;
const uint32_t v_2 = 200;
const uint8_t v_3 = 200;

/* This is a task control block carries private information of each task */
typedef struct
{
    uint32_t psp_value;
    uint32_t block_count;
    uint8_t  current_state;
    void (*task_handler)(void);
}TCB_t;

/* Each task has its own TCB */
TCB_t user_tasks[MAX_TASKS];
```

goes to .data

goes to .bss

goes to .rodata

goes where ?

The array in our program is having global scope and its uninitialised, so it must go to .bss section,

Analyzing memory map :

```
.data           0x20000000        0x4 load address 0x08000778
                0x20000000                _sdata = .
 *(.data)
 .data          0x20000000        0x1 main.o
                0x20000000                current_task
 .data          0x20000001        0x0 led.o
 .data          0x20000001        0x0 stm32_startup.o
                0x20000004                . = ALIGN (0x4)
 *fill*         0x20000001        0x3
                0x20000004                _edata = .

.igot.plt       0x20000004        0x0 load address 0x0800077c
 .igot.plt      0x20000004        0x0 main.o

.bss            0x20000004       0x54 load address 0x0800077c
                0x20000004                _sbss = .
 *(.bss)
 .bss           0x20000004        0x4 main.o
                0x20000004                g_tick_count
 .bss           0x20000008        0x0 led.o
 .bss           0x20000008        0x0 stm32_startup.o
                0x20000008                _ebss = .
 COMMON         0x20000008       0x50 main.o
                0x20000008                user_tasks
```

Starts here

Ends here

We have missed our global array

That global uninitialised array didn't go to .bss section, it went into COMMON section.

How to include this in .bss section ?

We have to make changes in LinkerScript

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

```
.bss            0x20000004       0x54 load address 0x0800077c
                0x20000004                _sbss = .
 *(.bss)
 .bss           0x20000004        0x4 main.o
                0x20000004                g_tick_count
 .bss           0x20000008        0x0 led.o
 .bss           0x20000008        0x0 stm32_startup.o
 *(COMMON)
 COMMON         0x20000008       0x50 main.o
                0x20000008                user_tasks
                0x20000058                . = ALIGN (0x4)
                0x20000058                _ebss = .
```

Now we have boundaries for the section, we can start data copying and zeroing bss in the reset handler of our startup file

```
void Reset_Handler(void)
{
    // copy .data section to SRAM

    // Initialize the .bss section to zero in SRAM

    // call main()
}
```

Implementing reset handler :

We can access the symbol boundaries in our C program using extern keyword.

If you want to see all the symbols used in a .elf file, we can run the following command :

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-nm.exe final.elf
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-nm.exe final.elf
20000058 B _ebss
20000004 D _edata
08000778 T _etext
20000004 B _sbss
20000000 D _sdata
08000758 W ADC_IRQHandler
08000654 T BusFault_Handler
08000758 W CAN1_RX0_IRQHandler
08000758 W CAN1_RX1_IRQHandler
08000758 W CAN1_SCE_IRQHandler
08000758 W CAN1_TX_IRQHandler
08000758 W CAN2_RX0_IRQHandler
08000758 W CAN2_RX1_IRQHandler
08000758 W CAN2_SCE_IRQHandler
08000758 W CAN2_TX_IRQHandler
08000758 W CRYP_IRQHandler
20000000 D current_task
08000758 W DCMI_IRQHandler
08000758 W DebugMon_Handler
08000758 T Default_Handler
0800065c T delay
```

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-nm.exe final.elf
20000058 B _ebss
20000004 D _edata
08000778 T _etext
20000004 B _sbss
20000000 D _sdata
08000758 W ADC_IRQHandler
08000654 T BusFault_Handler
08000758 W CAN1_RX0_IRQHandler
08000758 W CAN1_RX1_IRQHandler
08000758 W CAN1_SCE_IRQHandler
08000758 W CAN1_TX_IRQHandler
08000758 W CAN2_RX0_IRQHandler
08000758 W CAN2_RX1_IRQHandler
08000758 W CAN2_SCE_IRQHandler
08000758 W CAN2_TX_IRQHandler
08000758 W CRYP_IRQHandler
20000000 D current_task
08000758 W DCMI_IRQHandler
08000758 W DebugMon_Handler
08000758 T Default_Handler
0800065c T delay
```

This is called symbol table maintained by the linker.

Makefile | main.c | final.map | stm32_ls.ld | stm32_startup.c

