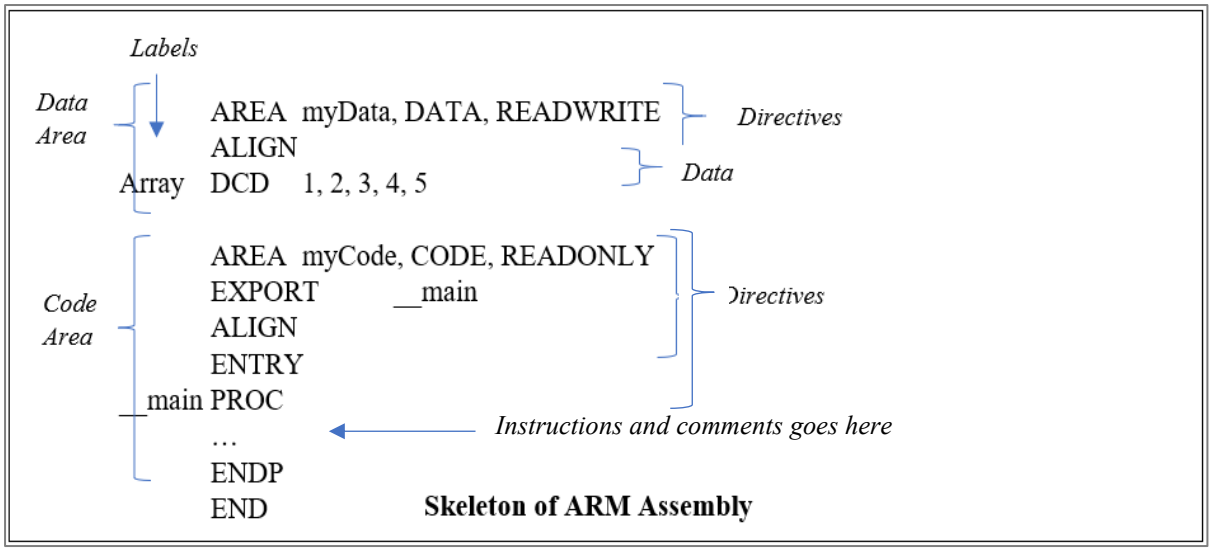
Keil uVision ကို အသုံးပြု၍ ARM Assembly program ရေးသားခြင်း

အားလုံးပဲမင်္ဂလာပါ။

ဒီနေ့ကျွန်တော်မျှဝေပေးချင်တဲ့အကြောင်းအရာကတော့ Keil uVision ကို အသုံးပြုပြီး ARM Assembly program များ ရေးသားနည်းပဲ ဖြစ်ပါတယ်။

ပထမဦးဆုံး Arm assembly program တွေရဲ့ မူကြမ်း(skeleton)ကို ကြည့်ရအောင်။



Label

Label တွေက data (သို့) instruction တွေရဲ့ memory address ကို ကိုယ်စားပြုတာဖြစ်ပါတယ်။ Program ကို execute လုပ်တဲ့အခါမှာ assembler ကနေပြီးတော့ အဲဒီ label တွေကို memory address တွေနဲ့ အစားထိုးပေးပါတယ်။ Assembly program ရေးသားတဲ့အခါ Label တွေကို ဘာ space မှမခံဘဲ စာကြောင်းစစချင်းမှာ ကပ်ရေးရမှာပါ။

Instructions and comments

Instructions နဲ့ comments ကတော့ C program တွေမှာရေးတဲ့ instruction တွေ comments တွေနဲ့ သဘောတရားအတူတူပဲမို့ မရှင်းတော့ပါဘူးxD။

Directives

Directives တွေကတော့ assembler ကို အရေးကြီး information တွေပေးပြီး ကူညီနေတာ ဖြစ်ပါတယ်။

(1) ENTRY ကတော့ program တစ်ခု execute လုပ်ဖို့ ပထမဆုံး instructionကို သတ်မှတ်ပေးထားတာဖြစ်ပါတယ်။ Compiler က ENTRY ကို တွေ့တာနဲ့ အောက်က instructions တွေကို စလုပ်တော့မှာဖြစ်ပါတယ်။ Program တစ်ခုမှာ source files တွေဘယ်လောက်များများ ENTRY တစ်ခါပဲရေးရမှာဖြစ်ပါတယ်။ မရေးလို့တော့ မရပါဘူး။

