Embedded Systems

Embedded systems software structure, code and data

Lesson 12

Francesco Menichelli

francesco.menichelli@uniroma1.it

Assembly program source file (ARM)

- The assembly program source file consists of a sequence of statements, one per line. Each statement has the following format.

```
label:   instruction          @ comment
```

- Each of the components is optional.
- Label:
  - The label is a convenient way to refer to the location of the instruction in memory. The label can be used where ever an address can appear, for example as an operand of the branch instruction. The label name should consist of alphabets, digits, _ and $.
- Comment:
  - A comment starts with an @, and the characters that appear after an @ are ignored.
- Instruction:
  - The instruction could be an ARM instruction or an assembler directive. Assembler directives are commands to the assembler. Assembler directives always start with a . (period).

Example: adding two numbers

```
 .text
start:                          @ Label, not really required
        mov   r0, #5            @ Load register r0 with the value 5
        mov   r1, #4            @ Load register r1 with the value 4
        add   r2, r1, r0        @ Add r0 and r1 and store in r2

stop:   b stop                  @ Infinite loop to stop execution
```

- The .text is an assembler directive, which says that the following instructions have to be assembled into the code section, rather than the .data section.

More Assembler Directives

- .byte Directive
  - The byte sized arguments of .byte are assembled into consecutive bytes in memory.
  - There are similar directives .2byte and .4byte for storing 16 bit values and 32 bit values, respectively.

```
.byte   exp1, exp2, ...
.2byte  exp1, exp2, ...
.4byte  exp1, exp2, ...
```

- The arguments could be simple integer literal, represented as binary (prefixed by 0b or 0B), octal (prefixed by 0), decimal or hexadecimal (prefixed by 0x or 0X).
- The integers could also be represented as character constants (character surrounded by single quotes), in which case the ASCII value of the character will be used.

```
pattern:  .byte 0b01010101, 0b00110011, 0b00001111
npattern: .byte npattern - pattern
halpha:   .byte 'A', 'B', 'C', 'D', 'E', 'F'
dummy:    .4byte 0xDEADBEEF
nalpha:   .byte 'Z' - 'A' + 1
```

More Assembler Directives

- .align Directive
  - ARM requires that the instructions be present in 32-bit aligned memory locations. The address of the first byte, of the 4 bytes in an instruction, should be a multiple of 4.
  - To adhere to this, the .align directive can be used to insert padding bytes till the next byte address will be a multiple of 4.
  - This is required only when data bytes or half words are inserted within code.
- .asciz Directive
  - The .asciz directive accepts string literals as arguments. String literal are a sequence characters in double quotes.
  - The string literals are assembled into consecutive memory locations. The assembler automatically inserts a nul character (\0 character) after each string.

```
str:    .asciz "Hello World"
```

- .ascii Directive
  - same as .asciz, but the assembler does not insert a nul character after each string.

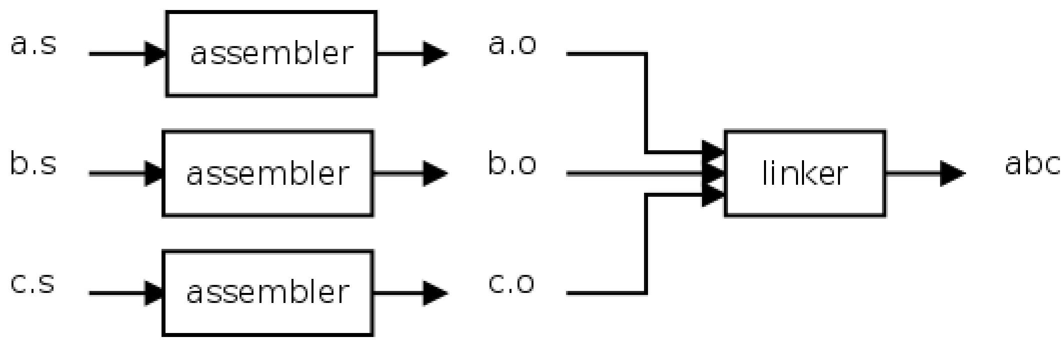
More Assembler Directives

- .equ Directive
  - The assembler maintains something called a symbol table. The symbol table maps label names to addresses. Whenever the assembler encounters a label definition, the assembler makes an entry in the symbol table. And whenever the assembler encounters a label reference, it replaces the label by the corresponding address from the symbol table.
  - Using the assembler directive .equ, it is also possible to manually insert entries in the symbol table, to map names to values, which are not necessarily addresses. Whenever the assembler encounters these names, it replaces them by their corresponding values. These names and label names are together called symbol names.

```
.equ name, expression
```

Linker

- While writing a multi-file program, each file is assembled individually into object files. The linker combines these object files to form the final executable.



