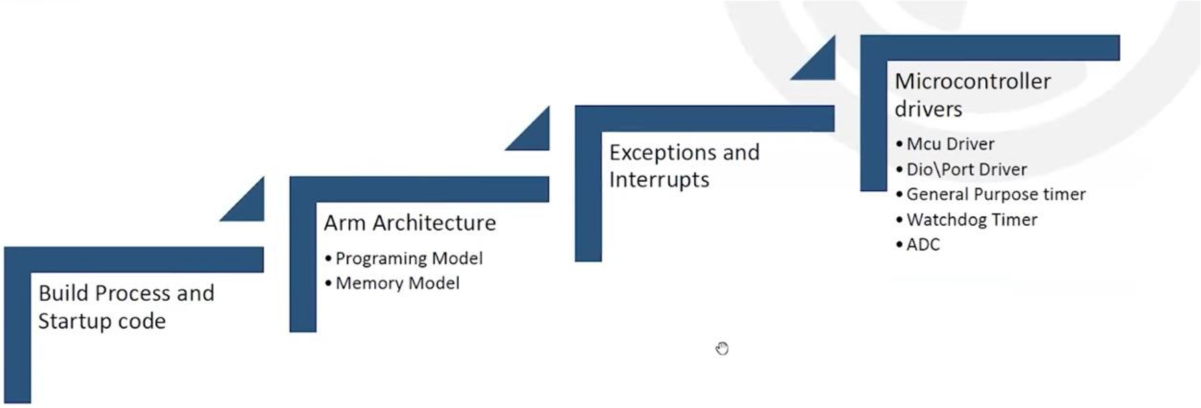
# Course Roadmap



Build Process and Startup code

Arm Architecture

- Programing Model
- Memory Model

Exceptions and Interrupts

Microcontroller drivers

- Mcu Driver
- Dio\Port Driver
- General Purpose timer
- Watchdog Timer
- ADC

# What is ARM?

- Advanced RISC Machine
- low-power processor
- low interrupt latency
- low-cost debug.

## ARM INSTRUCTION SET VS. THUMB INSTRUCTION SET

- Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model.
- On execution, 16-bit Thumb instructions are transparently decompressed to full 32-bit ARM instructions in real time, without performance loss.
- Thumb code is typically 65% of the size of ARM code.
- The availability of both 16-bit Thumb and 32-bit ARM instruction sets gives designers the flexibility to emphasize performance or code size on a subroutine level, according to the requirements of their applications.
  - For example, critical loops for applications such as fast interrupts and DSP algorithms can be coded using the full ARM instruction set then linked with Thumb code.

ARMv7-M only supports execution of Thumb instructions.

## ARMV7 ARCHITECTURE PROFILES.

### CORTEX-A

**The application profile** for systems supporting the ARM and Thumb instruction sets, and *requiring virtual address* support in the memory management model.

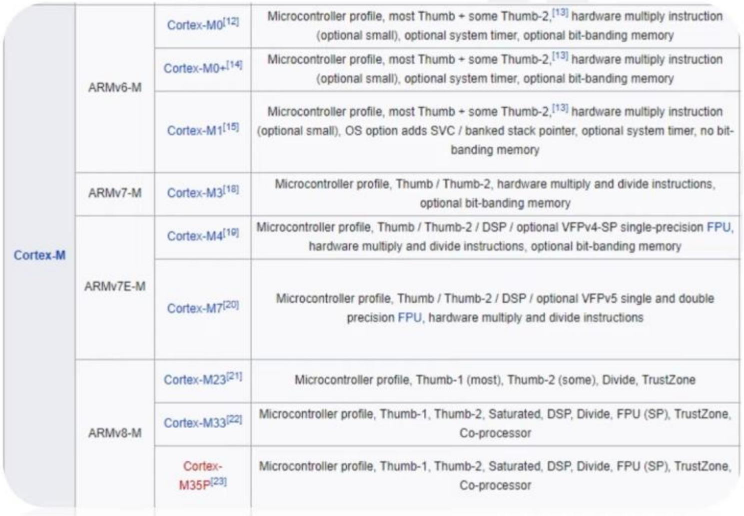
### CORTEX-R

**The real-time profile** for systems supporting the *ARM and Thumb* instruction sets, and requiring physical address only support in the memory management model

### CORTEX-M

**The micro controller profile** for systems supporting only the Thumb instruction set, and where *overall size and deterministic operation* for an implementation are more important than absolute performance.