(2) ALIGN ကတော့ processor performance ပိုကောင်းအောင် data တွေကို ချိန်ညှိပေးတာပါ။ halfword aligned (2 bytes)၊ word aligned (4 bytes)၊ double word aligned (8 bytes) ဆိုပြီး အသီးသီးညှိလို့ရပါတယ်။ default ကတော့ 4 bytes ဖြစ်ပါတယ်။ ALIGN (or) ALIGN 4 ဆိုပြီး ရေးရပါမယ်။ ဥပမာ 32-bit variable တစ်ခုကို word aligned လုပ်မယ်ဆိုပါစို့။ တကယ်လို့ နောက်ထပ်လွတ်နေတဲ့ memory address က 0x8001 ဆိုလျှင် word aligned ဖြစ်တဲ့အတွက်ကြောင့် အဲ့နေရာမှာမသိမ်းပဲနဲ့ 0x8004-0x8007 မှာ သိမ်းထားမှာ ဖြစ်ပါတယ်။ 0x8001-0x8003 နေရာမှာ *padding bytes* လို့ခေါ်တဲ့ ပေါက်ကရတန်ဖိုးတွေ ထည့်သွားမှာပါ။

(3) PROC ရဲ့ အဓိပ္ပာယ်ကတော့ Procedure ဖြစ်ပြီးတော့ ENDP ရဲ့ အဓိပ္ပာယ်ကတော့ End of Procedure ဖြစ်ပါတယ်။ သူတို့ရဲ့အလုပ်ကတော့ fucntion (သို့) subroutine တွေရဲ့ အစနဲ့အဆုံးကို ကြေညာပေးတာဖြစ်ပါတယ်။

(4) END ကတော့ source files တွေရဲ့အဆုံးမှာ ပြီးဆုံးကြောင်းသတ်မှတ်တဲ့အနေနဲ့ ရေးရမှာ ဖြစ်ပါတယ်။

(5) EXPORT __main ဆိုတာ ကျွန်တော်တို့ရေးထားတဲ့ main file ကို startup.s မှာရှိတဲ့ Reset_Handler မှာ အရင် export လုပ်ပြီး Reset_Handler ကနေ ပြန် load တာပဲဖြစ်ပါတယ်။ MCU အလုပ်စလုပ်တဲ့အခါမှာ Reset_Handler ကနေစလုပ်ပါတယ်။ အဲဒါကြောင့် Reset_Handler ထဲက __main နဲ့ EXPORT လုပ်တဲ့ __main က တူရပါမယ်။ မတူရင် ရေးထားတဲ့ main file က အလုပ်မလုပ်တော့ပါဘူး။ ပြောင်းချင်ရင် ၂ ခုစလုံး ပြောင်းရပါမယ်။

```
193  ; Reset handler
194  Reset_Handler    PROC
195                   EXPORT  Reset_Handler             [WEAK]
196          IMPORT  SystemInit
197          IMPORT  __main
198
199                   LDR     R0, =SystemInit
200                   BLX     R0
201                   LDR     R0, =__main
202                   BX      R0
203                   ENDP
204
```

(6) AREA ဆိုတာသည် data (သို့) code section တွေရဲ့အစကို ညွှန်းတာဖြစ်ပါတယ်။ Code section ဆိုတာ instruction ရေးတဲ့ block ဖြစ်ပြီးတော့ Data section ဆိုတာ variable တွေ data တွေကို initialization လုပ်တဲ့ block ဖြစ်ပါတယ်။ Assembly program တစ်ခုမှာ Area အများကြီး ထည့်သုံးနိုင်ပါတယ်။ ဒါပေမယ့် အနည်းဆုံး Code Area တစ်ခုတော့ ပါရပါမယ်။ Area တစ်ခုမှာ နာမည်တစ်ခုရှိရပါမယ် (အပေါ်ကပုံမှာဆိုရင် myData, myCode ပေါ့)။ Program တစ်ပုဒ်မှာ Area နာမည် တစ်ခုနဲ့တစ်ခု တူလို့မရပါဘူး။ CODE Area ရဲ့ default ကတော့ Readonly ဖြစ်ပြီး DATA Area ရဲ့ default ကတော့ ReadWrite ဖြစ်ပါတယ်။

READWRITE နဲ့ READONLY အကြောင်းနားလည်ဖို့ နောက်ပြန်နည်းနည်းဆုတ်ရအောင်။ ကျွန်တော်တို့ရေးတဲ့ source file တွေကို processor က တိုက်ရုက်မသုံးနိုင်ပါဘူး။ အောက်ပါ အဆင့်ဆင့်တွေလုပ်ဆောင်ပြီးမှပဲ သုံးလို့ရတာ ဖြစ်ပါတယ်။