```c
#include <stdint.h>

/*Calculating end of SRAM*/
#define SRAM_START 0x20000000U
#define SRAM_SIZE  (128 * 1024)  // 128KB
#define SRAM_END   ((SRAM_START) + (SRAM_SIZE))

#define STACK_START SRAM_END

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

```c
void Reset_Handler(void)
{
    // copy .data section to SRAM
    uint32_t size = _edata - _sdata;
    // Initialize the .bss section to zero in SRAM

    // call main()
}
```

If we do this, we get the size,

amount of bytes needs to be transferred

The above implementation is wrong,

which means we are intending to use the value of the symbol,

we know there is no value for that symbol

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-nm.exe final.elf
20000058 B _ebss
20000004 D _edata
08000778 T _etext
20000004 B _sbss
20000000 D _sdata
```

This is a address

This is a name used for this address

There is no value in the address,

we just want that address

```c
void Reset_Handler(void)
{
    // copy .data section to SRAM
    uint32_t size = &_edata - &_sdata;
    // Initialize the .bss section to zero in SRAM

    // call main()
}
```

```c
void Reset_Handler(void)
{
    // copy .data section to SRAM
    uint32_t size = &_edata - &_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++;
    }

    // Initialize the .bss section to zero in SRAM

    // call main()
}
```

```c
void Reset_Handler(void)
{
    // copy .data section to SRAM
    uint32_t size = &_edata - &_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++;
    }