## CORTEX-M ARCHITECTURE



| | | | |
|---|---|---|---|
| Cortex-M | ARMv6-M | Cortex-M0[12] | Microcontroller profile, most Thumb + some Thumb-2,[13] hardware multiply instruction (optional small), optional system timer, optional bit-banding memory |
| | | Cortex-M0+[14] | Microcontroller profile, most Thumb + some Thumb-2,[13] hardware multiply instruction (optional small), optional system timer, optional bit-banding memory |
| | | Cortex-M1[15] | Microcontroller profile, most Thumb + some Thumb-2,[13] hardware multiply instruction (optional small), OS option adds SVC / banked stack pointer, optional system timer, no bit-banding memory |
| | ARMv7-M | Cortex-M3[18] | Microcontroller profile, Thumb / Thumb-2, hardware multiply and divide instructions, optional bit-banding memory |
| | ARMv7E-M | Cortex-M4[19] | Microcontroller profile, Thumb / Thumb-2 / DSP / optional VFPv4-SP single-precision FPU, hardware multiply and divide instructions, optional bit-banding memory |
| | | Cortex-M7[20] | Microcontroller profile, Thumb / Thumb-2 / DSP / optional VFPv5 single and double precision FPU, hardware multiply and divide instructions |
| | ARMv8-M | Cortex-M23[21] | Microcontroller profile, Thumb-1 (most), Thumb-2 (some), Divide, TrustZone |
| | | Cortex-M33[22] | Microcontroller profile, Thumb-1, Thumb-2, Saturated, DSP, Divide, FPU (SP), TrustZone, Co-processor |
| | | Cortex-M35P[23] | Microcontroller profile, Thumb-1, Thumb-2, Saturated, DSP, Divide, FPU (SP), TrustZone, Co-processor |

## ARM CORTEX-M PROCESSOR FAMILY

### CORTEX-M0

- *Uses the Armv6-M (only supports 16-bit thumb instructions)*

### CORTEX-M0+

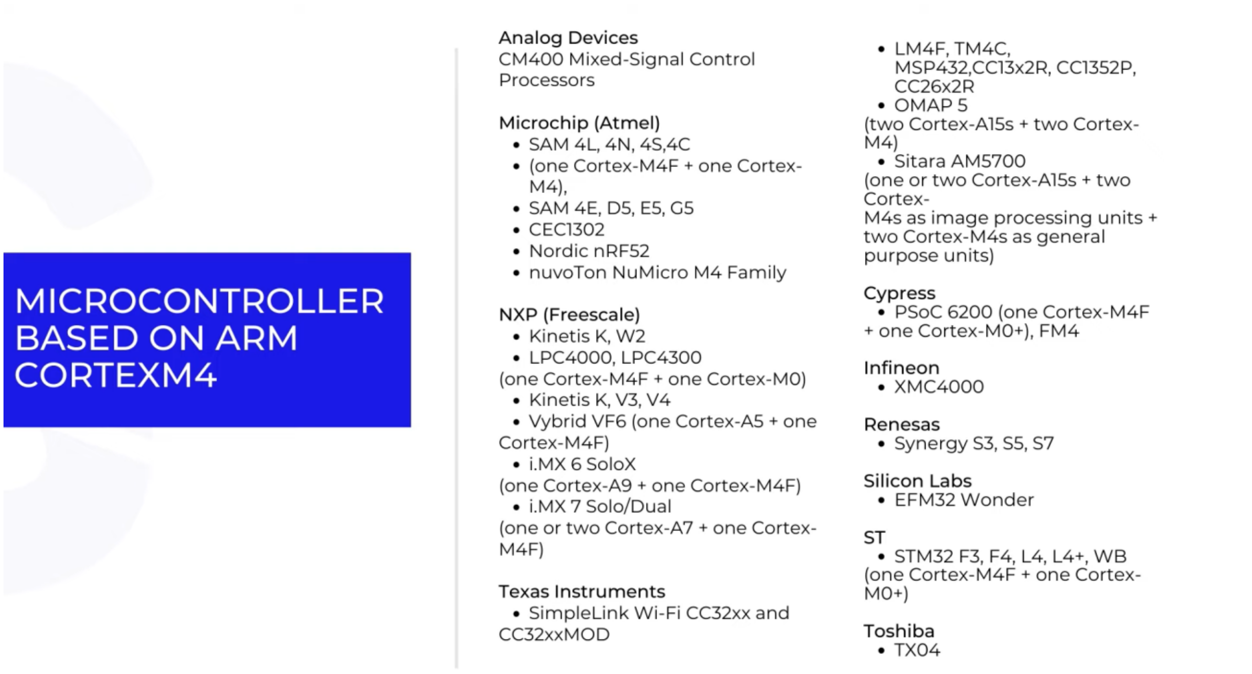
- *Uses the Armv7-M which supports the Thumb2 instruction set (16-bit + 32- bit instructions)*

