C Programming Language Review and Dissection II:

Start-Up

These lecture notes created by Mr. James B. (Jim) Carlson, NCSU

In these notes . . .

- C Start-Up Module
  - Why is it needed?
  - How is it done?

Start-Up Module – What and Why?

- What?
  - The code that "sets the table" for the C code
- Why?
  - Breaking out this processor-specific information makes the C code more portable
- MCU may need to be configured to run in correct mode
  - Switch to faster (external) clock?
  - Are memory wait states needed?
- Some C constructs must be initialized *before* the program starts running
  - Where does the stack (or stacks) start?
  - What about initialized static variables?
- (Other C constructs are created or managed "on the fly")
  - Building a stack frame (activation record) to call a subroutine
  - Dynamically allocating blocks of memory
  - *Don't have to do anything about these right now*
- Main() function must be called
- Exit from Main() must be accounted for

Run-Time Initialization

- You must link all C/C++ programs with a **bootstrap routine,** which will initialize the C/C++ environment and begin the program. The bootstrap routine is responsible for the following tasks:
  - Set up the stack
  - Process the .cinit run-time initialization table to auto initialize global variables (when using the --rom_model option)
  - Call all global constructors (.pinit for COFF or .init_array for EABI) for C++
  - Call the function main
  - Call exit when main returns
- A sample bootstrap routine is **_c_int00**, provided in **boot.obj** in the run-time support object libraries. **The entry point is usually set to the starting address of the bootstrap routine.**

- **NOTE: The _c_int00 Symbol**
  - If you use the --ram_model or --rom_model link option, _c_int00 is automatically defined as the entry point for the program.

Initialization by the Interrupt Vector

- If your program begins running from load time, you must set up the **reset vector** to branch to **_c_int00**. This causes **boot.obj** to be loaded from the library and your program is initialized correctly. The **boot.obj** places the address of **_c_int00** into a section named **.reset**. This section can then be allocated at the reset vector location using the linker.

Aren't you glad this is taken care of for you with the compiler?

Specifying Where to Allocate Sections in Memory

- **The compiler produces relocatable blocks of code and data.**
- These blocks, called *sections*, *are allocated* in memory in a variety of ways to conform to a variety of system configurations.
- The compiler creates two basic kinds of sections:

  initialized and **uninitialized**.

  - The next slide summarizes the initialized sections created under the COFF ABI mode.
  - The following slide summarizes the initialized sections created under the EABI mode.
  - The third slide summarizes the uninitialized sections.
  - Be aware that the COFF ABI .cinit and .pinit (.init_array in EABI) tables have different formats in EABI.

Initialized Sections Created by the Compiler

Initialized Sections created under the COFF ABI mode

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	Global and static const variables that are explicitly initialized and contain string literals
.pinit	Table of constructors to be called at startup
.text	Executable code and constants

Initialized Sections Created by the Compiler

Initialized Sections Created by the Compiler for EABI

Name	Contents
.args	Command argument for host-based loader; read-only (see the --arg_size option)
.cinit	In EABI mode, the compiler does not generate a .cinit section. However, when the --rom_mode linker option is specified, the linker creates this section, which contains tables for explicitly initialized global and static variables.
.const	Far, const global and static variables, and string constants
.mspabi.exidx	Index table for exception handling; read-only (see --exceptions option)
.mspabi.extab	Unwinded instructions for exception handling; read-only (see --exceptions option)
.init_array	Table of constructors to be called at startup
.name.load	Compressed image of section name; read-only
(See the MSP430 Assembly Language Tools User's Guide for information on copy tables.)	
.ppdata	Data tables for compiler-based profiling (see the --gen_profile_info option)
.ppinfo	Correlation tables for compiler-based profiling (see the --gen_profile_info option)
.rodata	Global and static variables that have near and const qualifiers
.switch	Jump tables for large switch statements
.text	Executable code and constants

Uninitialized Sections Created by the Compiler for Both ABIs

Name	Contents
.args	Linker-created section used to pass arguments from the command line of the loader to the program
.bss	Global and static variables
.stack	Stack
.sysmem	Memory for malloc functions (heap)

Initialized Sections Created by the Compiler

- When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. With the exception of .text, the initialized and uninitialized sections created by the compiler cannot be allocated into internal program memory.

- The linker provides MEMORY and SECTIONS directives for allocating sections.

Typical

```
.bss : {} > RAM          /* GLOBAL & STATIC VARS */
.sysmem : {} > RAM       /* DYNAMIC MEMORY ALLOCATION AREA */
.stack : {} > RAM        /* SOFTWARE SYSTEM STACK */
.text : {} > FLASH       /* CODE */
.cinit : {} > FLASH       /* INITIALIZATION TABLES */
.const : {} > FLASH       /* CONSTANT DATA */
.cio : {} > RAM          /* C I/O BUFFER */
.pinit : {} > RAM        /* C++ CONSTRUCTOR TABLES */
.intvecs : {} > VECTORS  /* MSP430 INTERRUPT VECTORS */
.reset : {} > RESET      /* MSP430 RESET VECTOR */
```