(၁) Preprocessing stage မှာ compiler ကနေပြီးတော့ ကိုယ့် source file ရဲ့ pre-processing directives တွေကို resolve လုပ်တဲ့အဆင့် ဖြစ်ပါတယ်။ pre-processing directives ဆိုတာသည် #include တွေ #define တွေ marco တွေပဲ ဖြစ်ပါတယ်။ .i file တွေကို ထုတ်ပေးပါတယ်။

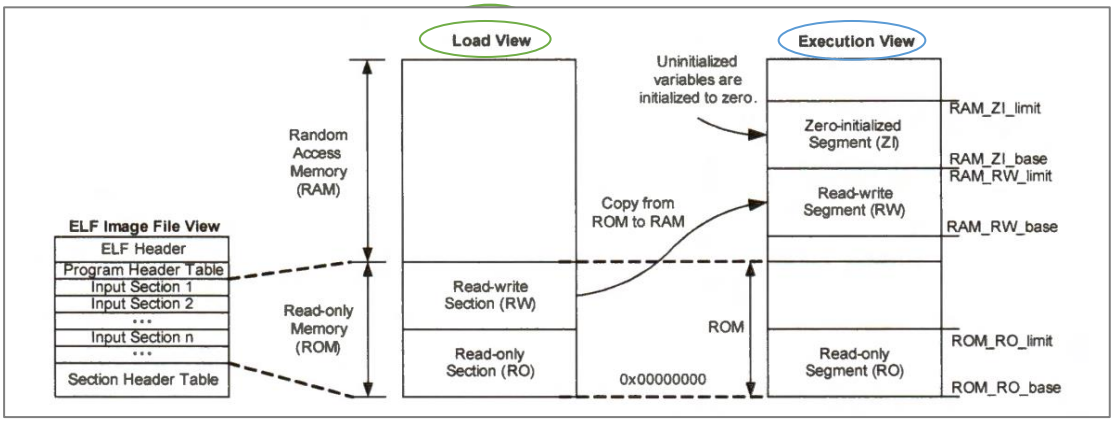
(၂) Code generation stage ဆိုတာ compiler ကနေပြီးတော့ စောနက .i file တွေကို .s file (assembly) ပြောင်းပေးတာ ဖြစ်ပါတယ်။ တစ်နည်းအားဖြင့် High level language code statements တွေကို assembly level language သို့ ပြောင်းပေးတာဖြစ်ပါတယ်။ ကျွန်တော်တို့က assembly နဲ့ရေးတာဆိုတော့ ဒီ stage အထိကလုပ်ပြီးသားပေါ့။

(၃) အဲဒီ .s file တွေကို assembler က opcodes အဖြစ်ပြောင်းပေးပြီး .o file(relocatable object file) တွေကို ELF format (Executable and Linkable Format) နဲ့ထုတ်ပေးပါတယ်။ ဒီ stage ကို assembler stage လို့ ခေါ်ပါတယ်။

(၄) Linking stage မှာတော့ Linker ကနေပြီးတော့ .o file တွေကို final (.elf) file တစ်ခု ထုတ်ပေးပါတယ်။ ELF file က ၂ မျိုး အလုပ်လုပ်ပါတယ်: Linkable interface နဲ့ Executable interface။

(က) Linkable interface က program တစ်ခုကို compiling နဲ့ building process မှာ .o file တွေကို ပေါင်းတဲ့နေရာမှာသုံးပြီး၊

(ခ) Executable interface က program တစ်ခုကို memory မှာ Load၊ Execute လုပ်တဲ့အခါ process image တွေကို ဖန်တီးပေးပါတယ်။ Executable interface မှာ ဘယ်လို အလုပ်လုပ်သလဲဆိုတာ ကြည့်ရအောင်။ ။



Interface of an executable binary file in an elf

Load View က processor က ဘယ်နေရာမှာ load ရမလဲဆိုတာသိဖို့ region အသီးသီးရဲ့ base memory address တွေကိုဖော်ပြပေးပါတယ်။

Execution View က data region တွေကို runtime မှာ ဘယ်လို initialize လုပ်ရမလဲ ဆိုတာကို ဖော်ပြပါတယ်။ Execution view ကိုမှ segment ၄ မျိုး ထပ်ခွဲထားပါသေးတယ်။

