Embedded Operating Systems

Che-Wei Chang

chewei@mail.cgu.edu.tw

Department of Computer Science and Information Engineering, Chang Gung University

Course Roadmap

Basic Concepts

- Embedded System Design Concepts
- Embedded System Developing Tools and Operating Systems
- Embedded Linux and Android Environment

Core Technology

- Real-Time System Design and Scheduling Algorithms
- System Synchronization Protocols

Real Implementation

- System Initialization and Memory Management
- Power Management Techniques and System Routine
- Embedded Linux Labs and Exercises on Android

Development Tools of Embedded Systems

Outline

- Introduction
- Embedded Software Development Process
- Embedded Software Architecture
- Embedded System Initialization

Introduction

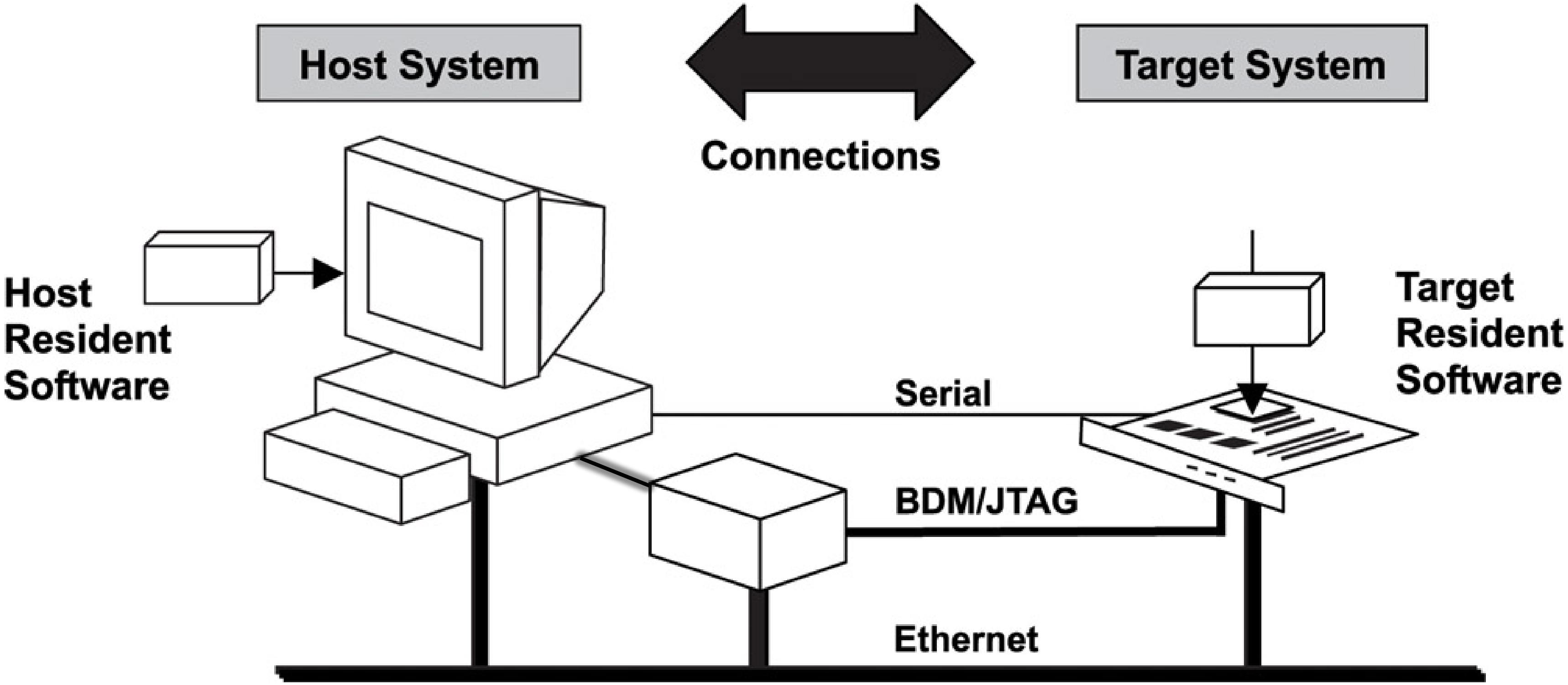
- Developing a "Hello World" application on an embedded system is not trivial
  - As compared with developing in a PC platform
- First, we have to understand how to boot the target system, i.e., the booting process
  - How to load the image onto the target system?
  - What is the memory address to load the image?
  - How to initiate program execution?
  - How the program produces recognizable output?