- While combining the object files together, the linker performs the following operations.
  – Symbol Resolution
  – Relocation

Symbol Resolution

- In a single file program, while producing the object file, all references to labels are replaced by their corresponding addresses by the assembler.
- But in a multi-file program, if there are any references to labels defined in another file, the assembler marks these references as "unresolved".
- When these object files are passed to the linker, the linker determines the values for these references from the other object files, and patches the code with the correct values.

main.s - Subroutine Invocation:

```
.text
        b start                 @ Skip over the data
arr:    .byte 10, 20, 25        @ Read-only array of bytes
eoa:                            @ Address of end of array + 1

        .align
start:
        ldr   r0, =arr          @ r0 = &arr
        ldr   r1, =eoa          @ r1 = &eoa

        bl    sum               @ Invoke the sum subroutine

stop:   b stop
```

Symbol Resolution

- the .global directive
  - In C, all variables declared outside functions are visible to other files, until explicitly stated as static.
  - In assembly, all labels are static (local to the file), until explicitly stated that they should be visible to other files, using the .global directive.

sum-sub.s - Subroutine Definition:

```
@ Args
        @ r0: Start address of array
        @ r1: End address of array
        @
        @ Result
        @ r3: Sum of Array

        .global sum

sum:    mov   r3, #0          @ r3 = 0
loop:   ldrb  r2, [r0], #1    @ r2 = *r0++     ; Get array element
        add   r3, r2, r3      @ r3 += r2       ; Calculate sum
        cmp   r0, r1          @ if (r0 != r1) ; Check if hit end-of-array
        bne   loop            @    goto loop  ; Loop
        mov   pc, lr          @ pc = lr        ; Return when done
```

Symbol Resolution

- The files are assembled, and the symbol tables are dumped using the nm command.
- If we focus on the letter in the second column, which specifies the symbol type.
  - A t indicates that the symbol is defined, in the text section.
  - A u indicates that the symbol is undefined.
  - A letter in uppercase indicates that the symbol is .global.
- When the linker is invoked the symbol references will be resolved, and the executable will be produced.

arm-none-eabi-as -o main.o main.s
arm-none-eabi-as -o sum-sub.o sum-sub.s

arm-none-eabi-nm main.o
```
00000004 t arr
00000007 t eoa
00000008 t start
00000014 t stop
         U sum
```
arm-none-eabi-nm sum-sub.o
```
00000004 t loop
00000000 T sum
```

Relocation

- Relocation is the process of changing addresses already assigned to labels. This will also involve patching up all label references to reflect the newly assigned address.
- Primarily, relocation is performed for the following two reasons:
  - Section Merging
  - Section Placement
- Code and data have different run time requirements.
  - Code can be placed in read-only memory
  - Data might require read-write memory.
- It would be convenient, if code and data is not interleaved.

Relocation

- For this purpose, programs are divided into sections. Most programs have at least two sections, .text for code and .data for data.
- Assembler directives .text and .data, are used to switch back and forth between the two sections.

```
         .data
arr:     .word 10, 20, 30, 40, 50
len:     .word 5
         .text
start:   mov r1, #10
         mov r2, #20
         .data
result:  .skip 4
         .text
         add r3, r2, r1
         sub r3, r2, r1
```

.data section

```
0000_0000 arr:     .word 10, 20, 30, 40, 50
0000_0014 len:     .word 5
0000_0018 result:  .skip 4
```

.text section

```
0000_0000 start:   mov r1, #10
0000_0004          mov r2, #20
0000_0008          add r3, r2, r1
0000_000C          sub r3, r2, r1
```

Section Merging

- When dealing with multi-file programs, the sections with the same name (example .text) might appear, in each file.
- The linker is responsible for merging sections from the input files, into sections of the output file.
- By default, the sections, with the same name, from each file is placed contiguously and the label references are patched to reflect the new address.
- The effects of section merging can be seen by looking at the symbol table of the object files and the corresponding executable file.