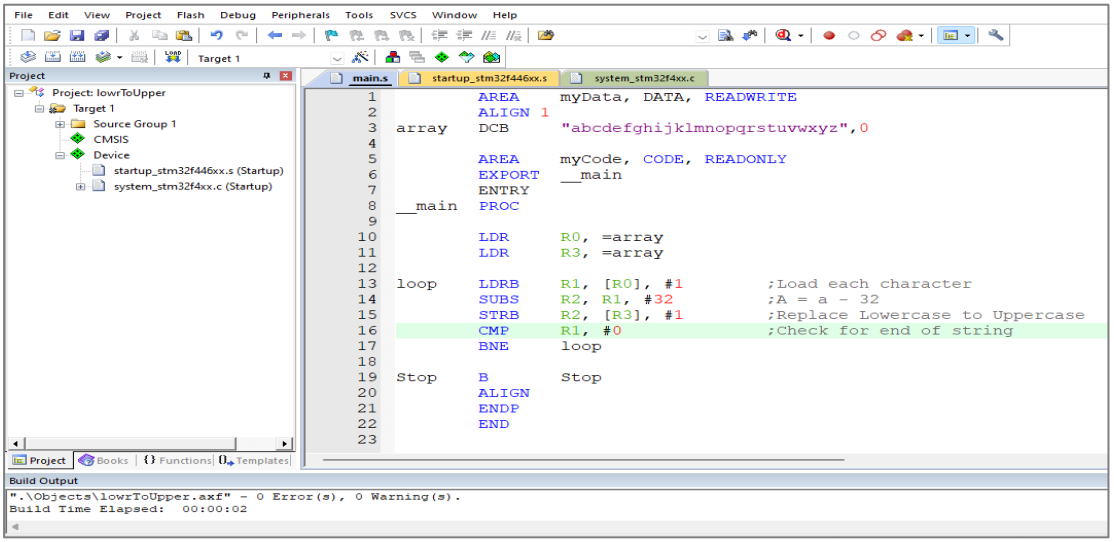
- Text segment: binary machine instructions တွေကို သိုလှောင်ထားပါတယ်။
- Read-only data segment: runtime မှာ ပြောင်းလဲလို့မရတဲ့ variable တွေကို သိမ်းထားပါတယ်။ ဆိုချင်တာက write လုပ်လို့မရပါဘူး။
- Read-write data segment: initialize လုပ်ထားတဲ့ variable တွေကို သိမ်းထား ပါတယ်။ ဒီ data တွေက read/write လုပ်လို့ရပါတယ်။
- Zero-initialized data segment: initialize မလုပ်ထားတဲ့ variable တွေကို သိမ်းထား ပါတယ်။

(၅) နောက်ဆုံးအဆင့်မှာတော့ dynamic linker ကနေပြီးတော့ နောက်ဆုံးရရှိတဲ့ .elf file ကို procesor ကတိုက်ရိုက်သုံးနိုင်တဲ့ machine program (သို့) binary executable အဖြစ် ပြောင်းပြီး CPU ကိုမောင်းနှင်ပါတယ်။

AREA မှာ READONLY လို့ ကြေညာထားရင် data တွေဟာ ROM ထဲရောက်ပြီးတော့ READWRITE လို့ ကြောညာထားရင် RAM ထဲရောက်သွားရမှာပဲဖြစ်ပါတယ်။ ဒါပေမဲ့ သတိချပ်ရမှာက assembly နဲ့ရေးရင် Keil uVision မှာ default ပါလာတဲ့ startup file က အဲ့အလုပ်ကို မလုပ်ဆောင်ပေးပါဘူး။ ကိုယ့်ဟာကို လုပ်ရပါတယ်။ (C language နဲ့ရေးမယ်ဆိုရင်တော့ C runtime က automatic လုပ်ဆောင်ပေးပါတယ်။)

Example program တစ်ပုဒ်ကြည့်ရအောင်။ ။

*စာရွက်သွားမှာစိုးလို့ ကျွန်တော် Instruction တွေ မရှင်းပြတော့ပါဘူး။ Hexa Dev ရဲ့ ARM Assembly Online Class မှာ အသေးစိတ် လေ့လာနိုင်ပါတယ်။



အပေါ်က program လေးကတော့ Data Area မှာ initialize လုပ်ထားတဲ့ alphabet တွေကို lowercase ကနေ uppercase ပြောင်းတဲ့ program ဖြစ်ပါတယ်။ ပုံမှန်ရေးနေကျအတိုင်း default ပါလာတဲ့ startup file ကို modify မလုပ်ဘဲ သုံးမယ်ဆိုရင် error တော့မတက်ပေမဲ့ လိုချင်တဲ့ရလာဒ် ရမှာမဟုတ်ပါဘူး။ ဘာလို့လဲဆိုတော့ Data Area မှာ READWRITE လုပ်ခဲ့တာကြောင့် array ရဲ့ address က SRAM ရဲ့ base address 0x20000000 (ကိုယ်သုံးတဲ့ board ရဲ့ memory map ကိုကြည့်ပါ) ကို ပြနေပေမယ့် data တွေကတော့ အဲ့နေရာမှာ ရှိနေမှာမဟုတ်ပါဘူး။ Flash memory (ROM) ထဲမှာပဲ ရှိနေတာပါ။