### CORTEX-M3

- *Richer instruction set*
- *architecture*
- *Harvard Architecture*
- *Write buffer*
- *Fewer Interrupt latency cycles*

### CORTEX-M4

- *Capability for DSP*



## MICROCONTROLLER BASED ON ARM CORTEXM4

**Analog Devices**
CM400 Mixed-Signal Control Processors

**Microchip (Atmel)**

- SAM 4L, 4N, 4S,4C
- (one Cortex-M4F + one Cortex-M4),
- SAM 4E, D5, E5, G5
- CEC1302
- Nordic nRF52
- nuvoTon NuMicro M4 Family

**NXP (Freescale)**

- Kinetis K, W2
- LPC4000, LPC4300

(one Cortex-M4F + one Cortex-M0)

- Kinetis K, V3, V4
- Vybrid VF6 (one Cortex-A5 + one Cortex-M4F)
- i.MX 6 SoloX

(one Cortex-A9 + one Cortex-M4F)

- i.MX 7 Solo/Dual

(one or two Cortex-A7 + one Cortex-M4F)

**Texas Instruments**

- SimpleLink Wi-Fi CC32xx and CC32xxMOD
- LM4F, TM4C, MSP432,CC13x2R, CC1352P, CC26x2R
- OMAP 5

(two Cortex-A15s + two Cortex-M4)

- Sitara AM5700

(one or two Cortex-A15s + two Cortex-M4s as image processing units + two Cortex-M4s as general purpose units)

**Cypress**

- PSoC 6200 (one Cortex-M4F + one Cortex-M0+), FM4

**Infineon**

- XMC4000

**Renesas**

- Synergy S3, S5, S7

**Silicon Labs**

- EFM32 Wonder

**ST**

- STM32 F3, F4, L4, L4+, WB

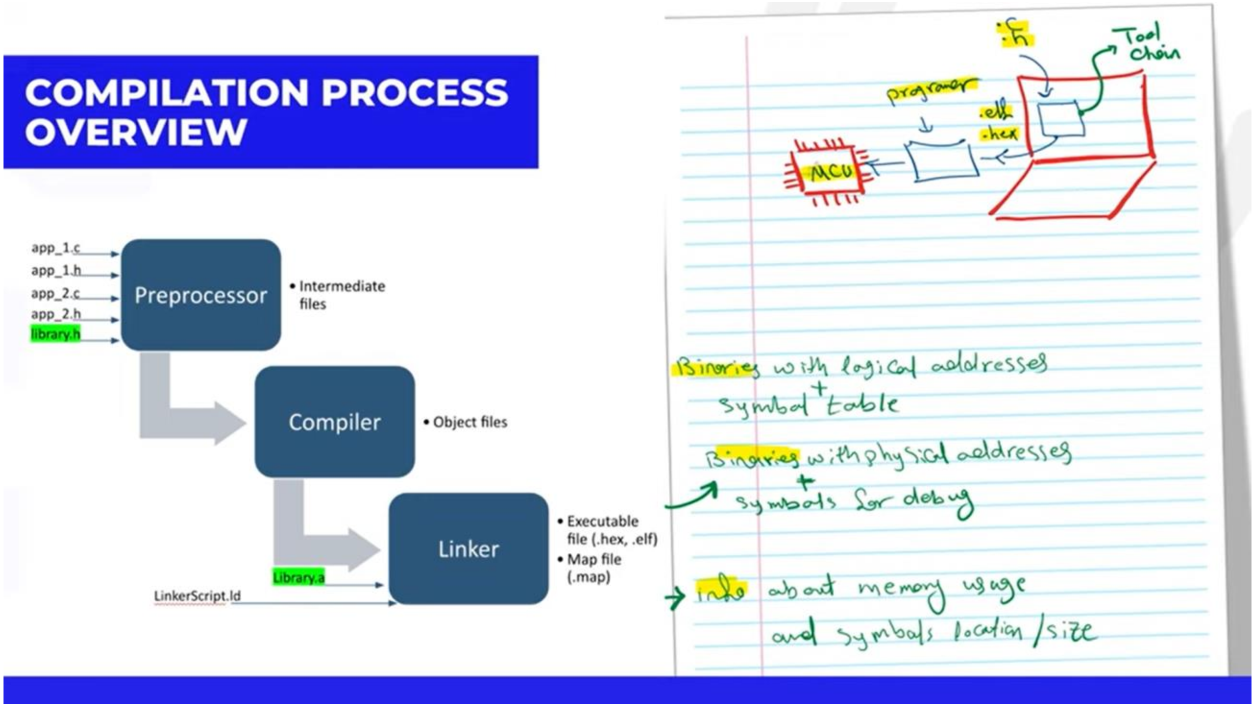
(one Cortex-M4F + one Cortex-M0+)