Section Merging example

arm-none-eabi-nm main.o

```
00000004 t arr
00000007 t eoa
00000008 t start
00000014 t stop
         U sum
```

arm-none-eabi-nm sum-sub.o

```
00000004 t loop ❶
00000000 T sum
```

arm-none-eabi-ld -Ttext=0x0 -o sum.elf main.o sum-sub.o

arm-none-eabi-nm sum.elf

```
...
00000004 t arr
00000007 t eoa
00000008 t start
00000014 t stop
00000024 t loop ❷
00000020 T sum
```

```
Disassembly of section .text:
00000000 <sum>:
   0:    e3a03000 mov     r3, #0
00000004 <loop>:
   4:    e4d02001 ldrb    r2, [r0], #1
   8:    e0823003 add     r3, r2, r3
   c:    e1500001 cmp     r0, r1
  10:    1afffffb bne     4 <loop>
  14:    e1a0f00e mov     pc, lr
```

❶ ❷ The loop symbol has address 0x4 in sum-sub.o, and 0x24 in sum.elf, since the .text section of sum-sub.o is placed right after the .text section of main.o.

Section Merging example

Disassembly of sum.elf

```
Disassembly of section .text:
00000000 <arr-0x4>:
   0:    ea000000 b        8 <start>
00000004 <arr>:
   4:    140a              .short   0x140a
   6:    19                .byte    0x19
00000007 <eoa>:
         ...
00000008 <start>:
   8:    e59f0008 ldr      r0, [pc, #8]       ; 18 <stop+0x4>
   c:    e59f1008 ldr      r1, [pc, #8]       ; 1c <stop+0x8>
  10:    eb000002 bl       20 <sum>
00000014 <stop>:
  14:    eafffffe b        14 <stop>
  18:    00000004 .word    0x00000004
  1c:    00000007 .word    0x00000007
00000020 <sum>:
  20:    e3a03000 mov      r3, #0
00000024 <loop>:
  24:    e4d02001 ldrb     r2, [r0], #1
  28:    e0823003 add      r3, r2, r3
  2c:    e1500001 cmp      r0, r1
  30:    1afffffb bne      24 <loop>
  34:    e1a0f00e mov      pc, lr
```

Section Placement

- When a program is assembled, each section is assumed to start from address 0.
- Labels are assigned values relative to start of the section.
- When the final executable is created, the section is placed at some address X. And all references to the labels defined within the section, are incremented by X, so that they point to the new location.
- The effects of section placement can be seen by looking at the symbol table of the object file and the corresponding executable file.
- For example, place the .text section at address 0x100.

```
arm-none-eabi-ld -Ttext=0x100 -o sum.elf main.o sum-sub.o
arm-none-eabi-nm -n sum.elf
00000104 t arr
00000107 t eoa
00000108 t start
00000114 t stop
00000120 T sum
00000124 t loop
```

Section Placement

- Since ARM uses PC relative branches (offset from program counter) branch instructions are not patched
- References to data must be patched during linking

```
Disassembly of section .text:
00000100 <arr-0x4>:
 100:    ea000000 b        108 <start>
00000104 <arr>:
 104:    140a              .short   0x140a
 106:    19                .byte    0x19
00000107 <eoa>:
         ...
00000108 <start>:
 108:    e59f0008 ldr      r0, [pc, #8]       ; 118 <stop+0x4>
 10c:    e59f1008 ldr      r1, [pc, #8]       ; 11c <stop+0x8>
 110:    eb000002 bl       120 <sum>
00000114 <stop>:
 114:    eafffffe b        114 <stop>
 118:    00000104 .word    0x00000104
 11c:    00000107 .word    0x00000107
00000120 <sum>:
 120:    e3a03000 mov      r3, #0
00000124 <loop>:
 124:    e4d02001 ldrb     r2, [r0], #1
 128:    e0823003 add      r3, r2, r3
 12c:    e1500001 cmp      r0, r1
 130:    1afffffb bne      124 <loop>
 134:    e1a0f00e mov      pc, lr
```

Linker - ld

- Creates an executable file (or a library) from object files created during compilation of a software project
- A linker script may be passed to GNU ld to exercise greater control over the linking process
- If gcc is called without options (-S, -c), it will call ld at the end of the process. The linker script can be passed directly to gcc

Linker script

```
SECTIONS { ❶
        . = 0x00000000; ❷
        .text : { ❸
                abc.o (.text);
                def.o (.text);
        } ❹
}
```

- ❶ The SECTIONS command is the most important linker command, it specifies how the sections are to be merged and at what location they are to be placed.
- ❷ Within the block following the SECTIONS command, the . (period) represents the location counter. The location is always initialized to 0x0. It can be modified by assigning a new value to it. Setting the value to 0x0 at the beginning is superfluous.
- ❸ ❹ This part of the script specifies that, the .text section from the input files abc.o and def.o should go to the .text section of the output file.
- The linker script can be further simplified and generalized by using the wild card character * instead of individually specifying the file names.

