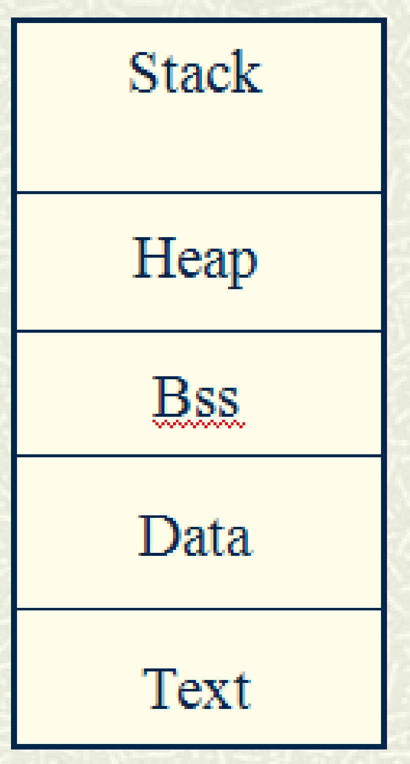
Linker script in CW10

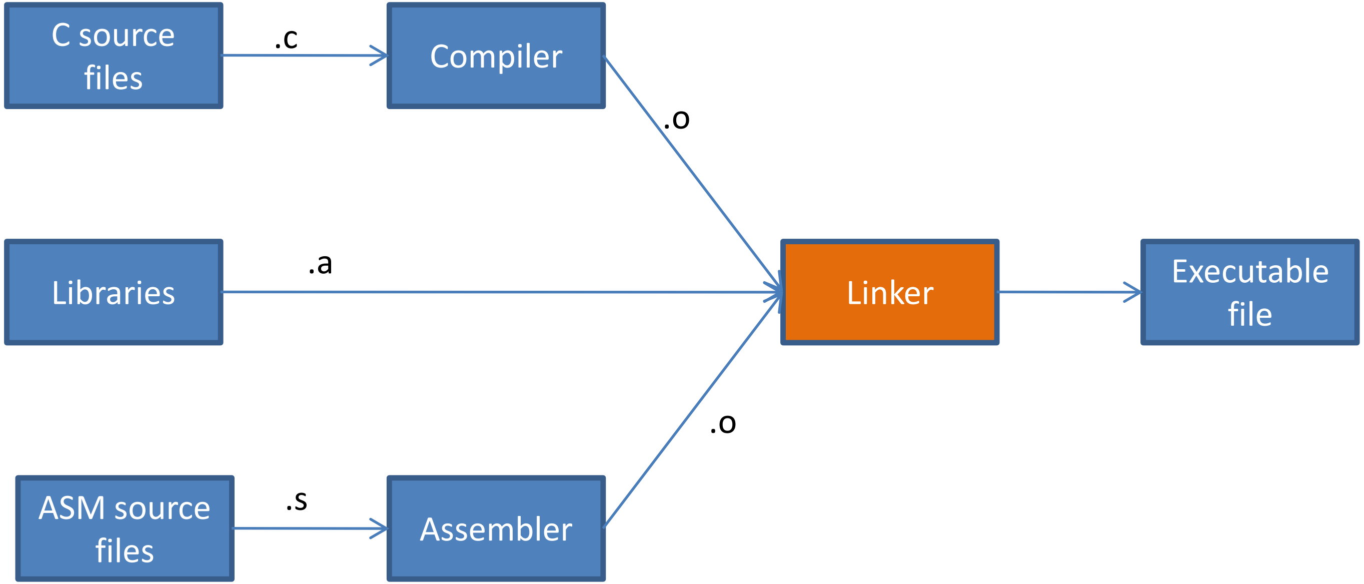
MinhNQ2

Memory Sections

- Text- Instructions that the program runs
- Data – Initialized global variables.
- Bss – Uninitialized global variables. They are initialized to zeroes.
- Heap – Memory returned when calling malloc/new.
- Stack – It stores local variables and return addresses.
- **Each section has different permissions: read/write/execute or a combination of them.**



Building a program



LCF Structure

- Linker command files consist of three kinds of segments, which must be in this order:

- A ***memory*** segment, which begins with the ***MEMORY{}*** directive
- Optional ***closure*** segments, which begin with the ***FORCE_ACTIVE{}***, ***KEEP_SECTION{}***, or ***REF_INCLUDE{}*** directives
- A ***sections*** segment, which begins with the ***SECTIONS{}*** directive

Memory Segment

- Use the memory segment to divide available memory into segments.
- The (***RWX***) portion consists of ELF-access permission flags: R = read, W = write, or X= execute.
- ***ORIGIN*** specifies the start address of the memory segment, either an actual memory address or, via the ***AFTER*** keyword, the name of the preceding segment.
- ***LENGTH*** specifies the size of the memory segment. The value 0 means unlimited length.

Closure Segments

- Closure segments let you make symbols immune from deadstripping.
- ***FORCE_ACTIVE*** - Use this directive to make the linker include a symbol that it otherwise would not include.
- ***KEEP_SECTION*** - Use this directive to keep a section in the link, particularly a user-defined section.
- ***REF_INCLUDE*** - Use this directive to keep a section in the link, provided that there is a reference to the file that contains the section. This is a useful way to include version numbers.