**Toshiba**

- TX04

## BUILD PROCESS

GNU ARM TOOL CHAIN

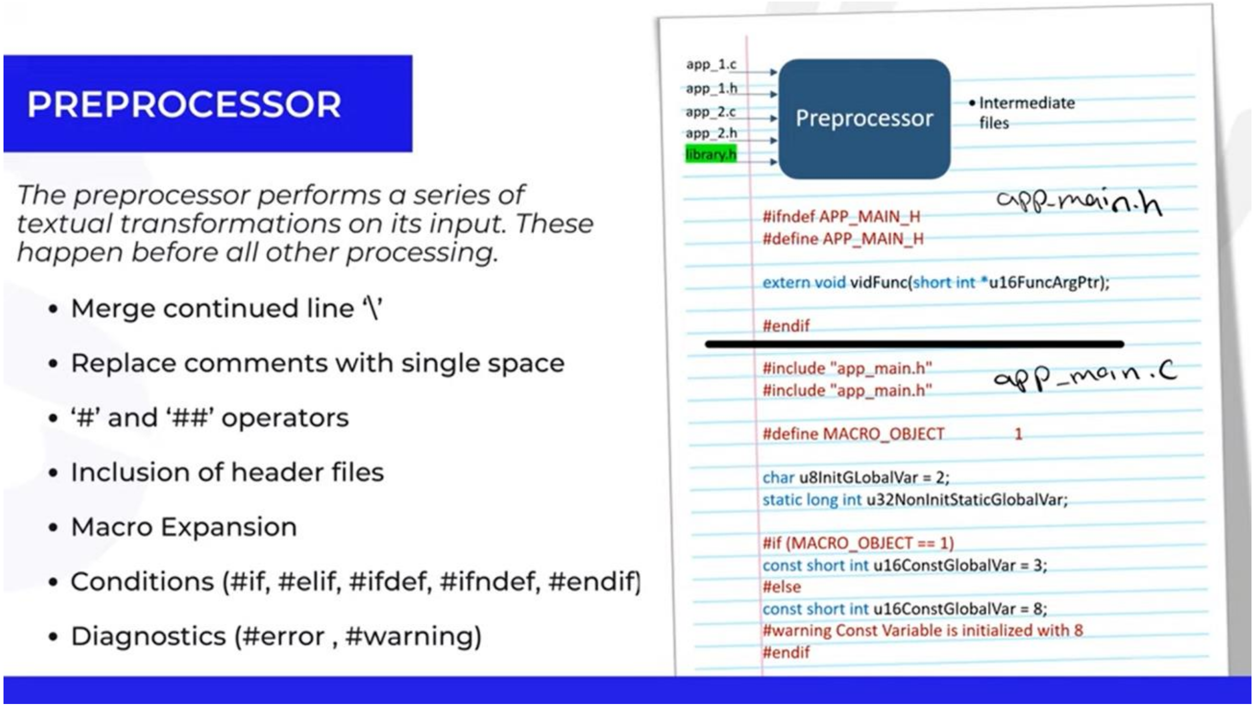


## COMPILATION PROCESS OVERVIEW

Binaries with logical addresses
+
symbol table

Binaries with physical addresses
+
symbols for debug

info about memory usage
and symbols location /size



## PREPROCESSOR

*The preprocessor performs a series of textual transformations on its input. These happen before all other processing.*

- Merge continued line '\'
- Replace comments with single space
- '#' and '##' operators
- Inclusion of header files
- Macro Expansion
- Conditions (#if, #elif, #ifdef, #ifndef, #endif)
- Diagnostics (#error , #warning)

app_1.c
app_1.h
app_2.c
app_2.h
library.h
Preprocessor
• Intermediate files

app-main.h

```
#ifndef APP_MAIN_H
#define APP_MAIN_H

extern void vidFunc(short int *u16FuncArgPtr);

#endif
```

app_main.c

```
#include "app_main.h"
#include "app_main.h"

#define MACRO_OBJECT          1

char u8InitGLobalVar = 2;
static long int u32NonInitStaticGlobalVar;

#if (MACRO_OBJECT == 1)
const short int u16ConstGlobalVar = 3;
#else
const short int u16ConstGlobalVar = 8;
#warning Const Variable is initialized with 8
#endif
```