Linker script

```
SECTIONS {
          . = 0x00000000;
          .text : { * (.text); }

          . = 0x00000400;
          .data : { * (.data); }
}
```

- Here, the .text section is located at 0x0 and .data is located at 0x400.
- If the location counter is not assigned a different value, the .text and .data sections will be located at adjacent memory locations

Linker script example – sum of array

```
.data
arr:    .byte 10, 20, 25        @ Read-only array of bytes
eoa:                            @ Address of end of array + 1

        .text
start:
        ldr   r0, =eoa          @ r0 = &eoa
        ldr   r1, =arr          @ r1 = &arr
        mov   r3, #0            @ r3 = 0
loop:   ldrb  r2, [r1], #1      @ r2 = *r1++
        add   r3, r2, r3        @ r3 += r2
        cmp   r1, r0            @ if (r1 != r2)
        bne   loop              @    goto loop
stop:   b stop
```

- This ARM assembly program sum the elements of the array *arr* in *r3 register*
- Assemble
  - *arm-none-eabi-as -o sum-data.o sum-data.s*
- Link
  - *arm-none-eabi-ld -T linker-script.lds -o sum-data.elf sum-data.o*

Linker script example – sum of array

- List symbols
  - *arm-none-eabi-nm -n sum-data.elf*

```
00000000 t start
0000000c t loop
0000001c t stop
00000400 d arr
00000403 d eoa
```

- Display headers
  - *arm-none-eabi-objdump -x sum-data.elf*

```
Sections:
Idx Name              Size      VMA       LMA       File off  Algn
  0 .text             00000028  00000000  00000000  00008000  2**2
                      CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data             00000003  00000400  00000400  00008400  2**0
                      CONTENTS, ALLOC, LOAD, DATA
  2 .ARM.attributes 00000014  00000000  00000000  00008403  2**0
                      CONTENTS, READONLY
```

Linker script example – sum of array

- Disassemble
  - *arm-none-eabi-objdump -d sum-data.elf*

```
0000000 <start>:
   0:    e59f0018          ldr      r0, [pc, #24]    ; 20 <stop+0x4>
   4:    e59f1018          ldr      r1, [pc, #24]    ; 24 <stop+0x8>
   8:    e3a03000          mov      r3, #0

0000000c <loop>:
   c:    e4d12001          ldrb     r2, [r1], #1
  10:    e0823003          add      r3, r2, r3
  14:    e1510000          cmp      r1, r0
  18:    1afffffb          bne      c <loop>

0000001c <stop>:
  1c:    eafffffe          b        1c <stop>
  20:    00000403          .word    0x00000403
  24:    00000400          .word    0x00000400
```

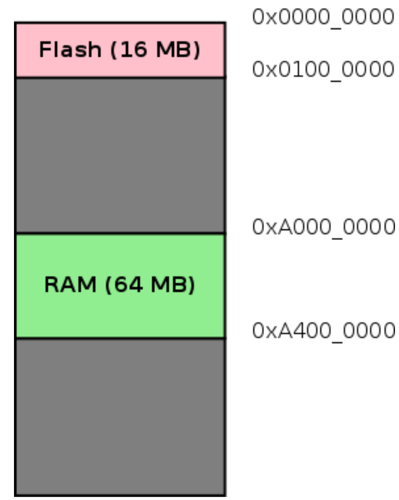
Linker script example – data in RAM

```
.data
val1:   .4byte 10             @ First number
val2:   .4byte 30             @ Second number
result: .4byte 0              @ 4 byte space for result

        .text
        .align
start:
        ldr   r0, =val1       @ r0 = &val1
        ldr   r1, =val2       @ r1 = &val2

        ldr   r2, [r0]        @ r2 = *r0
        ldr   r3, [r1]        @ r3 = *r1

        add   r4, r2, r3      @ r4 = r2 + r3

        ldr   r0, =result     @ r0 = &result
        str   r4, [r0]        @ *r0 = r4

stop:   b stop
```

- RAM is volatile memory, it is not possible to directly make the data available in RAM, on power up.
- All code and data should be stored in Flash before power-up.
- On power-up, a startup code is supposed to copy the data from Flash to RAM, and then proceed with the execution of the program.
- So the program .data section has two addresses, a load address in Flash and a run-time address in RAM.