Sections Segment

- Use the sections segment to define the contents of memory sections, and to define any global symbols that you want to use in your output file.
- Format

```
SECTIONS {
      sectionName : [AT (loadAddress)]
        {
            Contents
        } > segmentName
}
```

section_spec

Sections Segment

- ***sectionName***: Name for the output section
- ***AT (loadAddress)***: Optional specify for the load address of the section. The default value is the relocation address.
- ***Contents***: Statements that assign a value to a symbol or specify section placement, including input sections.
- ***segmentName***: Predefined memory-segment destination for the contents of the section. The two variants are:
  - ❖ > segmentName: puts section contents at the beginning of memory segment segmentName.
  - ❖ >> segmentName: appends section contents to the end of memory segment segmentName

LCF syntax

- Variables, Expressions, and Integrals
- Arithmetic, Comment Operators
- Alignment
- Specifying Files and Functions
- Stack and Heap
- ROM-RAM Copying

Variables, Expressions, and Integrals

- All symbol names must start with the underscore character (_). The other characters can be letters, digits, or underscores.
- ***_symbolicname = some_expression;***
- There are 2 types of expression
  - ***Absolute expression*** — the symbol contains the value that it will have in the output file.
  - ***Relocatable expression*** — the value expression is a fixed offset from the base of a section
- LCF syntax for expressions is very similar to the syntax of the C programming language
- _decimal_number = 123245;
- _hex_number = 0x999999FF;

Arithmetic Operators

Precedence	Operators
1	- ~ !
2	* / %
3	+ -
4	>> <<
5	== != > < <= >=
6	&
7	|
8	&&
9	||

Comment Operators

- Use the sharp character “#” for one line comment
  - #this is one line comment
- Use slash and asterisk “/*” for multi-line comment
  - /* this is
  - multiline comment */
- Use double slash “//” for partial-line comment
  - //this is partial-line comment

Specifying Files and Functions

- Specifying Files: Defining the contents of a sections segment includes specifying the source file of each section.
  - ❖ Method 1: listing the files
  - ❖ Method 2: use the asterisk (*) wild-card character, which represents the names of every file in your project.
- Specifying Files: For precise control over function placement within a section, use the **OBJECT** keyword.

```
SECTIONS {
   .program_section :
   {
       OBJECT (beta, main.c)
       OBJECT (alpha, main.c)
       * (.text)
     } > ROOT
```

Stack and Heap

- Reserving space for the stack and heap requires some arithmetic operations to set the symbol values used at runtime.

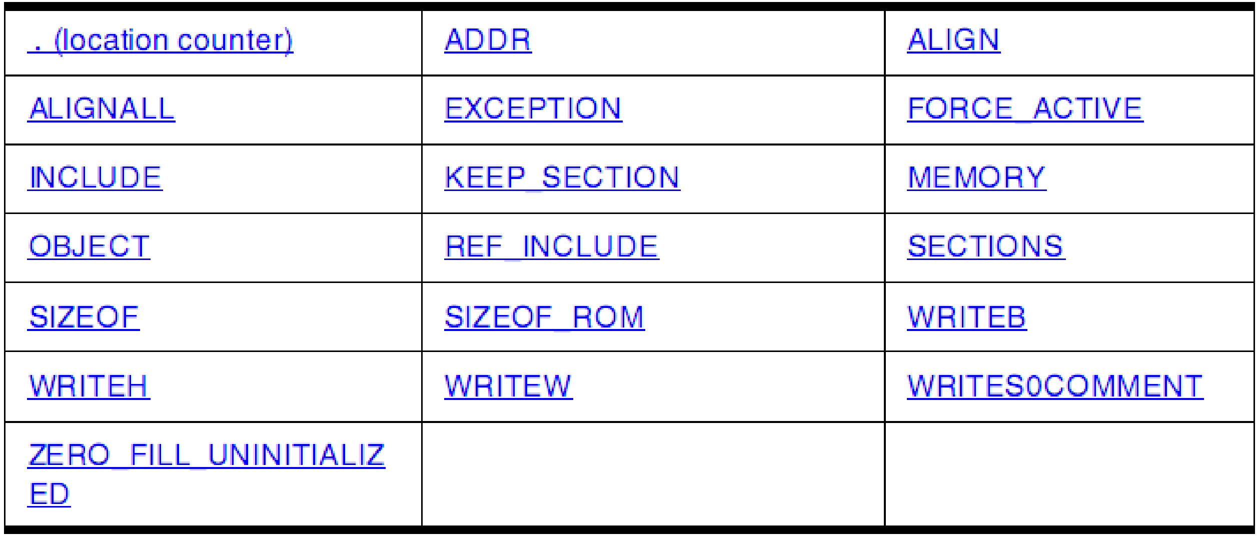
- Stack Setup Operations
- _stack_address = __END_BSS;
- _stack_address = _stack_address & ~7; /*align top of stack by 8 */
- __SP_INIT = _stack_address + 0x4000; /*set stack to 16KB*/

