SC2017/CE2107 Microprocessor System Design and Development

Tutorial 1 (with Solutions)

Microprocessor Toolchain, ARM Cortex-M Assembly and Embedded C Programming

1. Below are snippets of the linker command file and map file of a CCS project.

Linker Command File

```
MEMORY
{
    MAIN       (RX) : origin = 0x00000000, length = 0x00040000
    INFO       (RX) : origin = 0x00200000, length = 0x00004000
    SRAM_CODE  (RWX): origin = 0x01000000, length = 0x00010000
    SRAM_DATA  (RW) : origin = 0x20000000, length = 0x00010000
}

SECTIONS
{
    .intvecs:   > 0x00000000
    .text   :   > MAIN
    .const  :   > MAIN
    .cinit  :   > MAIN
    .pinit  :   > MAIN
    .data   :   > SRAM_DATA
    .bss    :   > SRAM_DATA
    .sysmem :   > SRAM_DATA
    .stack  :   > SRAM_DATA
}
```

Map File

```
MEMORY CONFIGURATION

         name            origin    length      used     unused   attr    fill
----------------------  --------  ---------  --------  --------  ----  --------
  MAIN                  00000000   00040000  0000087a  0003f786  R  X
  INFO                  00200000   00004000  00000000  00004000  R  X
  SRAM_CODE             01000000   00010000  000007c0  0000f840  RW X
  SRAM_DATA             20000000   00010000  000007c0  0000f840  RW
```

```
SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.intvecs   0    00000000    000000e4
                  00000000    000000e4     startup_msp432p401r_ccs.obj (.intvecs:retain)

.binit     0    00000000    00000000

.text      0    000000e4    00000756
                  000000e4    0000032c     system_msp432p401r.obj (.text)
                  00000410    000000ec     SineFunction.obj (.text)
                  000004fc    0000009c     rtsv7M4_T_le_v4SPD16_eabi.lib : memcpy_t2.obj (.text)
                  00000598    0000007a                                   : memset_t2.obj (.text)
                  00000612    00000002                                   : mpu_init.obj (.text)
                  00000614    00000070                                   : autoinit.obj (.text)
                  00000684    00000068                                   : copy_decompress_lzss.obj (.text:decompress:lzss)
                  000006ec    00000054                                   : boot.obj (.text)
                  00000740    00000054                                   : exit.obj (.text)
                  00000794    0000004c                                   : cpy_tbl.obj (.text)
                  000007e0    00000018                                   : args_main.obj (.text)
                  000007f8    00000014                                   : _lock.obj (.text)
                  0000080c    0000000e                                   : copy_decompress_none.obj (.text:decompress:none)
                  0000081a    0000000e     startup_msp432p401r_ccs.obj (.text)
                  00000828    0000000c     rtsv7M4_T_le_v4SPD16_eabi.lib : copy_zero_init.obj (.text:decompress:ZI)
                  00000834    00000004                                   : pre_init.obj (.text)
                  00000838    00000002                                   : startup.obj (.text)
```

```
GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address   name
-------   ----
00000827  ADC14_IRQHandler
00000827  AES256_IRQHandler
00000827  BusFault_Handler
00000741  C$$EXIT
00000827  COMP_E0_IRQHandler
00000827  COMP_E1_IRQHandler
00000827  CS_IRQHandler
00000411  CubicSin
```

a. If 'MAIN' memory corresponds to the on-chip flash memory of the processor, how much on-chip flash is free for future application expansion?
   - 0x3F786 bytes

b. What is the total code size for the application?
   - 0x756 bytes

c. Which file contribute to the largest code size to the application?
   - system_msp432p401r.obj (0x32C bytes)

d. It appears that many Interrupt Handlers shared the same starting address of 0x827. Why is that so?
   - These interrupts are not used in the application. They are assigned to point to the same default ISR routine that handles unused interrupts. It's a good practice to direct unused interrupts to a specific routine to trap any unexpected triggers.

2. ARM Cortex-M processor supports the Thumb-2 ISA which consists of 16-bit and 32-bit instructions. What is the advantage of this approach, and why is it considered as a 32-bit processor design?

Solution:

ARM processor's ISA originally started with fixed length 32-bit ARM (A32) instructions. To reduce the cost of memory requirement (which during the 80's is expensive), the 16-bit Thumb instruction was introduced. This increase the code density, but at the expense of some efficient 32-bit A32 instructions.

Thumb-2 (T32) is introduced as a superset of the Thumb ISA, with 32-bit instructions added to 16-bit instruction. Hence Thumb-2 is able to provide the code density of 16-bit Thumb instruction set as well as performance of 32-bit ARM instruction set.

While the Cortex-M uses mixed 16-bit and 32-bit instructions, it is considered a 32-bit processor since the data operations are performed based on 32-bit length data. I.e. All registers, which are used to store data, are 32-bit size.