Linker script example – data in RAM

```
SECTIONS {
        . = 0x00000000;
        .text : { * (.text); }
        etext = .; ❶

        . = 0xA0000000;
        .data : AT (etext) { *
(.data); } ❷
}
```



- ❶ Symbols can be created within the SECTIONS command by assigning values to them. Here etext is assigned the value of the location counter at that position (etext contains the address of the next free location in Flash right after all the code)
- ❷ The AT keyword specifies the load address of the .data section. An address or symbol (whose value is a valid address) could be passed as argument to AT. Here the load address of .data is specified as the location right after all the code in Flash.

Copying .data to RAM

- To copy the data from Flash to RAM, the following information is required.
  - Address of data in Flash (flash_sdata)
  - Address of data in RAM (ram_sdata)
  - Size of the .data section. (data_size)

```
ldr   r0, =flash_sdata
        ldr   r1, =ram_sdata
        ldr   r2, =data_size

copy:
        ldrb  r4, [r0], #1
        strb  r4, [r1], #1
        subs  r2, r2, #1
        bne   copy
```

Linker Script with Section Copy Symbols

```
SECTIONS {
        . = 0x00000000;
        .text : {
              * (.text);
        }
        flash_sdata = .; ❶

        . = 0xA0000000;
        ram_sdata = .; ❷
        .data : AT (flash_sdata) {
              * (.data);
        };
        ram_edata = .; ❸
        data_size = ram_edata - ram_sdata; ❹
}
```

- ❶ Start of data in Flash is right after all the code in Flash.
- ❷ Start of data in RAM is at the base address of RAM.
- ❸ ❹ Obtaining the size of data is not straight forward. The data size is calculated from the difference in the start of data in RAM and the end of data in RAM. Simple expressions are allowed within the linker script.

Add Data in RAM (with copy)

```
 .data
val1:   .4byte 10               @ First number
val2:   .4byte 30               @ Second number
result: .space 4                @ 1 byte space for result

        .text

        ;; Copy data to RAM.
start:
        ldr   r0, =flash_sdata
        ldr   r1, =ram_sdata
        ldr   r2, =data_size

copy:
        ldrb  r4, [r0], #1
        strb  r4, [r1], #1
        subs  r2, r2, #1
        bne   copy

        ;; Add and store result.
        ldr   r0, =val1         @ r0 = &val1
        ldr   r1, =val2         @ r1 = &val2

        ldr   r2, [r0]          @ r2 = *r0
        ldr   r3, [r1]          @ r3 = *r1

        add   r4, r2, r3        @ r4 = r2 + r3

        ldr   r0, =result       @ r0 = &result
        str   r4, [r0]          @ *r0 = r4

stop:   b stop
```

Exception handling

- The first 8 words in the memory map are reserved for the exception vectors. When an exception occurs the control is transferred to one these 8 locations.
- These locations are supposed to contain a branch that will transfer control the appropriate exception handler.

Assembly code for exception vectors

```
.section "vectors"
reset:  b     start
undef:  b     undef
swi:    b     swi
pabt:   b     pabt
dabt:   b     dabt
        nop
irq:    b     irq
fiq:    b     fiq
```

Link script to place *vector* section at 0x0

```
SECTIONS {
        . = 0x00000000;
        .text : {
                * (vectors);
                * (.text);
                ...
        }
        ...
}
```

C Startup

- It is not possible to directly execute C code, when the processor comes out of reset. Since, unlike assembly language, C programs need some basic pre-requisites to be satisfied.
- Before transferring control to C code, the following have to be setup correctly.
  - Stack
  - Global variables
    - Initialized
    - Uninitialized
    - Read-only data

```
static int arr[] = { 1, 10, 4, 5, 6, 7 };
static int sum;
static const int n = sizeof(arr) / sizeof(arr[0]);

int main()
{
        int i;

        for (i = 0; i < n; i++)
                sum += arr[i];
}
```