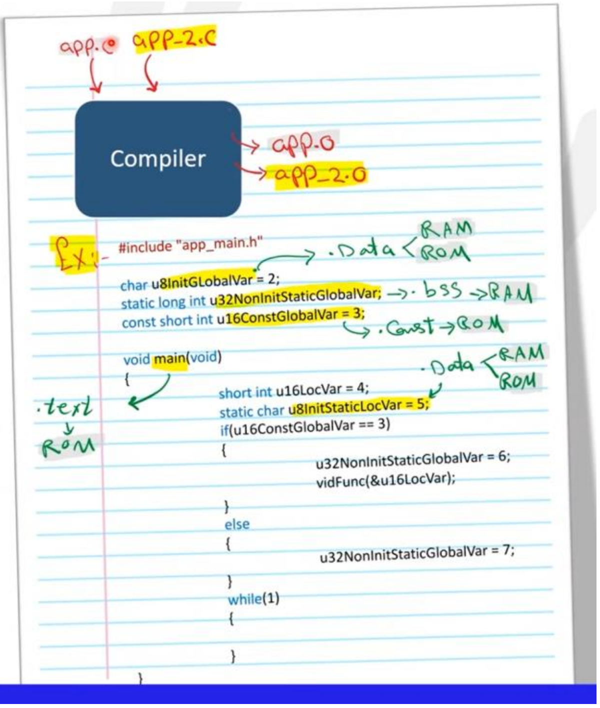
# COMPILER

- Check syntax
- Allocate Symbols to logical addresses according to memory sections

Symbols types

- Global variables ➔.bss \ .data (initialized)
- Static variables ➔ .bss \ .data (not initialized)
- Functions ➔.text

- Convert C Code into binary according to ISA
- Assembler stage to convert assembly into binary
- Code Optimization



```
#include "app_main.h"

char u8InitGLobalVar = 2;
static long int u32NonInitStaticGlobalVar;
const short int u16ConstGlobalVar = 3;

void main(void)
{
        short int u16LocVar = 4;
        static char u8InitStaticLocVar = 5;
        if(u16ConstGlobalVar == 3)
        {
                u32NonInitStaticGlobalVar = 6;
                vidFunc(&u16LocVar);

        }
        else
        {
                u32NonInitStaticGlobalVar = 7;

        }
        while(1)
        {

        }
}
```

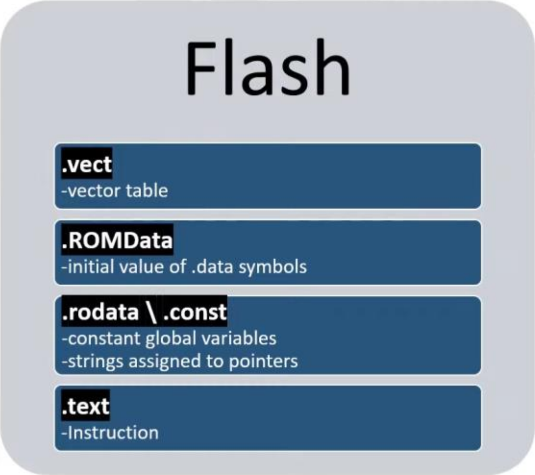
# COMPILER

- Check syntax
- Allocate Symbols to logical addresses according to memory sections

Symbols types

- Global variables ➔.bss \ .data (initialized)
- Static variables ➔ .bss \ .data (not initialized)
- Functions ➔.text

- Convert C Code into binary according to ISA
- Assembler stage to convert assembly into binary
- Code Optimization

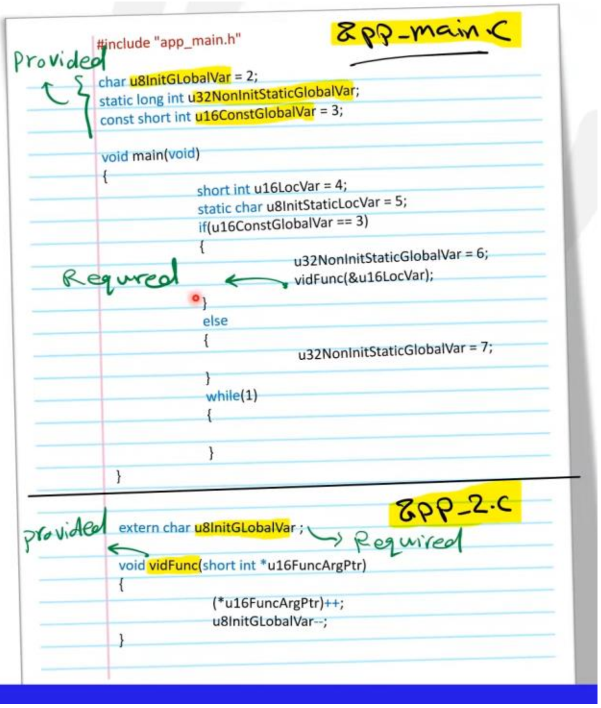

Flash
.vect
-vector table
.ROMData
-initial value of .data symbols
.rodata \ .const
-constant global variables
-strings assigned to pointers
.text
-Instruction

## OBJECT FILES AND SYMBOL TABLE

- An object file is machine code (binary) that has info allows the linker to work.
- The info contains

Symbols (Provided \ Required).