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

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

    // call main()
    main();
}
```

Build is successful :

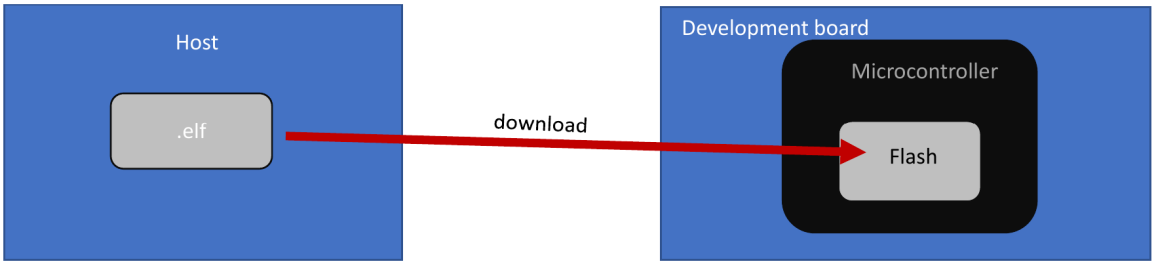
```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make all
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o main.o main.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o led.o led.c
arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o stm32_startup.o stm32_startup.c
arm-none-eabi-gcc -nostdlib -T stm32_ls.ld -Wl,-Map=final.map -o final.elf main.o led.o stm32_startup.o
```

Our next job is to download this elf file into our target for that we have to know about OpenOCD programmer and debugger

OpenOCD and Debug adapters :

We have created our application in the form of elf format,

now we have to download this into the internal FLASH of our controller



There are two ways to do it.

1. Using debug adapter

This is also called as incircuit programming and debugging.

It does protocol conversion, it does host protocol to native target protocol.

OpenOCD :

The Open On-Chip Debugger (OpenOCD) aims to provide debugging, in-system programming, and boundary-scan testing for embedded target devices.

Its is free and opensource host application allows you to program, debug, and analyze your applications using GDB

It supports various target boards based on different processor architecture OpenOCD currently supports many types of debug adapters: USB-based, parallel port-based, and other standalone boxes that run OpenOCD internally

GDB Debug: It allows ARM7 (ARM7TDMI and ARM720t), ARM9 (ARM920T, ARM922T, ARM926EJ–S, ARM966E–S), XScale (PXA25x, IXP42x), Cortex-M3 (Stellaris LM3, ST STM32, and Energy Micro EFM32) and Intel Quark (x10xx) based cores to be debugged via the GDB protocol.