Stack

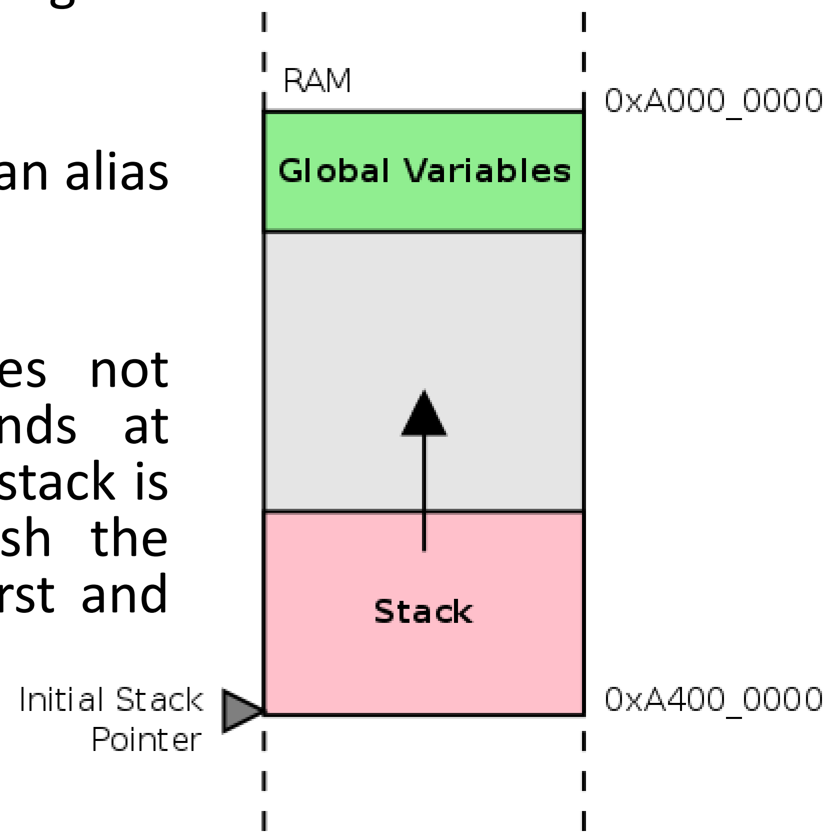
- C uses the stack for storing local variables, passing function arguments, storing return address, etc. So it is essential that the stack be setup correctly, before transferring control to C code.
- Stacks are highly flexible in the ARM architecture, since the implementation is completely left to the software.
- To make sure that code generated by different compilers is interoperable, ARM has created the ARM Architecture Procedure Call Standard (AAPCS).
- The register to be used as the stack pointer and the direction in which the stack grows is all dictated by the AAPCS. According to the AAPCS, register r13 is to be used as the stack pointer. Also the stack should be full-descending.

Stack Placement

- The startup code must point r13 at the highest RAM address, so that the stack can grow downwards (towards lower addresses).
- In our example this can be acheived using the following ARM instruction.
  - ldr sp, =0xA4000000
- Note that the the assembler provides an alias sp for the r13 register.

Note:

- The address 0xA4000000 itself does not correspond to RAM. The RAM ends at 0xA3FFFFFF. But that is OK, since the stack is full-descending, during the first push the stack pointer will be decremented first and the value will be stored.