Other information for discussion: T32 is backwards compatible with 16-bit Thumb while covers most of the functionality of A32 instructions. Main difference between A32 and T32 32-bit instructions are most T32 instructions are unconditional, while the A32 instruction can be conditional. To handle both 16-bit and 32-bit instruction, the core fetch and decode the first half-word of the instruction. If it the instruction is decoded to be 32-bit, the second half-word is fetched from the next address location (offset of 2 from current instruction).

3. ARM processors design such as Cortex-M utilizes 32-bit memory mapped I/O for its I/O related devices and peripherals.
Explain the difficulty of such an approach faced in the design of the Thumb-2 (and A32 as well) ISA, and how it is resolved.

Solution:

By using 32-bit memory mapped I/O, all addresses are of 32-bit size. Similarly, data size are of 32-bit size. However, T32 and A32 uses fixed length instruction, where the 16-bit and 32-bit instruction contain the information of both the opcode and the operand(s). As such the size available for the operand in the instruction would be (much) less than 16 and 32 bits. Hence, 16-bit instruction as well as 32-bit instruction would not be able to encode a 32-bit memory address (e.g. LDR R1, 0xF3241022) or handle 32-bit data (e.g. MOV R1, 0xF7340135).

To overcome this limitation, ARM ISA introduces the LDR pseudo-instruction, which is used to store a register with a 32-bit constant data value, or a 32-bit memory address. Unlike the standard LDR usage, which is used to load a register with the content of a memory location through indirect addressing, (e.g. LDR R1, [R2]), the pseudo instruction LDR has the syntax like:

`LDR R1, =value` e.g. `LDR R1, =0xF7340135`

During assembling, the assembler will check whether the constant value specified in the pseudo instruction can be encoded together with the opcode. If not, the constant value will be placed in a nearby memory location (at the end of the area defined by the instruction's AREA directive, but must be within 4KB for ARM, and 1KB for Thumb). A standard LDR instruction with PC relative address is then generated to fetch the value into the register.

`LDR R1, =value` becomes `LDR R1, [PC, offset]`

where offset is the relative location that the 32-bit value is stored from the current instruction, created by the assembler.

4. It is important to manage the memory usage in microprocessor-based systems, as the memory resources are typically rather limited due to cost and/or power considerations.
For the following C program snippet, identify where the various data will be located in the memory when the program is loaded into a processor system.

```
int a = 100;
int b;
char const c[4]="1234";

typedef struct
{ int d;
  char e;
} rec;


void main(void)
 {
  int x;
  rec *ptr_rec;

  ptr_rec = malloc (sizeof(rec));
      :
 }

 int func1(int arg1, arg2, char arg3, arg4, arg5)
 {
    int y;
    static int z = 0;
      :
 }
```

Solution:

Data in a C program when compiled and linked can be located in the following memory segments by the compiler: code/text segment, data segment, bss segment, stack and heap.

i. global variables (a, b and c)
If the variable is initialized/defined, it will be placed in data. If it is not initialized/defined, or initialized to 0 value, it will be placed in the bss segment.

Note: The 0 initialized global variables can be moved to data segment by using different compiling option. For example, in GCC, the default option is

`-fzero-initialized-in-bss`

which make all 0 initialized variables to be stored in bss segment.

But compiling with `-fno-zero-initialized-in-bss option` will put them in data segment.

ii. constant data types (c)
The value of the data will be stored in read_only area, which could be either the text (code), or data that is marked read-only.

iii. local variable declared in functions (x, y)
This will always be stored on the stack (x in stack frame 0, and y in stack frame 1), i.e. dynamic allocated when called

iv. static variables declared in a function (z)
This will be in data (if initialized) or bss(if uninitialized) since it needs to retain the value.

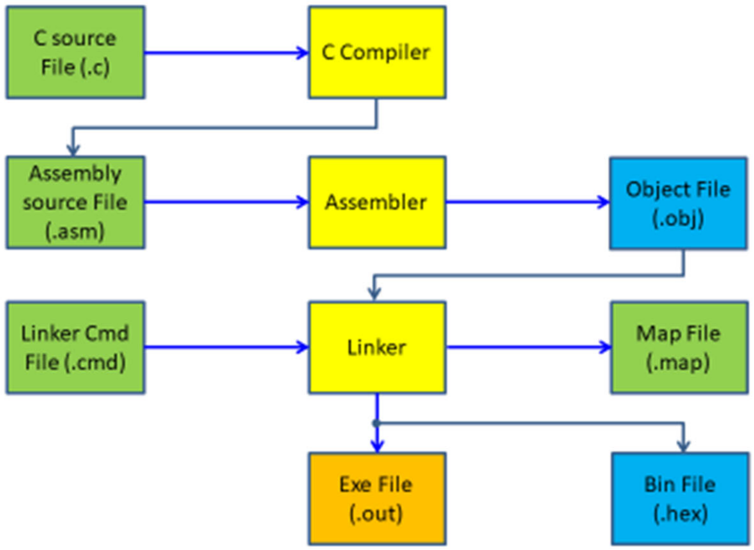
v. Pointer (*ptr_rec)
The pointer will be in stack segment (stack frame 0) since it is a local variable in main function