Components in a Development Environment

- Host System
  - Cross compiler, linker, and source-level debugger
- Target Embedded System
  - Dynamic loader, linker, monitor, debug agent
- Potentially Connectivity Solutions
  - Use serial port, BDM/ICE JTAG, Ethernet
  - Download program images from the host system to the target system
  - Transmit debugger information between the host debugger and the target debug agent

BDM: Background Debug Mode, JTAG (Joint Test Action Group)

Cross-Platform Development Environments



Source: Qing Li and Caroline Yao, "real-time concepts for embedded systems"

Development Environments

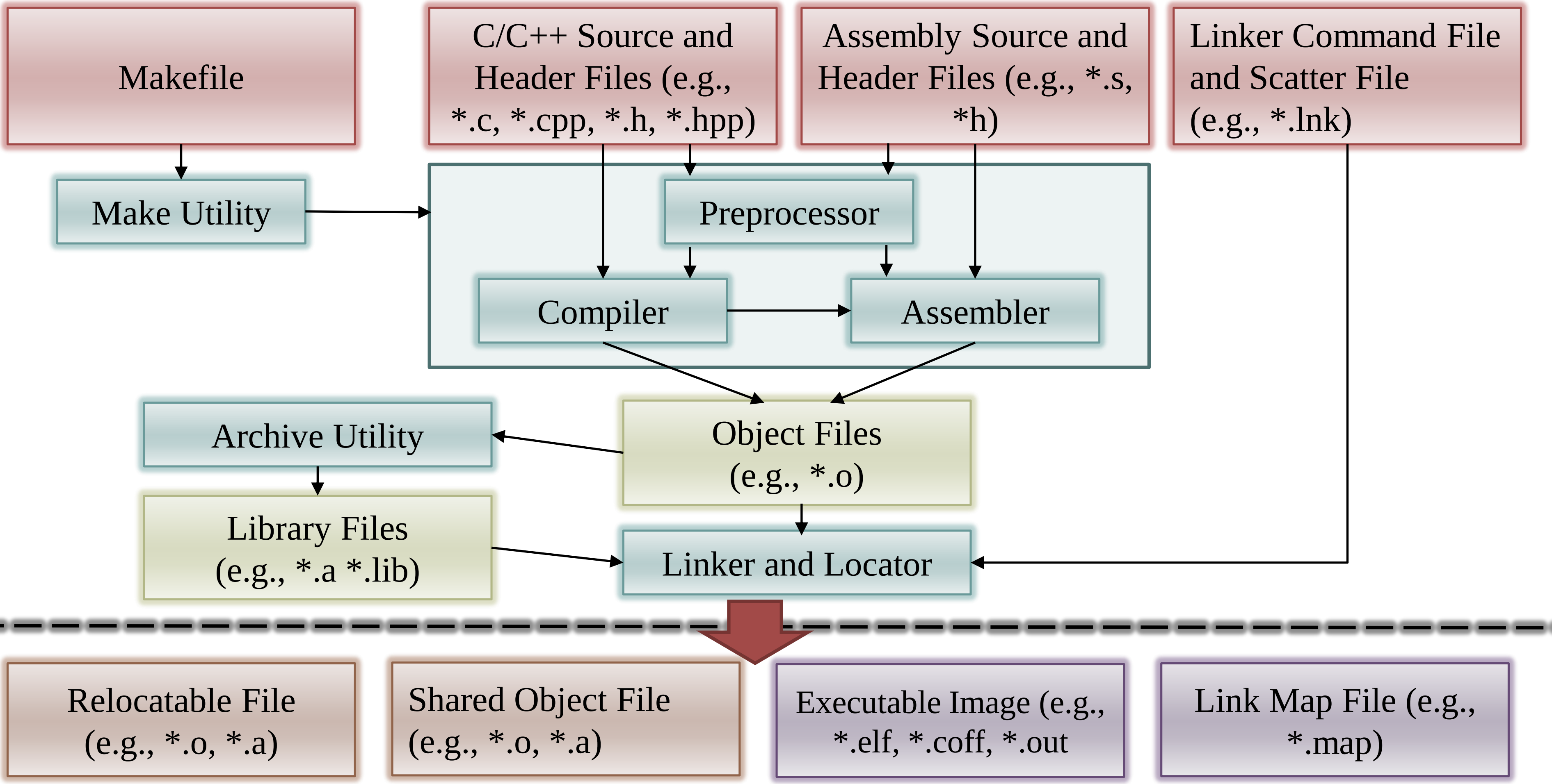
- Native Development
  - Programmers develop and execute applications in the same environment
- Embedded Development
  - Conduct the cross-platform development
  - Understand the target system layout
  - Store the program image on the target system
  - Load the program image during runtime
  - Develop and debug the system iteratively