- *Name and value (if exist)*
- *Which memory section located in*
- *Logical address (Offset from the start of section)*
- *Size*
- *Line number (for debugging purpose)*

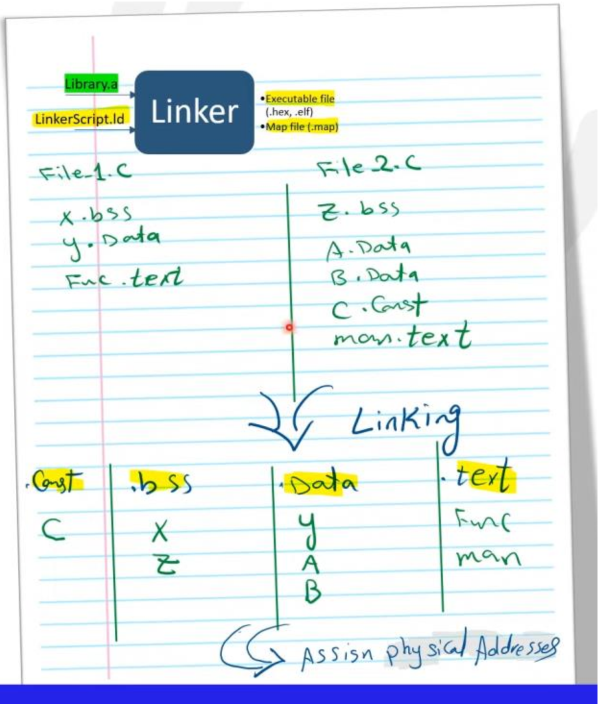


```
#include "app_main.h"

char u8InitGLobalVar = 2;
static long int u32NonInitStaticGlobalVar;
const short int u16ConstGlobalVar = 3;

void main(void)
{
                short int u16LocVar = 4;
                static char u8InitStaticLocVar = 5;
                if(u16ConstGlobalVar == 3)
                {
                                u32NonInitStaticGlobalVar = 6;
                                vidFunc(&u16LocVar);
                }
                else
                {
                                u32NonInitStaticGlobalVar = 7;

                }
                while(1)
                {

                }
}
```

```
extern char u8InitGLobalVar ;

void vidFunc(short int *u16FuncArgPtr)
{
                (*u16FuncArgPtr)++;
                u8InitGLobalVar--;
}
```

## LINKER

- The linker combines all input files(object files and libraries) into a single output file.
- Sections Concatenation
- Resolve the unresolved symbols
- Optimization



| .Const | .bss | .Data | .text |
|---|---|---|---|
| C | X | y | Func |
| | Z | A | man |
| | | B | |

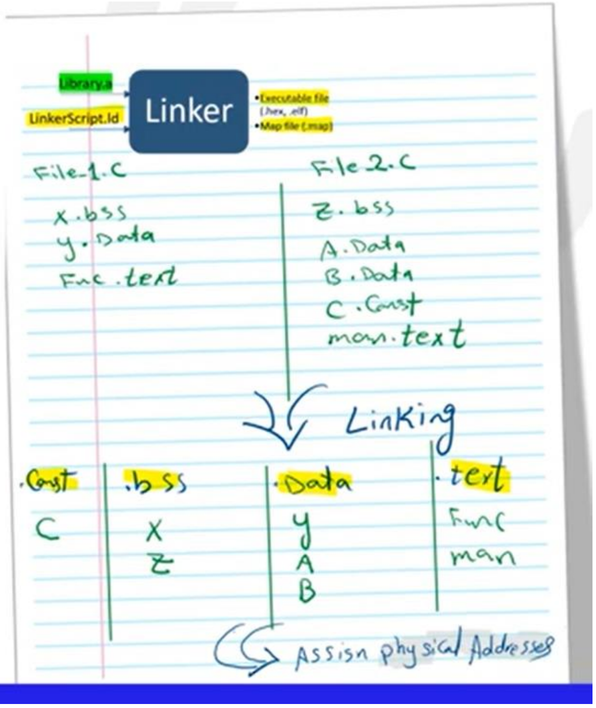
# LINKER

- The linker combines all input files(object files and libraries) into a single output file.
- Sections Concatenation
- Resolve the unresolved symbols
- Optimization

### LINKER SCRIPT

*The main purpose of the linker script is*

- To describe how the sections in the input files should be mapped into the output file,
- To control the memory layout of the output file.