vi. dynamically allocated space(using malloc, calloc, realloc)
The memory associated with the pointer will be dynamic allocated on the heap

vii. function arguments (arg1 to arg5)
First four arguments will be passed through registers, the rest pass through the stack.

Optional

5. The diagram below shows the development toolchain of for embedded system programming.

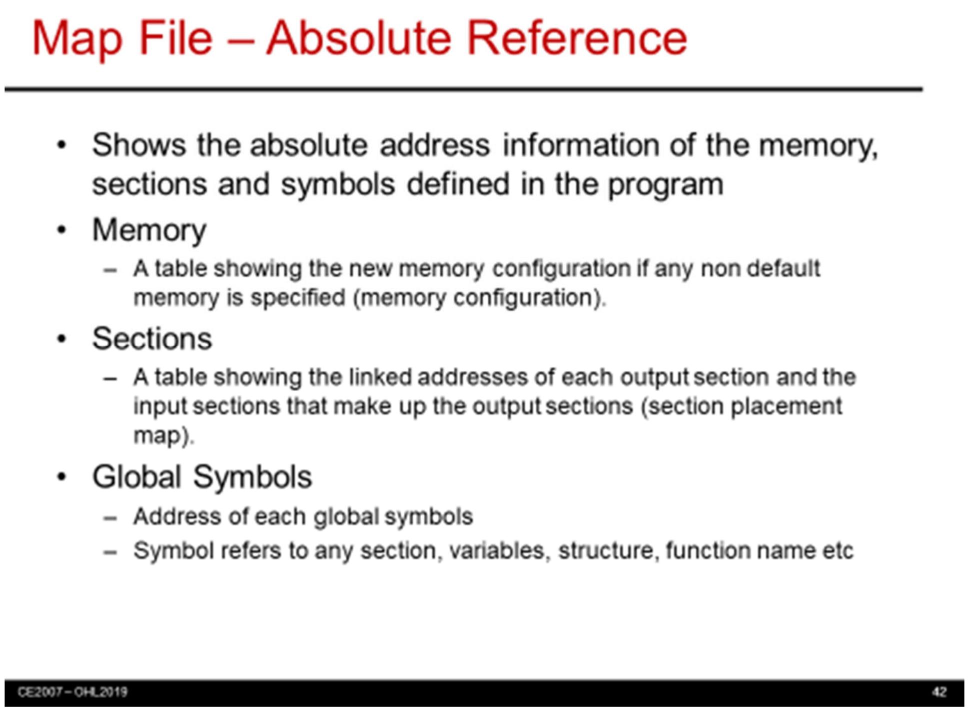


a. What is the function of a Linker Command File?

- A file reference by linker when assigning absolute addresses to different software sections.
- Contains configurations of the target processor's physical memory and software sections to physical memory mapping information.
- May include other misc linker parameters.

b. What information is stored in the <.map> file? What is the difference between addresses stored in map file and listing file generated by the Assembler?

- Shows the absolute address information of the memory, sections and symbols defined in the program. Slide from lecture notes shown below.



- Listing file contains symbolic/relative address references while map file contains absolute address references.

c. What information is stored in a hex file? What is the purpose of <.hex> file?
   - Hex file contains machine codes of the program, together with associated information such as the section, address of each section.
   - It is coded in ASCII text form for readability.
   - It is used typically as an input file to programmer for programming into processor or standalone system memory during production. As such, the machine code are typically partitioned into smaller chunks with CRC code to verify its integrity during programming.

d. What is the difference between an executable file such as the <.out> file and the hex file?
   - An executable contains not only the program code but information for program execution, such as entry point of the program, symbol tables, debug information etc for the processor/debugger.
   - Hex file is meant for programming purpose so contains code/data placement and error correction information.

e. What is the difference between the <.out> file generated by the Linker and the <.obj> file generated by the Assembler? Both seem to contain the machine code that made up the user program.

- <.obj> only contains the machine code of functions in a specific file and contain symbolic/relative address information.
- <.out> contains machine code for the entire program, together with absolute address information and other information needed to execute and debug a program (mentioned in (d) above).

f. Can you spot any Modules/Components that is typically used when compiling a software project but is missing in the diagram?

- Listing file generated by the Assembler. Contains ASM instructions, machine code and relative addressing information within a file.