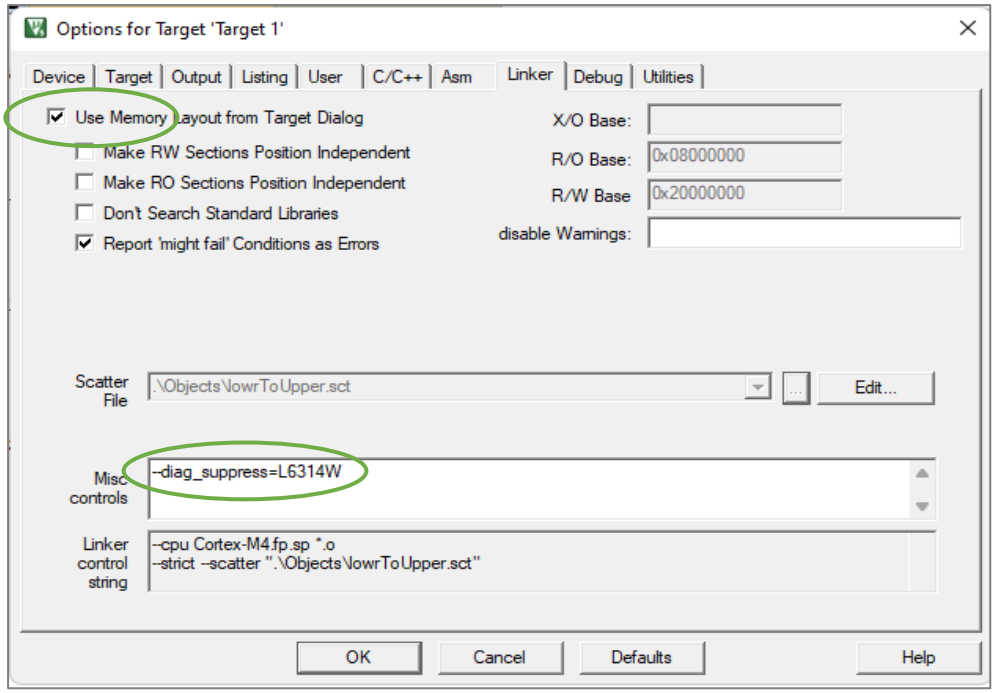
ဒီနေရာမှာ ရွေးချယ်စရာ ၂ ခုရှိပါတယ်။ DATA Area ကို READONLY သတ်မှတ်ပြီး Flash memory (Instruction memory) ထဲမှာပဲ သိမ်းထားမလား (သို့) startup file ကို modify လုပ်မလားပေါ့။

- READONLY နဲ့ရေးမယ်ဆိုရင် instruction memory မှာရှိနေတဲ့ data တွေကို load လို့ရပေမယ့် flash memory ဖြစ်တဲ့အတွက် store လို့မရပါဘူး။ အဲဒါကြောင့် register တစ်ခုမှာ SRAM ရဲ့ base address ကိုသိမ်းပြီး __main file ထဲကနေ အဲဒီ address နေရာမှာ ပြန် store ရမှာပါ။ အဲဒါဆိုရင်တော့ program execute ပြီးတဲ့အခါမှာ output data (Uppercase) တွေက SRAM (Data memory) ထဲ ရောက်သွားပါပြီ။