- Heap Setup Operations
- ___heap_addr = __SP_INIT; /* heap grows opposite stack */
- ___heap_size = 0x50000; /* heap size set to 500KB */

ROM-RAM Copying

- It is common that data or code of a program residing in ROM gets copied into RAM at runtime. To indicate such data or code, use the LCF to assign it two addresses:
  - ❖ The memory segment specifies the intended location in RAM
  - ❖ The sections segment specifies the resident location in ROM, via its AT (address) parameter
- For program execution to copy the section from ROM to RAM, a copy table must supply the information that the program needs at runtime. This copy table, which the symbol __S_romp identifies, contains a sequence of three word values per entry:
  - ❖ ROM start address
  - ❖ RAM start address
  - ❖ size

Commands, Directives, and Keywords - 1



. (location counter)	ADDR	ALIGN
ALIGNALL	EXCEPTION	FORCE_ACTIVE
INCLUDE	KEEP_SECTION	MEMORY
OBJECT	REF_INCLUDE	SECTIONS
SIZEOF	SIZEOF_ROM	WRITEB
WRITEH	WRITEW	WRITES0COMMENT
ZERO_FILL_UNINITIALIZED		

Commands, Directives, and Keywords - 2

- **(location counter):** Denotes the current output location

```
.data :
    {
            *.(data)
            *.(bss)
            *.(COMMON)
           __start = .;
            . = __start + 0x10000;
           __end = .;
    } > DATA
```

- **ADDR:** Returns the address of the named section or memory segment

```
ADDR (sectionName | segmentName)
```

  •sectionName: Identifier for a file section.

  •segmentName: Identifier for a memory segment

Commands, Directives, and Keywords - 3

- **ALIGN:** Returns the location-counter value, aligned on a specified boundary.

```
ALIGN(alignValue)
```

alignValue: Alignment-boundary specifier; must be a power of two.

- **ALIGNALL:** Forces minimum alignment of all objects in the current segment to the specified value.

```
ALIGNALL(alignValue);
```

alignValue: Alignment-value specifier; must be a power of two.

Commands, Directives, and Keywords - 4

- **FORCE_ACTIVE:** Starts an optional LCF closure segment that specifies symbols the linker should not deadstrip.

  ```
  FORCE_ACTIVE{ symbol[, symbol] }
  ```

  symbol: Any defined symbol.

- **INCLUDE:** Includes a specified binary file in the output file.

  ```
  INCLUDE filename
  ```

  filename: Name of a binary file. The path of the binary file needs to be specified as linker command line argument.

- **KEEP_SECTION:** Starts an optional LCF closure segment that specifies sections the linker should not deadstrip.

  ```
  KEEP_SECTION{ sectionType[, sectionType] }
  ```

  sectionType: Identifier for any user-defined or predefined section.

Commands, Directives, and Keywords - 5

- **MEMORY:** Starts the LCF memory segment, which defines segments of target memory.

```
MEMORY { memory_spec[, memory_spec] }
```

memory_spec:

```
segmentName (accessFlags) : ORIGIN = address,
LENGTH = length [> fileName]
```

- ❖ segmentName: Name for a new segment of target memory.
- ❖ accessFlags: ELF-access permission flags — R = read, W = write, or X = execute.
- ❖ address: A memory address or an AFTER command
- ❖ Length: Size of the new memory segment
- ❖ fileName: Optional, binary-file destination

Commands, Directives, and Keywords - 6

- **SIZEOF:** Returns the size (in bytes) of the specified segment or section.

```
SIZEOF(segmentName | sectionName)
```

❖segmentName: Name of a segment

- **SIZEOF_ROM:** Returns the size (in bytes) of the specified segment or section.

```
SIZEOF(segmentName | sectionName)
```

❖segmentName: Name of a ROM segment

- **WRITEB, WRITEH, WRITEW:** Inserts a byte of data at the current address of a section.

```
WRITEx (expression);
```

❖expression: Any expression that returns a value in range

Commands, Directives, and Keywords - 7

- **WRITES0COMMENT:** Inserts an S0 comment record into an S-record file.

```
WRITES0COMMENT "comment"
```

  - comment: Comment text

- **ZERO_FILL_UNINITIALIZED:** Forces the linker to put zeroed data into the binary file for uninitialized variables. This directive must lie between the directives MEMORY and SECTIONS; placing it anywhere else would be a syntax error.

```
ZERO_FILL_UNINITIALIZED
```

Defining Sections in Source Code

- **Format**

```
#pragma define_section sname ".istr" [.ustr] [.rostr] [addrmode] [accmode]
#pragma section sname begin
/* Code */
#pragma section sname end
__declspec(section " sname ") <prototype>;
```

- **Parameters**
  - *sname*: Identifier for source references to this user-defined section.
  - *istr*: Section-name string for initialized data assigned to this section
  - *ustr*: Optional: ELF section name for uninitialized data assigned to this section.
  - *rostr*: Optional: ELF section name for read only data assigned to this section.
  - *addrmode*: Optional: optional parameter indicates how the linker addresses the section.
  - *acc*: access permission in this section

Thank you

Q&A