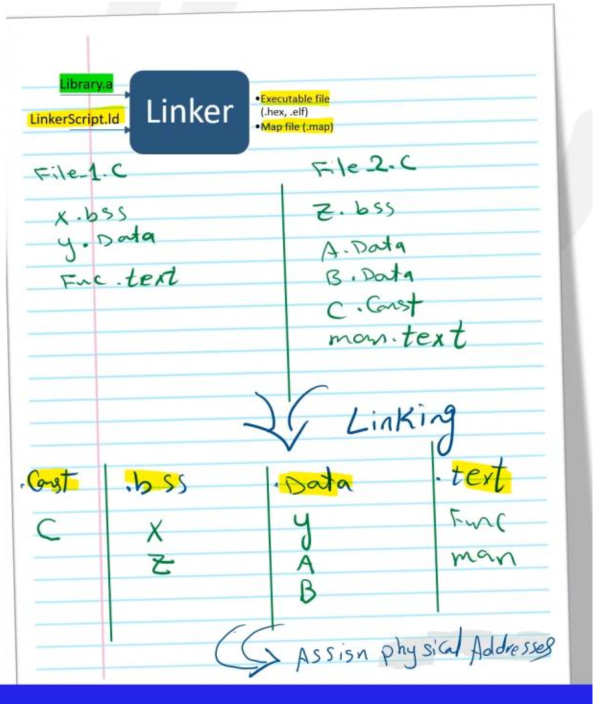
# LINKER

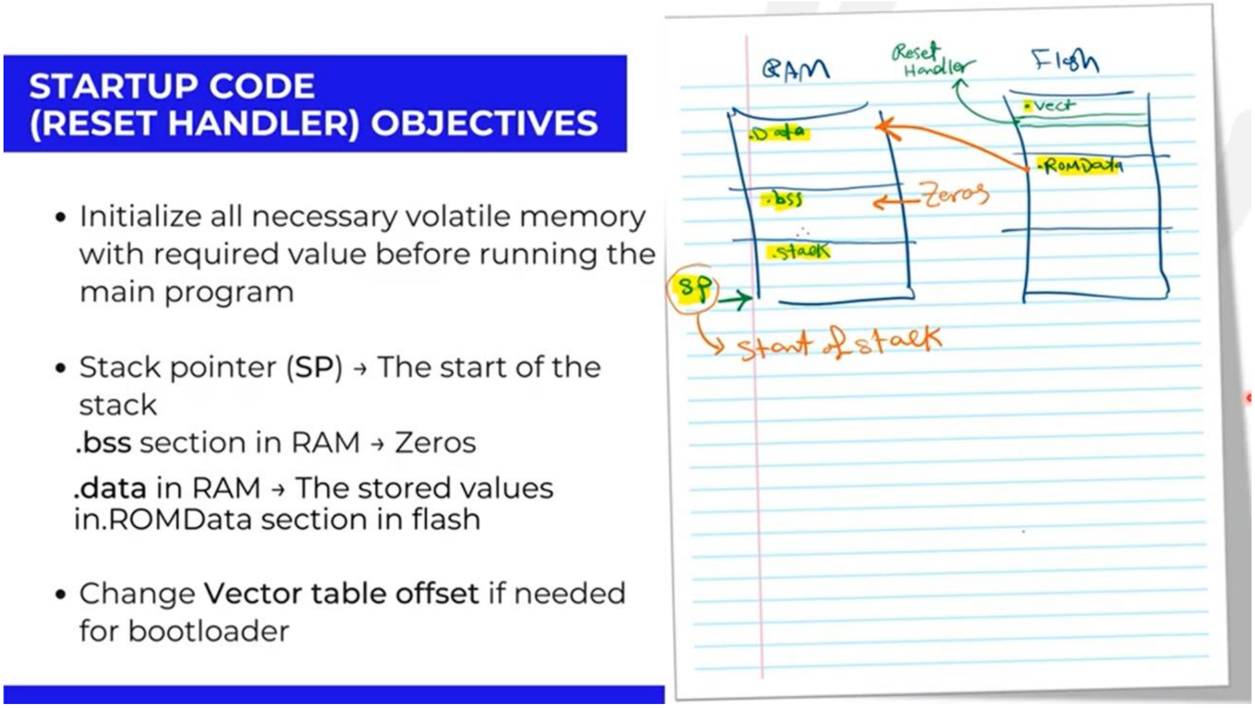
- The linker combines all input files(object files and libraries) into a single output file.
- Sections Concatenation
- Resolve the unresolved symbols
- Optimization

### LINKER SCRIPT

*The main purpose of the linker script is*

- To describe how the sections in the input files should be mapped into the output file,
- To control the memory layout of the output file.