Flash Programming: Flash writing is supported for external CFI-compatible NOR flashes (Intel and AMD/Spansion command set) and several internal flashes (LPC1700, LPC1800, LPC2000, LPC4300, AT91SAM7, AT91SAM3U, STR7x, STR9x, LM3, STM32x, and EFM32). Preliminary support for various NAND flash controllers (LPC3180, Orion, S3C24xx, more) is included

Conclusion :

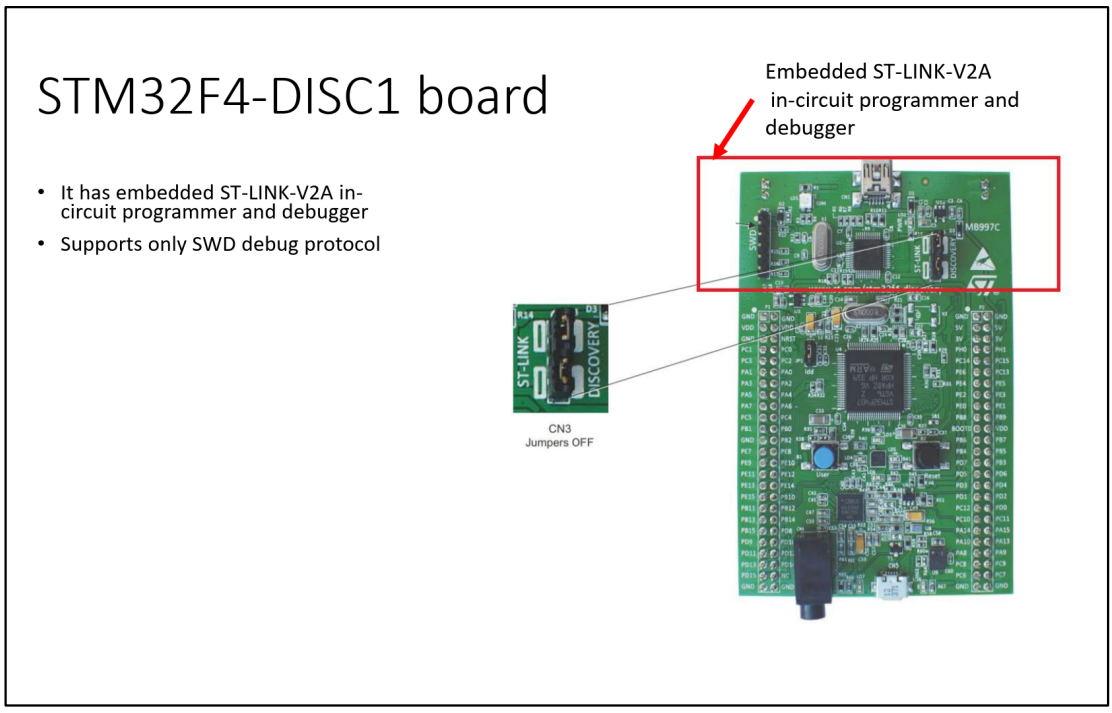
We use OpenOCD for two important purposes :

1. Download our elf into internal FLASH
2. Debug our code using GDB

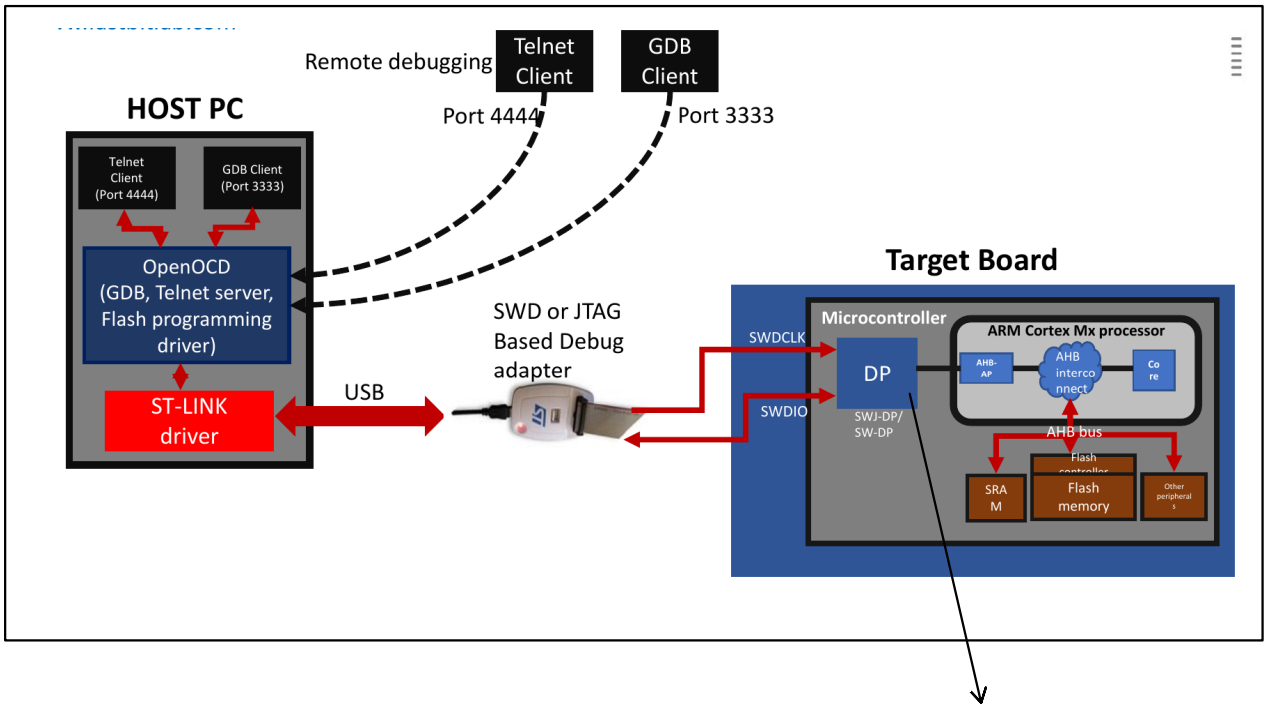
Programming adapters :

- 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

2. Embedded debug adapter on the board



Using JTAG based debug adapter :

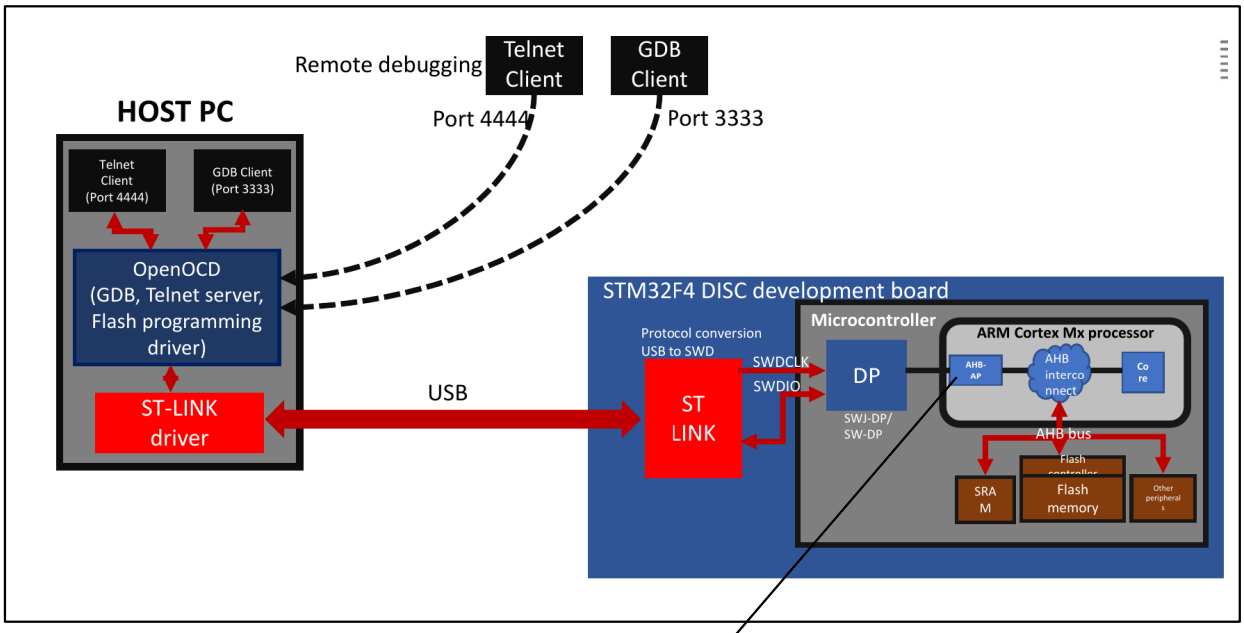


Debug port or debug access port (DP)

This DP block know how to talk to the processor,

your goal is to catch processor bus system which can be achieved by DP port

Using embedded ST link debug adapter :



AP is access point, there are various access point in our processor

In this case our goal is to talk to FLASH and data, so AHB-AP

You can control the core using AP,

controlling the core means resetting, put breakpoints, watchpoints etc...

even you can program code into the SRAM.

Steps to download the code using openOCD :

Download and install OpenOCD

Install Telnet client (for windows you can use PuTTY software)
- If you cannot use Telnet application you can also use “GDB Client”

Run OpenOCD with the board configuration file

Connect to the OpenOCD via Telnet Client or GDB client

Issue commands over Telnet or GDB Client to OpenOCD to download and debug the code