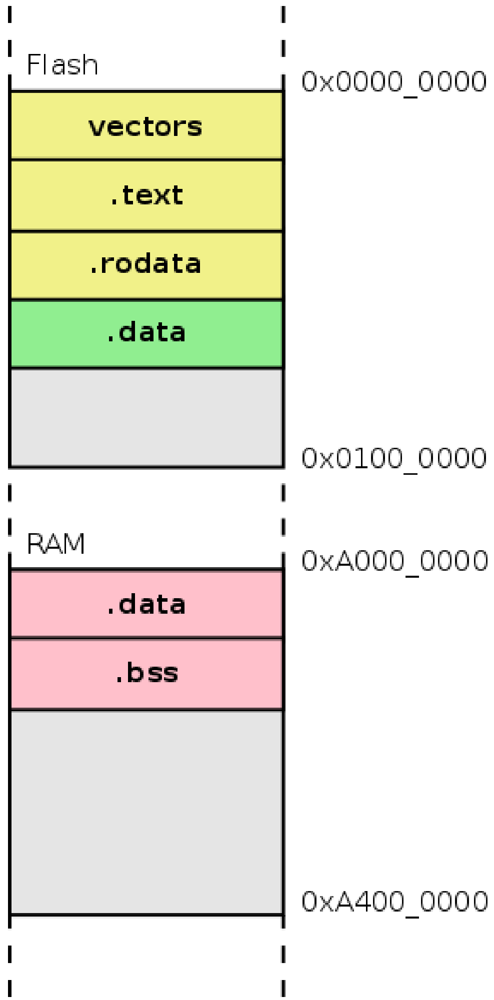
Global Variables

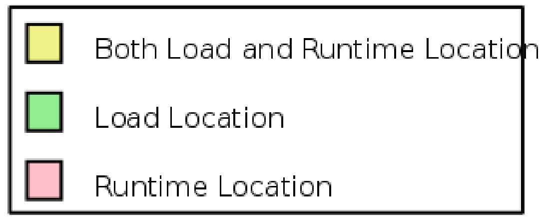
- When C code is compiled, the compiler places initialized global variables in the .data section.
- As with the assembly, the .**data** has to be copied from Flash to RAM.
- The C language guarantees that all uninitialized global variables will be initialized to zero.
- When C programs are compiled, a separate section called .**bss** is used for uninitialized variables.
- Since the value of these variables are all zeroes to start with, <u>they do not have to be stored in Flash</u>.
- Before transferring control to C code, the memory locations corresponding to these variables have to be initialized to zero.

Read-only Data

- GCC places global variables marked as *const* in a separate section, called **.rodata**.
- The .rodata is also used for storing string constants.
- Since contents of .rodata section will not be modified, they can be placed in Flash and never moved. The linker script has to modified to accomodate this.

Linker Script for C code





- Now that we know the pre-requisites we can create the linker script and the startup code.
  - .bss section placement
  - .data section placement
  - vectors section placement
  - .rodata section placement
- The .bss is placed right after .data section in RAM.
- Symbols to locate the start of .bss and end of .bss are also created in the linker script.
- The .rodata is placed right after .text section in Flash.

Linker Script for C code

```
SECTIONS {
        . = 0x00000000;
        .text : {
              * (vectors);
              * (.text);
        }
        .rodata : {
              * (.rodata);
        }
        flash_sdata = .;

        . = 0xA0000000;
        ram_sdata = .;
        .data : AT (flash_sdata) {
              * (.data);
        }
        ram_edata = .;
        data_size = ram_edata - ram_sdata;

        sbss = .;
        .bss : {
             * (.bss);
        }
        ebss = .;
        bss_size = ebss - sbss;
}
```

C Startup Assembly

- The startup code has the following parts
  1. exception vectors
  2. code to copy the .data from Flash to RAM

```
.section "vectors"
reset:  b     start
undef:  b     undef
swi:    b     swi
pabt:   b     pabt
dabt:   b     dabt
        nop
irq:    b     irq
fiq:    b     fiq
```

```
        .text
start:
        @@ Copy data to RAM.
        ldr   r0, =flash_sdata
        ldr   r1, =ram_sdata
        ldr   r2, =data_size

        @@ Handle data_size == 0
        cmp   r2, #0
        beq   init_bss
copy:
        ldrb   r4, [r0], #1
        strb   r4, [r1], #1
        subs   r2, r2, #1
        bne    copy
```

C Startup Assembly

- The startup code has the following parts
  3. code to zero out the .bss
  4. code to setup the stack pointer
  5. branch to main

```
init_bss:
        @@ Initialize .bss
        ldr   r0, =sbss
        ldr   r1, =ebss
        ldr   r2, =bss_size

        @@ Handle bss_size == 0
        cmp   r2, #0
        beq   init_stack

        mov   r4, #0
zero:
        strb  r4, [r0], #1
        subs  r2, r2, #1
        bne   zero

init_stack:
        @@ Initialize the stack pointer
        ldr   sp, =0xA4000000

        bl    main

stop:   b     stop
```

C code compilation

- To compile the code, it is not necessary to invoke the assembler, compiler and linker individually.
  - *arm-none-eabi-gcc -nostdlib -o csum.elf -T csum.lds csum.c startup.s*