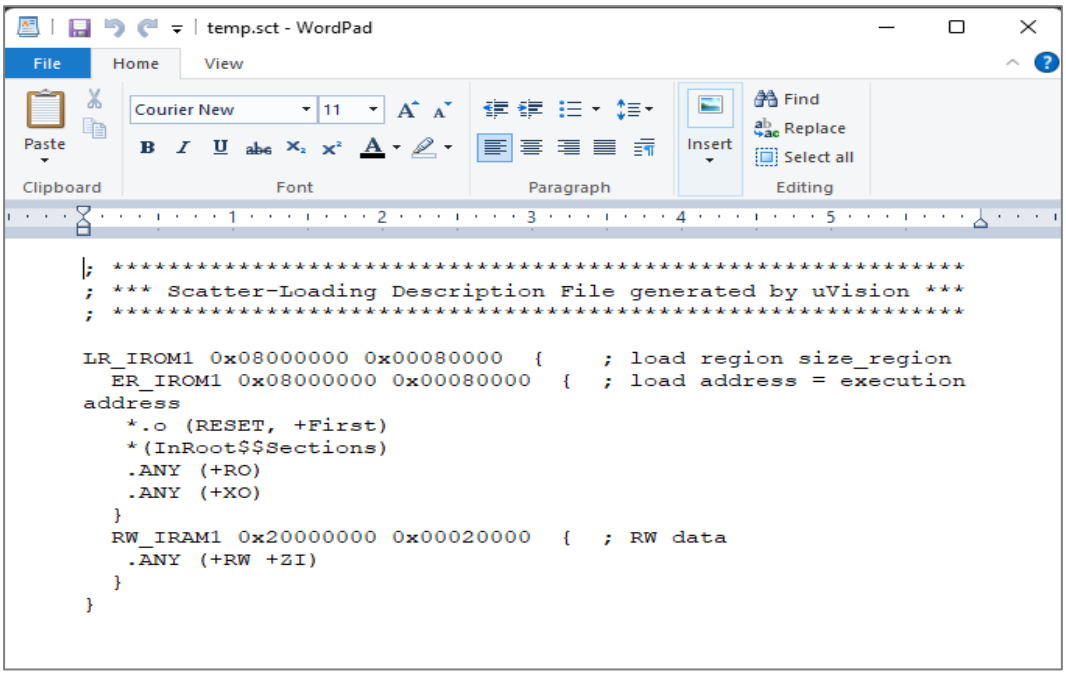
main.s* | startup_stm32f446xx.s | system_stm32f4xx.c

```
1  SRAM_Addr   EQU     0x20000000
2
3              AREA    myData, DATA, READONLY
4              ALIGN
5  alphabet    DCB     "abcdefghijklmnopqrstuvwxyz", 0
6
7              AREA    myCode, CODE, READONLY
8              EXPORT  __main
9              ENTRY
10 __main      PROC
11             LDR     R3, =SRAM_Addr
12             LDR     R0, =alphabet
13
14 loop        LDRB    R1, [R0], #1
15             CMP     R1, #0
16             BEQ     stop
17             SUB     R2, R1, #32
18             STRB    R2, [R3], #1    ; Store to SRAM or Data Memory
19             B       loop
20
21 stop        B       stop
22             ENDP
23             END
```

- READWRITE နဲ့ရေးမယ်ဆိုရင်တော့ startup file ကို အနည်းငယ် modify လုပ်ရပါတယ်။ ပထမဦးဆုံး Options for Target -> Linker tab မှာ Use Memory Layout from Target Dialog ကို tick လုပ်ထားပါ။ အဲဒါက scatter file (.sct) ထုတ်ပေးပါတယ်။ GNU Toolchain (GCC) မှာတော့ linker file (.ld) သုံးပြီးတော့ armcc (Keil uVision) မှာတော့ scatter file (.sct) သုံးပါတယ်။ နှစ်ခုစလုံးကတော့ သဘောတရား အတူတူပါပဲ။ အောက်က ဝိုင်းထားတဲ့ --diag_suppress=L6314W ကတော့ warning L6314W ကို suppress လုပ်ပေးတာဖြစ်ပါတယ်။



ပြီးရင်တော့ build လုပ်လိုက်ပါ။ Project folder ရဲ့ Objects folder ထဲမှာ .sct file တစ်ခု တွေ့ပါလိမ့်မယ်။ Notepad နဲ့ဖွင့်လိုက်ပါ။ အဲ့ဒါ scatter file ပဲ ဖြစ်ပါတယ်။



0x08000000 က Flash memory ရဲ့ start address ဖြစ်ပြီးတော့ ER_IROM1 ကတော့ သူ့ရဲ့ label နာမည်ပဲ ဖြစ်ပါတယ်။ ထိုနည်းတူစွာ 0x20000000 က RAM ရဲ့ start address ဖြစ်ပြီးတော့ RW_IRAM1 က label နာမည် ဖြစ်ပါတယ်။ အကုန်နားမလည်လည်း ကိစ္စမရှိပါဘူး :)။ ဒီ information ကို သုံးပြီးတော့ startup file ကို modify လုပ်ရမှာဖြစ်ပါတယ်။

အရင်ဦးဆုံး ARM Linker user guide ကို ဖွင့်လိုက်ပါ။ Keil uVision ရဲ့ books ထဲမှာရှိပါတယ်။