```
17  final.elf:main.o led.o stm32_startup.o
18      $(CC) $(LDFLAGS) -o $@ $^
19  clean:
20      rm -rf *.o *.elf
21
22  load:
23      openocd -f board/stm32f4discovery.cfg
```

After connecting the board with our computer, and running the openOCD, this is the result :

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make load
openocd -f board/stm32f4discovery.cfg
GNU MCU Eclipse OpenOCD, 64-bitOpen On-Chip Debugger 0.10.0+dev-00593-g23ad80df4 (2019-04-22-20:25)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 2000 kHz
Info : STLINK V2J46M31 (API v2) VID:PID 0483:374B
Info : Target voltage: 2.895575
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
```

Now we should be able to communicate with the board using openOCD commands, and those commands we have to issue through the clients

Using GDB Client :

OpenOCD server is running. Now you can connect to this server over various client programs such as telnet client, GDB clients, etc...

We need to open another bash and execute GDB client,

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ arm-none-eabi-gdb.exe
GNU gdb (GNU Tools for Arm Embedded Processors 9-2019-q4-major) 8.3.0.20190709-git
Copyright (C) 2019 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=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
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)
```

GDB client is running

To connect to the openOCD, we have to execute the following :

GDB server :

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target remote localhost:3333
```

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
(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.
0x0800035e in ?? ()
(gdb)
```

OpenOCD client :

```
harik@HarikeswaranPC MINGW64 ~/project_space/Bare_metal_embedded
$ make load
openocd -f board/stm32f4discovery.cfg
GNU MCU Eclipse OpenOCD, 64-bitOpen On-Chip Debugger 0.10.0+dev-00593-g23ad80df4 (2019-04-22-20:25)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 2000 kHz
Info : STLINK V2J46M31 (API v2) VID:PID 0483:374B
Info : Target voltage: 2.895575
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
Info : accepting 'gdb' connection on tcp/3333
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x0800035e msp: 0x2001fff0
Info : device id = 0x100f6413
Info : flash size = 1024 kbytes
Info : flash size = 512 bytes
```

Now the connection is successful

```
(gdb) monitor reset init
```

To differentiate native command and openOCD command

OpenOCD command

```
(gdb) monitor reset init
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000590 msp: 0x20020000
Unable to match requested speed 8000 kHz, using 4000 kHz
Unable to match requested speed 8000 kHz, using 4000 kHz
adapter speed: 4000 kHz
(gdb)
```

Lets load our executable :

```
(gdb) flash write_image erase final.elf
```

You can get these commands from openOCD general commands user guide

There was an error :

GDB client

