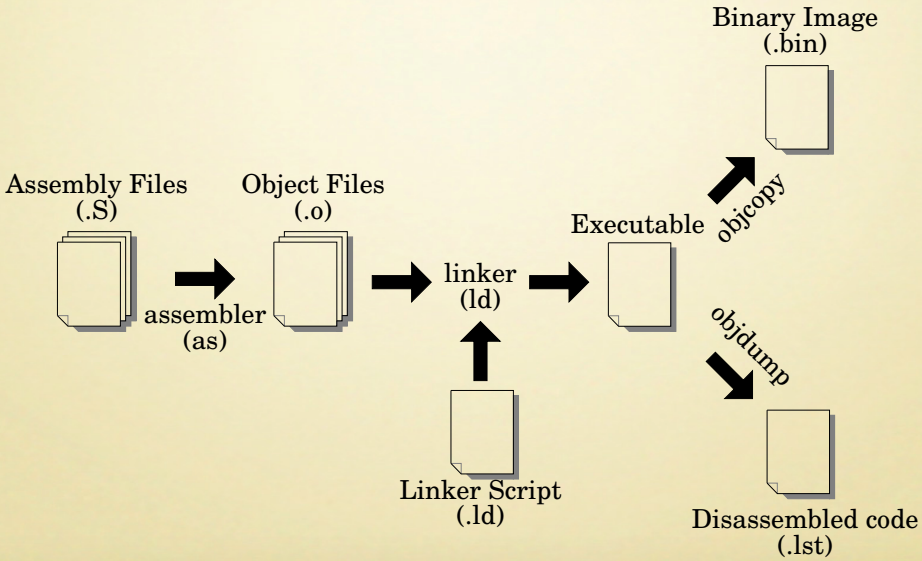
CENG 448 — Real Time Operating Systems

Images and Booting

Dr. Larry D. Pyeatt

South Dakota School of Mines and Technology

Buiding an Image

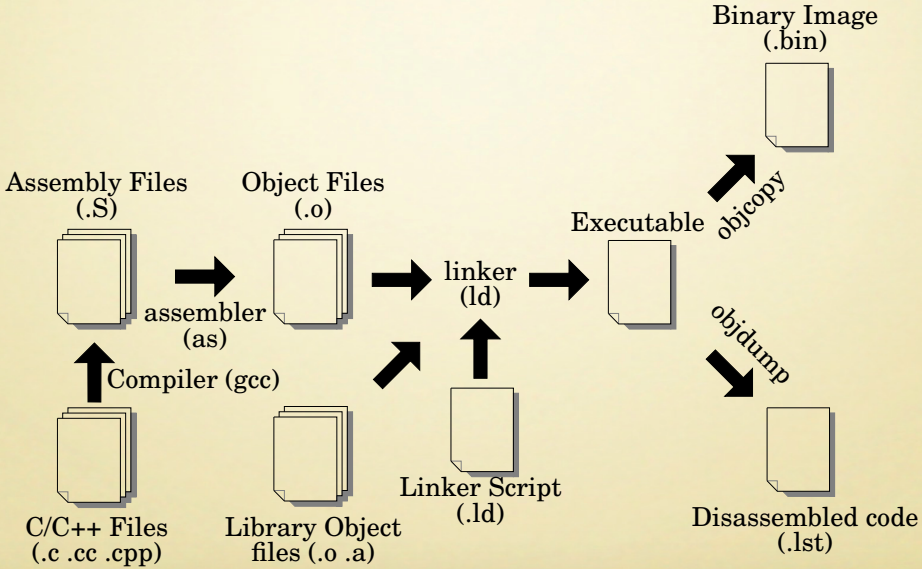


GNU cross development tools

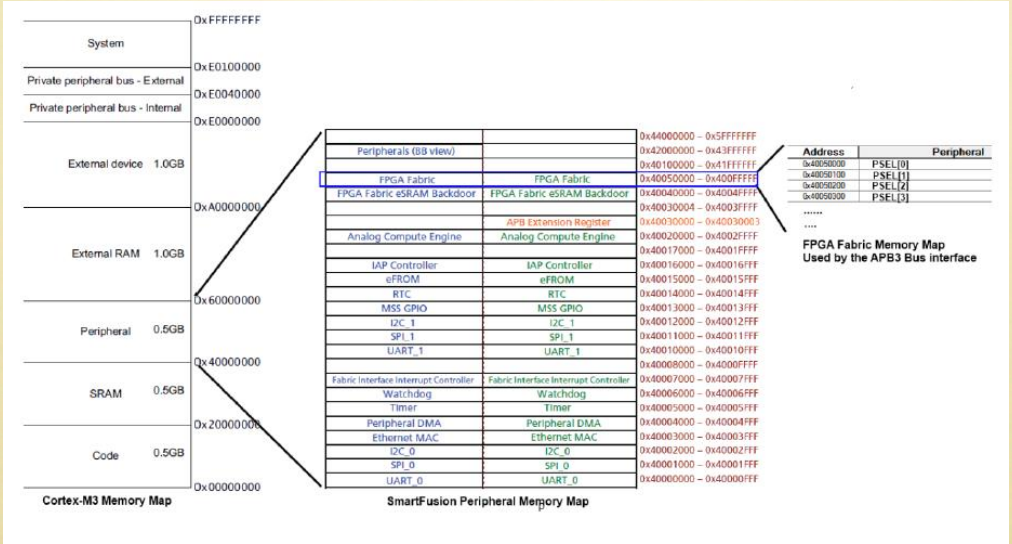
If you add the directory to your PATH, then:

Assembler	arm-none-eabi-as
Linker	arm-none-eabi-ld
Object Copy	arm-none-eabi-objcopy
Object Dump	arm-none-eabi-objdump
C Compiler	arm-none-eabi-gcc
C++ Compiler	arm-none-eabi-g++

Tool Flow


Binary Image
(.bin)
Assembly Files
(.S)
Object Files
(.o)
Executable
objcopy
linker
(ld)
assembler
(as)
objdump
Compiler (gcc)
Linker Script
(.ld)
C/C++ Files
(.c .cc .cpp)
Library Object
files (.o .a)
Disassembled code
(.lst)

Memory Mapping



Linker Script

```
OUTPUT_FORMAT("elf32-littlearm")  /* little endian ARM ELF format */
OUTPUT_ARCH(arm)                  /* ARM CPU */
ENTRY(main)                       /* start executing at "main"   */

MEMORY
{
 /* Tell linker about the internal sRAM on the SmartFusion */
 ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k  /* name it "ram" */
}

SECTIONS
{
 .text :   /* executable file will have a section called .text  */
 {
   . = ALIGN(4);     /* adjust location counter to word boundary */
    *(.text*)        /* insert all sections that match the pattern */
   . = ALIGN(4);     /* adjust location counter to word boundary */
    _etext = .;      /* define a linker variable "_etext" */
 } >ram              /* link so that it will run at 0x20000000 */
}
end = .;             /* define a linker variable "end" */
```

System Initialization

There are may ways that an embedded system can start up, but all involve getting the CPU to run some startup code.

Some questions that have to be answered are:

- how to load the image onto the target system,
- where in memory to load the image,
- how to initiate program execution, and
- how the program produces recognizable output.

Loading the Image

During development, loading the image on the target *usually* involves:

- programming the image into EEPROM or flash, or
- downloading the image over a serial or network connection, or
- downloading the image through a JTAG or BDM interface.

The final product often has a ROM to store the image.

Development boards often have jumpers to select the programming method.

Embedded Loader

It is common to write a *loader* program for the development phase. The loader may be stored in ROM or flash, and executes whenever the system starts up or resets.

Most systems also have a *boot image* which performs some basic initialization before transferring control to the boot loader.

It is common for the boot image and the loader to be stored in the same ROM.

The loader is there to provide a way to load the image. It must communicate with software on the host system, using a data transfer protocol (e.g. TFTP).

The loader typically places the image at a specific location in RAM, and branches to it.

Monitor

An alternative to the boot image/loader approach.

The monitor is a more complex embedded software application that is stored in ROM. It typically

- Initializes some or all peripheral devices (especially critical devices such as serial interfaces, timers, DRAM refresh circuitry, etc.)
- initializes the memory system and prepares for image download,
- initializes interrupt controller and installs default interrupt handler functions, and
- provides a command line interface through some device (typically RS232 UART).

Monitor (slide 2)

The command line interface typically allows the developer to

- download the image,
- read and write system memory locations,
- read and write system registers,
- set and clear program breakpoints,
- single step instructions, and
- reset the system.

Some monitors provide a *target debug agent*, instead of (or in addition to) a command-line interface.

The target debug agent can communicate with the debugger on the host to provide source-level debugging.

Boot Process

When power is applied (or on reset), the processor will immediately begin fetching and executing code from a predefined and hard-wired address.

That address must contain a valid instruction when power is applied, which implies that ROM or flash memory must be mapped to that address in memory.

This *bootstrap code* is usually located either at the beginning or end of the memory map.

Boot - Phase 1

Typical phase 1 boot loader code will

- initialize processor registers,
- initialize stack pointer,
- ensure that interrupts are disabled,
- initialize RAM and configure CPU caches,
- perform some diagnostics to ensure that devices needed for the boot is functioning, and
- initializes devices that are required to load the image.

Boot - Phase 2

The image can come from a variety of sources: ROM, flash, host system, disk drive, SD card, etc. It may be compressed.

The image is often in ELF, COFF, or similar format, and contains initialized and uninitialized data sections. Each section has a section header, which specifies the section type, size, and where it should be loaded in RAM.

The `.bss` and `.sbss` sections must be cleared to zero by the loader.

The `.text`, `.data` and `.sdata` sections are copied from the image to their target locations.

The image may also have `.rodata` and/or `.const` sections, which may be left in ROM. Frequently used read-only data should be transferred to RAM.

Running from ROM

Some systems may choose to run from ROM, rather than copying the entire image to RAM.

It may still be necessary to initialize the data sections.

This usually results in less performance, but can reduce costs.

Boot - Phase 3

After the image is loaded, control is transferred to its entry point. The image must now prepare to run the application. There are three main phases:

- hardware initialization,
- RTOS initialization, and
- application initialization.

Hardware Initialization

1. Install the reset vector
2. Configure the CPU
   - Get processor type
   - Set clock speed
3. Disable interrupts and caches
4. Initialize memory controller, memory chips, and cache
   - Find memory map
   - Test memory
5. Set up interrupt handlers
6. Initialize bus interfaces
7. Initialize peripherals
8. Transfer control to the application (RTOS or bare-metal)

RTOS Initialization

The RTOS typically performs several steps:

1. Initialize the RTOS stack
2. Initialize data structures:
   - Initialize task structs, semaphores, message queues, etc.
   - Set up timer services, interrupt services, memory management services, etc.
3. Create stacks for tasks
4. Initialize extensions
   - TCP/IP stack
   - File system
   - etc.
5. Start the RTOS scheduler and initial application task