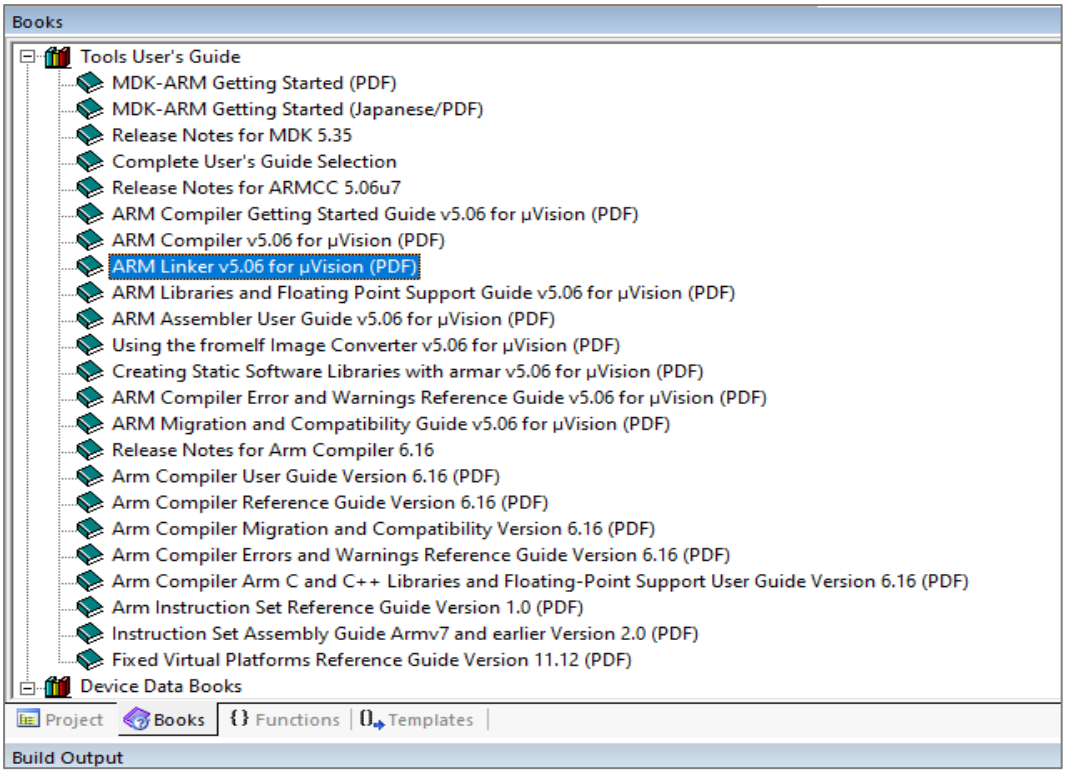


Table 6-1, page 93 ကို သွားလိုက်ပါ။

6.3.2 Image$$ execution region symbols

The linker generates `Image$$` symbols for every execution region present in the image.

The following table shows the symbols that the linker generates for every execution region present in the image. All the symbols refer to execution addresses after the C library is initialized.

Table 6-1 Image$$ execution region symbols

Symbol	Description
`Image$$region_name$$Base`	Execution address of the region.
`Image$$region_name$$Length`	Execution region length in bytes excluding ZI length.
`Image$$region_name$$Limit`	Address of the byte beyond the end of the non-ZI part of the execution region.
`Image$$region_name$$RO$$Base`	Execution address of the RO output section in this region.
`Image$$region_name$$RO$$Length`	Length of the RO output section in bytes.
`Image$$region_name$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region.
`Image$$region_name$$ZI$$Base`	Execution address of the ZI output section in this region.
`Image$$region_name$$ZI$$Length`	Length of the ZI output section in bytes.
`Image$$region_name$$ZI$$Limit`	Address of the byte beyond the end of the ZI output section in the execution region.

Region တွေရဲ့ address တွေကို ဖော်ထုတ်ပေးတဲ့ symbol တွေဖြစ်ပါတယ်။ *region_name* နေရာမှာ scatter file မှာရှိတဲ့ label တွေသုံးရပါတယ်။ ပြီးရင် startup file မှာ __main မလာခင် IMPORT လုပ်ရပါမယ်။