```
(gdb) flash write_image erase final.elf
Erasing flash memory region at address , size =
Erasing flash memory region at address , size =
Erasing flash memory region at address , size =
0x80000000x100000x80100000x100000x80200000xe0000Error erasing flash with vFlashErase packet
(gdb)
```

openOCD server

```
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0xfffffffe msp: 0xfffffffc
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
adapter speed: 4000 kHz
Error: Cannot erase OTP memory
Error: failed erasing sectors 0 to 15
Error: flash_erase returned -4
```

GDB client :

```
(gdb) monitor flash write_image erase final.elf
auto erase enabled
wrote 16384 bytes from file final.elf in 0.593622s (26.953 KiB/s)
(gdb)
```

openOCD server :

```
auto erase enabled
wrote 16384 bytes from file final.elf in 0.593622s (26.953 KiB/s)
```

```
(gdb) monitor reset halt
```

It will reset the microcontroller and will immediately halt it

GDB client :

```
(gdb) monitor reset halt
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800075e msp: 0x20020000
(gdb)
```

openOCD server :

```
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800075e msp: 0x20020000
```

To run the program, you have to issue :

```
(gdb) monitor resume
```

Halting the execution :

```
(gdb) monitor halt
target halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x0800066c psp: 0x2001f3e0
```

This will stop the execution right, where the LED is glowing

Resetting :

```
(gdb) monitor reset
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
(gdb)
```

This will reset the controller, and program will start executing from the start.

If you want to reset and halt the processor :

```
(gdb) monitor reset halt
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800075e msp: 0x20020000
```

Then you can resume the program using :

```
(gdb) monitor resume
(gdb)
```

Verifying some memory location value :

```
.data           0x20000000        0x4 load address 0x080007f4
                0x20000000                _sdata = .
*(.data)
.data           0x20000000        0x1 main.o
                0x20000000                current_task
.data           0x20000001        0x0 led.o
.data           0x20000001        0x0 stm32_startup.o
                0x20000004                . = ALIGN (0x4)
*fill*          0x20000001        0x3
                0x20000004                _edata = .
```

Lets verify, there is data in this memory location

We have to read from this memory location using openOCD

If you want to read targets memory location, then you need to read :

15.3 Memory access commands

These commands allow accesses of a specific size to the memory system. Often these are used to configure the current target in some special way. For example - one may need to write certain values to the SDRAM controller to enable SDRAM.

1. Use the `targets` (plural) command to change the current target.
2. In system level scripts these commands are deprecated. Please use their TARGET object siblings to avoid making assumptions about what TAP is the current target, or about MMU configuration.

Command: **mdd** *[phys] addr [count]*
Command: **mdw** *[phys] addr [count]*
Command: **mdh** *[phys] addr [count]*
Command: **mdb** *[phys] addr [count]*

Display contents of address *addr*, as 64-bit doublewords (`mdd`), 32-bit words (`mdw`), 16-bit halfwords (`mdh`), or 8-bit bytes (`mdb`). When the current target has an MMU which is present and active, *addr* is interpreted as a virtual address. Otherwise, or if the optional *phys* flag is specified, *addr* is interpreted as a physical address. If *count* is specified, displays that many units. (If you want to process the data instead of displaying it, see the `read_memory` primitives.)

Command: **mwd** *[phys] addr doubleword [count]*
Command: **mww** *[phys] addr word [count]*
Command: **mwh** *[phys] addr halfword [count]*
Command: **mwb** *[phys] addr byte [count]*

Writes the specified *doubleword* (64 bits), *word* (32 bits), *halfword* (16 bits), or *byte* (8-bit) value, at the specified address *addr*. When the current target has an MMU which is present and active, *addr* is interpreted as a virtual address. Otherwise, or if the optional *phys* flag is specified, *addr* is interpreted as a physical address. If *count* is specified, fills that many units of consecutive address.

```
(gdb) monitor mdw 0x20000000
```

Reading 4 words

```
(gdb) monitor mdw 0x20000000 4
0x20000000: 20050601 001294f4 2001efc0 00000000
(gdb)
```

Value of current task variable

In our main.c, the variable value is different,

```
Makefile | main.c | final.map | stm32_ls.ld | stm32_startup.c
34
35  /* This variable tracks the current_task being executed on the CPU */
36  uint8_t current_task = 1; //task1 is running
```

This suggest that, there is something wrong,

that is our data copy from FLASH to SRAM was wrong.