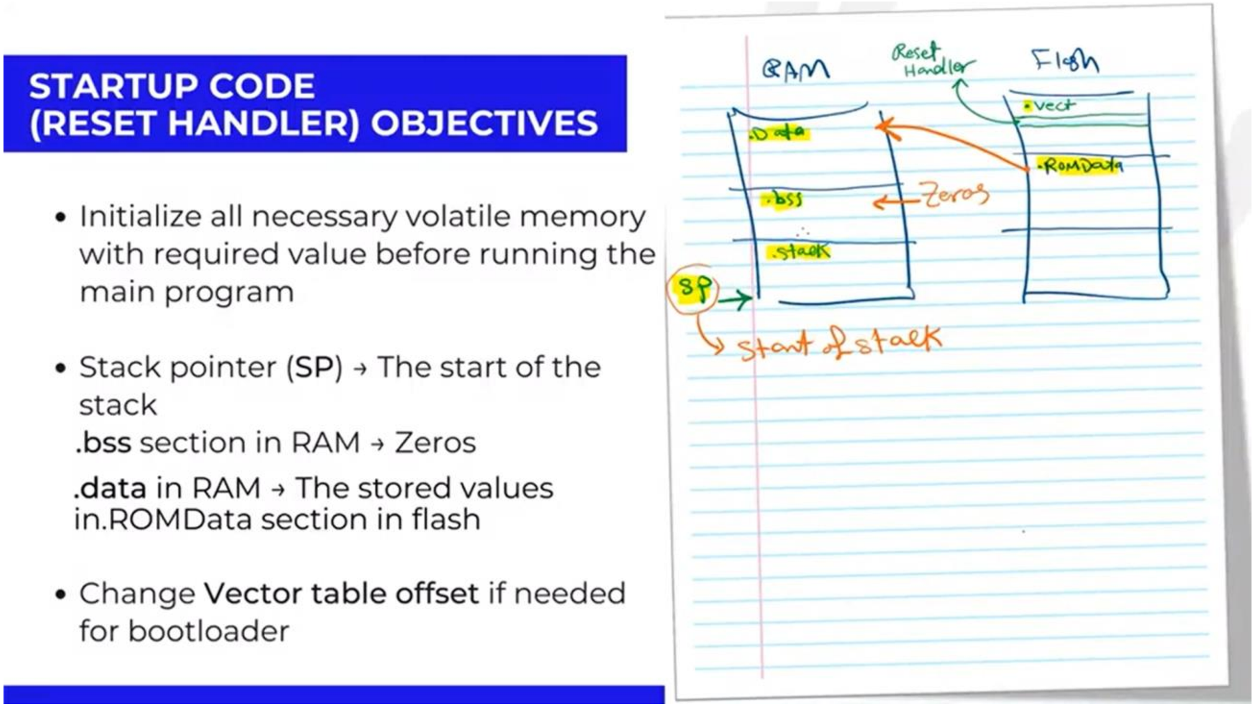


## STARTUP CODE (RESET HANDLER) OBJECTIVES

- Initialize all necessary volatile memory with required value before running the main program
- Stack pointer (**SP**) → The start of the stack

  **.bss** section in RAM → Zeros

  **.data** in RAM → The stored values in.ROMData section in flash
- Change **Vector table offset** if needed for bootloader



STARTUP CODE (RESET HANDLER) OBJECTIVES

- Initialize all necessary volatile memory with required value before running the main program
- Stack pointer (**SP**) → The start of the stack

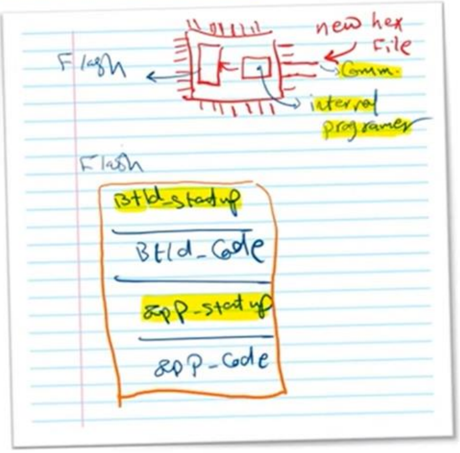
  **.bss** section in RAM → Zeros

  **.data** in RAM → The stored values in.ROMData section in flash
- Change **Vector table offset** if needed for bootloader

# STARTUP CODE VS BOOTLOADER

## STARTUP CODE

Initialize all necessary volatile memory with required value before running the main program



## BOOTLOADER

Separate Program performs application code update through Communication Protocol