main.s | startup_stm32f446xx.s

```
180  ; Reset handler
181  Reset_Handler    PROC
182                   EXPORT  Reset_Handler             [WEAK]
183          IMPORT  SystemInit
184          IMPORT  __main
185
186                   LDR     R0, =SystemInit
187                   BLX     R0
188
189  ;*********************** Added Manually ************************
190          IMPORT  |Image$$ER_IROM1$$RO$$Limit|     ; First byte beyond the end of RO output section
191          IMPORT  |Image$$RW_IRAM1$$Base|          ; Start of RW output section
192          IMPORT  |Image$$RW_IRAM1$$RW$$Length|     ; Size of RW output section
193
194                  ; Copy the RW Data from Flash to RAM
195                  LDR     R1, =|Image$$ER_IROM1$$RO$$Limit|
196                  LDR     R2, =|Image$$RW_IRAM1$$Base|
197                  LDR     R7, =|Image$$RW_IRAM1$$RW$$Length|
198
199  Copy_RW         CMP     R7, #0
200                  LDRB    R4, [R1], #1    ; load byte and increment address
201                  STRB    R4, [R2], #1    ; store byte and increment address
202                  SUB     R7, #1          ; decrement by 1 byte
203                  BGT     Copy_RW
204  ;***************************** END ******************************
205
206                   LDR     R0, =__main
207                   BX      R0
208                   ENDP
```

|Image$$ER_IROM1$$RO$$Limit| က flash memory မှာ data တွေကို စသိမ်းထားတဲ့ address ကို ထုတ်ပါတယ် - register R1 ထဲကို သိမ်းပါတယ်။ |Image$$RW_IRAM1$$Base| က scatter file မှာ ရေးထားတဲ့ RAM ရဲ့ base memory ကို ထုတ်ပေးပါတယ် - register r2 ထဲကို သိမ်းပါတယ်။ |Image$$RW_IRAM1$$RW$$Length| ကတော့ ကိုယ် initialize လုပ်ထားတဲ့ data ရဲ့ length ကို ဖော်ပြပါတယ် - register r7 ထဲကို သိမ်းပါတယ်။ data length အလိုက် loop ပတ်ပြီး RAM ထဲကိုသိမ်းတာဖြစ်ပါတယ်။ LDRB R4, [R1], #1 က post-index mode ကို သုံးထားတာဖြစ်ပါတယ်။ Flash ထဲက data 1 byte ကို register r4 ထဲထည့်ပြီးမှ Flash ရဲ့ address ကို 1 byte တိုးပါတယ်။ STRB R4, [R2], #1 ကတော့ register r4 ထဲက data 1 byte ကို Ram ထဲကိုသိမ်းပြီး increment တာ ဖြစ်ပါတယ်။ SUB R7, #1 က data length တန်ဖိုးကို ၁ ခုချင်းလျှော့ပေးတာပါ။ BGT Copy_RW ကတော့ R7 က 0 ထက်ကြီးနေသရွေ့ loop ပတ်ခိုင်းမှာဖြစ်ပါတယ်။ ဒီနေရာမှာ - data တွေက byte နဲ့ကြေညာထားတာမို့လို့ LDRB, STRB သုံးပြီး #1 (1byte) နဲ့ increment ရတာဖြစ်ပါတယ်။ word နဲ့ကြေညာထားရင်တော့ LDR, STR ပဲသုံးပြီး #4 (4byte) နဲ့ increment ရမှာပါ။

တစ်ခုသတိထားရမှာက ခုနကသုံးတဲ့ image symbol တွေက compiler v.5.06 အတွက် ရေးထားတာ ဖြစ်တဲ့အတွက် compile လုပ်တဲ့အခါမှာလည်း version 5 ကိုပဲ သုံးရမှာပါ။ version 6 သုံးထားရင် အနည်းငယ် လွဲချော်မှုတွေရှိပါတယ်။ L6314W warning တက်လာရင် ပြဿနာမရှိပါဘူး။ မျက်စိနောက်ရင်တော့ Options for Target ပါတဲ့ အပေါ်မှာပြောခဲ့တဲ့အတိုင်း Linker tab မှာ Misc controls မှာ --diag_suppress=L6314W လို့ရေးလိုက်ပါ။ ဒါဆိုရင် startup mofication အလုပ်ကတော့ ပြီးသွားပါပြီ။

ကျွန်တော်ကတော့ startup ကို modify လုပ်ပြီး data တွေကို Data Memory (RAM) မှာ ကြိုသိမ်းထားတာကို ပိုသဘောကျပါတယ်။ ဘာလို့လဲဆိုတော့ RAM က FLASH ထက်ပိုမြန်ပါတယ်။ ထို့အပြင် Instruction နဲ့ Data ခွဲထားတာကလည်း processor performance ကို တိုးမြှင့်ပေးပါတယ်။ စာဖတ်သူအနေနဲ့ ကိုယ် ကြိုက်နှစ်သက်ရာနည်းလမ်းကို အသုံးပြုနိုင်ပါတယ်။ အားလုံးပဲ အဆင်ပြေပါစေခင်ဗျာ။ ။ အမှားပါရင် ဝေဖန်ထောက်ပြပေးနိုင်ပါတယ်။ ကျေးဇူးတင်ပါတယ်။