```
void Reset_Handler(void)
{
    // copy .data section to SRAM
    uint32_t size = &_edata - &_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++;
    }
```

Here our subtraction is wrong, this is a pointer subtraction, we need to typecast it to value.

```
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++;
    }
```

After changing the code and reflashing it to the FLASH, the data is now correct :

```
(gdb) monitor mdw 0x20000000 4
0x20000000: 00000003 00003e62 2001efc0 00000000
(gdb)
```

To keep a breakpoint we can use the following, placing the breakpoint in task1_handler,

```
*(.text)
.text          0x08000184      0x4d8 main.o
               0x08000184              main
               0x080001b0              idle_task
               0x080001b6              task1_handler
               0x080001d8              task2_handler
```

```
(gdb) monitor bp 0x080001b6 2 hw
breakpoint set at 0x080001b6
(gdb)
```

hardware breakpoint

length

```
(gdb) monitor resume
target halted due to breakpoint, current mode: Thread
xPSR: 0x01000000 pc: 0x080001b6 psp: 0x2001ffc0
(gdb)
```

If we read the currest_task variable now, we can see the original value before manipulation,

```
(gdb) monitor mdw 0x20000000 4
0x20000000: 00000001 00000000 2001efc0 00000000
(gdb)
```

So our data copy is successful

We can resume the breakpoint,

```
(gdb) monitor rbp 0x080001b6
(gdb) monitor bp
(gdb)
```

Setting the breakpoint at task4_handler and checking the variable data,

```
*(.text)
.text          0x08000184      0x4d8 main.o
               0x08000184                main
               0x080001b0                idle_task
               0x080001b6                task1_handler
               0x080001d8                task2_handler
               0x080001fc                task3_handler
               0x08000220                task4_handler
```

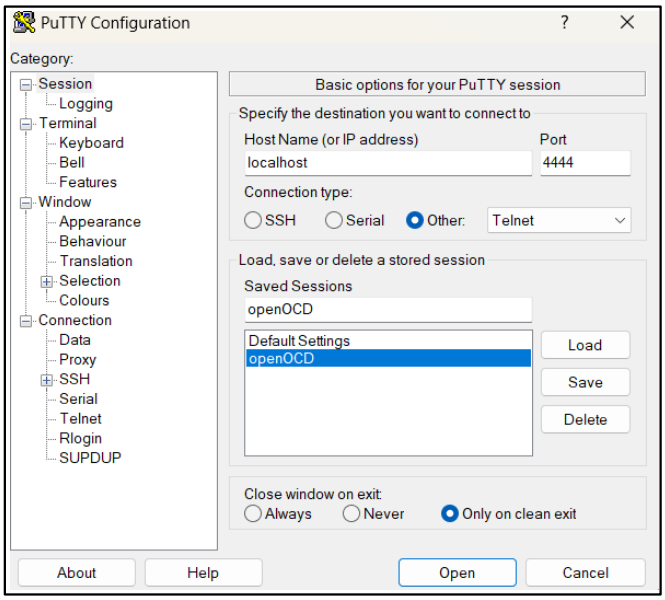
```
(gdb) monitor bp 0x08000220 2 hw
breakpoint set at 0x08000220
(gdb) monitor bp
Breakpoint(IVA): 0x08000220, 0x2, 1
(gdb)
```