Symbol table

- Output of

arm-none-eabi-nm –n csum.elf

```
00000000 t reset
00000004 A bss_size
00000004 t undef
00000008 t swi
0000000c t pabt
00000010 t dabt
00000018 A data_size
00000018 t irq
0000001c t fiq
00000020 T main
0000008c t start
000000a0 t copy
000000b0 t init_bss
000000c8 t zero
000000d4 t init_stack
000000dc t stop
000000f8 r n
000000fc R flash_sdata
a0000000 d arr
a0000000 D ram_sdata
a0000018 D ram_edata
a0000018 D sbss
a0000018 b sum
a000001c B ebss
```

Sections

- Output of

arm-none-eabi-objdump –x csum.elf

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000f8  00000000  00000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000004  000000f8  000000f8  000080f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000018  a0000000  000000fc  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00000004  a0000018  00000114  00010018  2**2
                  ALLOC
  4 .comment      0000005b  00000000  00000000  00010018  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002e  00000000  00000000  00010073  2**0
                  CONTENTS, READONLY
```

Disassembly of .text section

- Output of

arm-none-eabi-objdump –d csum.elf

```
00000000 <reset>:
   0:      ea000021    b          8c <start>
00000004 <undef>:
   4:      eafffffe    b          4 <undef>
. . .
00000018 <irq>:
  18:      eafffffe    b          18 <irq>
0000001c <fiq>:
  1c:      eafffffe    b          1c <fiq>
00000020 <main>:
  20:      e52db004    push       {fp}        ; (str fp, [sp, #-4]!)
  24:      e28db000    add        fp, sp, #0
  28:      e24dd00c    sub        sp, sp, #12
  2c:      e3a03000    mov        r3, #0
. . .
7c:        e49db004    pop        {fp}        ; (ldr fp, [sp], #4)
  80:      e12fff1e    bx         lr
  84:      a0000000    .word      0xa0000000
  88:      a0000018    .word      0xa0000018

0000008c <start>:
  8c:      e59f004c    ldr        r0, [pc, #76]          ; e0 <stop+0x4>
  90:      e59f104c    ldr        r1, [pc, #76]          ; e4 <stop+0x8>
  94:      e59f204c    ldr        r2, [pc, #76]          ; e8 <stop+0xc>
  98:      e3520000    cmp        r2, #0
. . .

000000dc <stop>:
  dc:      eafffffe    b          dc <stop>
  e0:      000000fc    .word      0x000000fc
  e4:      a0000000    .word      0xa0000000
  e8:      00000018    .word      0x00000018
  ec:      a0000018    .word      0xa0000018
  f0:      a000001c    .word      0xa000001c
  f4:      00000004    .word      0x00000004
```

Content of sections

- Output of

arm-none-eabi-objdump –s csum.elf

```
Contents of section .text:
 0000 210000ea feffffea feffffea feffffea  !...............
 0010 feffffea 0000a0e1 feffffea feffffea  ................
 0020 04b02de5 00b08de2 0cd04de2 0030a0e3  ..-.......M..0..
 0030 08300be5 0a0000ea 44309fe5 08201be5  .0......D0... ..
 0040 022193e7 3c309fe5 003093e5 032082e0  .!..<0...0... ..
 0050 30309fe5 002083e5 08301be5 013083e2  00... ...0...0..
 0060 08300be5 0630a0e3 08201be5 030052e1  .0...0... ....R.
 0070 f0ffffba 0300a0e1 00d04be2 04b09de4  ..........K.....
 0080 1eff2fe1 000000a0 180000a0 4c009fe5  ../.........L...
 0090 4c109fe5 4c209fe5 000052e3 0300000a  L...L ....R.....
 00a0 0140d0e4 0140c1e4 012052e2 fbffff1a  .@...@... R.....
 00b0 34009fe5 34109fe5 34209fe5 000052e3  4...4...4 ....R.
 00c0 0300000a 0040a0e3 0140c0e4 012052e2  .....@...@... R.
 00d0 fcffff1a 29d3a0e3 d0ffffeb feffffea  ....)...........
 00e0 fc000000 000000a0 18000000 180000a0  ................
 00f0 1c0000a0 04000000                    ........
Contents of section .rodata:
 00f8 06000000                             ....
Contents of section .data:
 a0000000 01000000 0a000000 04000000 05000000  ................
 a0000010 06000000 07000000                    ........
```