```
(gdb) monitor reset
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
(gdb) monitor bp
Breakpoint(IVA): 0x08000220, 0x2, 1
(gdb) monitor mdw 0x20000000 4
0x20000000: 00000004 00000003 2001efc0 00000000
(gdb)
```

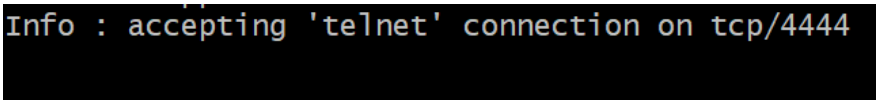
Yes the value is getting updated

Telnet client :

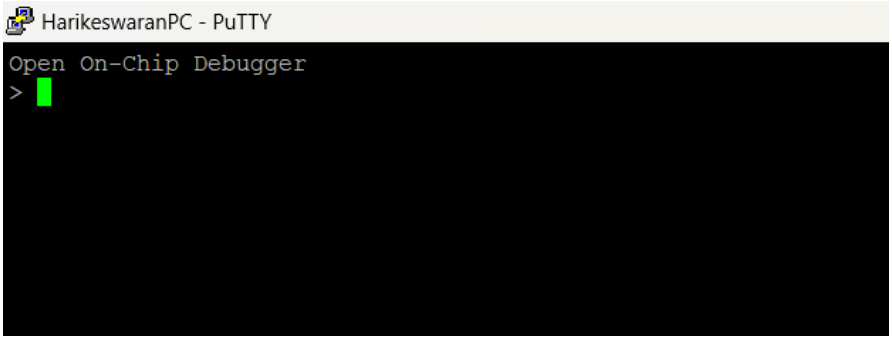
PuTTY software :



OpenOCD server :

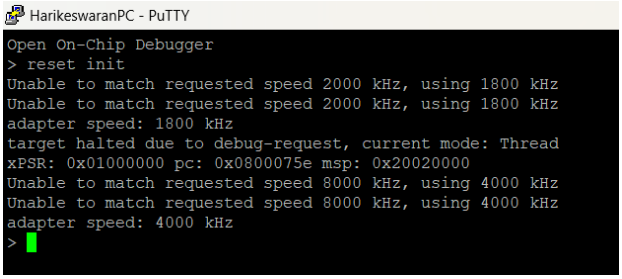


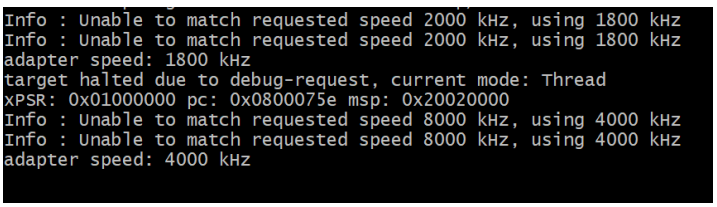
Telnet client :



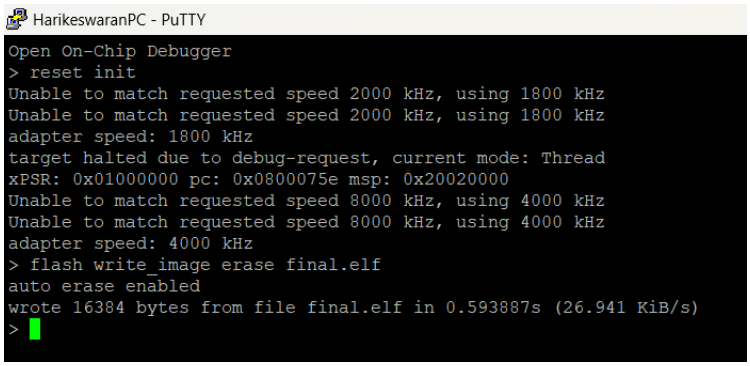
Same commands can be used here, without using the 1st word monitor

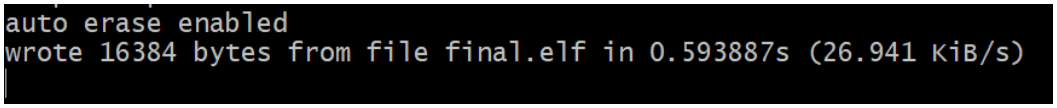
Example-1 :





Example - 2 :


HarikeswaranPC - 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
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800075e msp: 0x20020000
Unable to match requested speed 8000 kHz, using 4000 kHz
Unable to match requested speed 8000 kHz, using 4000 kHz
adapter speed: 4000 kHz
> flash write_image erase final.elf
auto erase enabled
wrote 16384 bytes from file final.elf in 0.593887s (26.941 KiB/s)
>


auto erase enabled
wrote 16384 bytes from file final.elf in 0.593887s (26.941 KiB/s)