Overview of Linkers and the Linking Process

Linkers and the Linking Process

- Compiler and assembler produce object files that contain both machine binary code and program data
- Linker takes these object files and produce
  - An executable image
  - Or an object file for further linking
- Linker command file instructs the linker process
- Make utility facilitates an environment of building process
- Archive utility concatenates a collection of object files to form a library

Creating an Image for a Target Embedded System

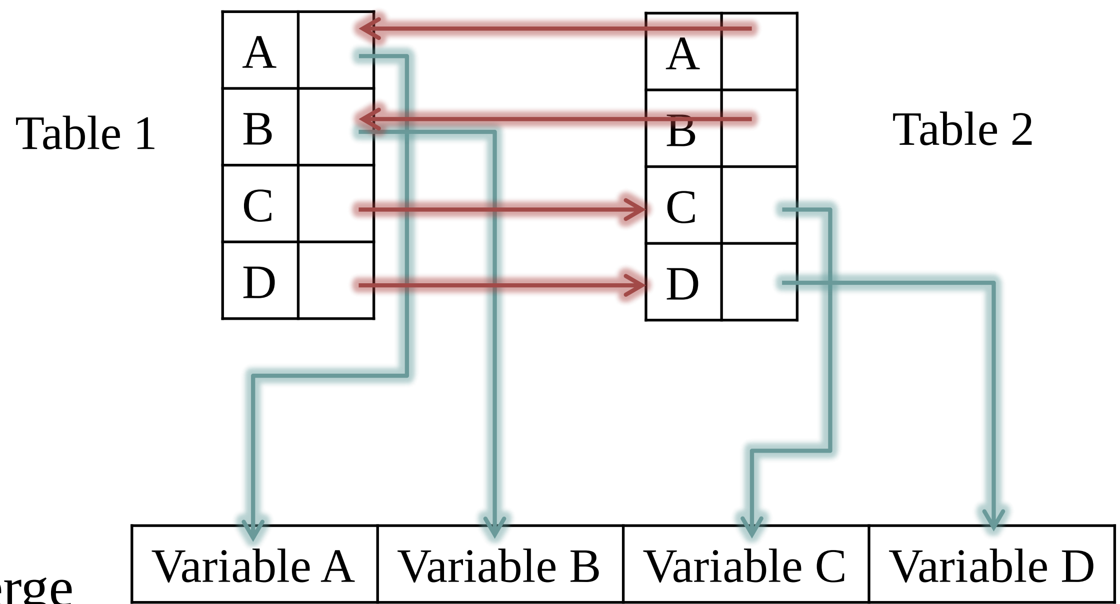


Linker

- Combine multiple object files into
  - A larger relocatable object file
  - A shared object file
  - An executable file
- However, in a source file, it may access another variable or call a function in another source file
- The compiler creates a symbol table in object file which contains the symbol name to its address mapping
  - Global symbols defined in the file being compiled
  - External symbols referenced in the file that the linker need to resolve
- The linker process involves symbol resolution and symbol relocation

Symbol Resolution and Relocation

- Symbol Resolution
  - Resolving references across symbols
  - Merging multiple symbol tables into one



- Relocation
  - Performing section merge
  - Resolving all resolvable relocation
  - Replacing symbolic references with actual addresses (binding)

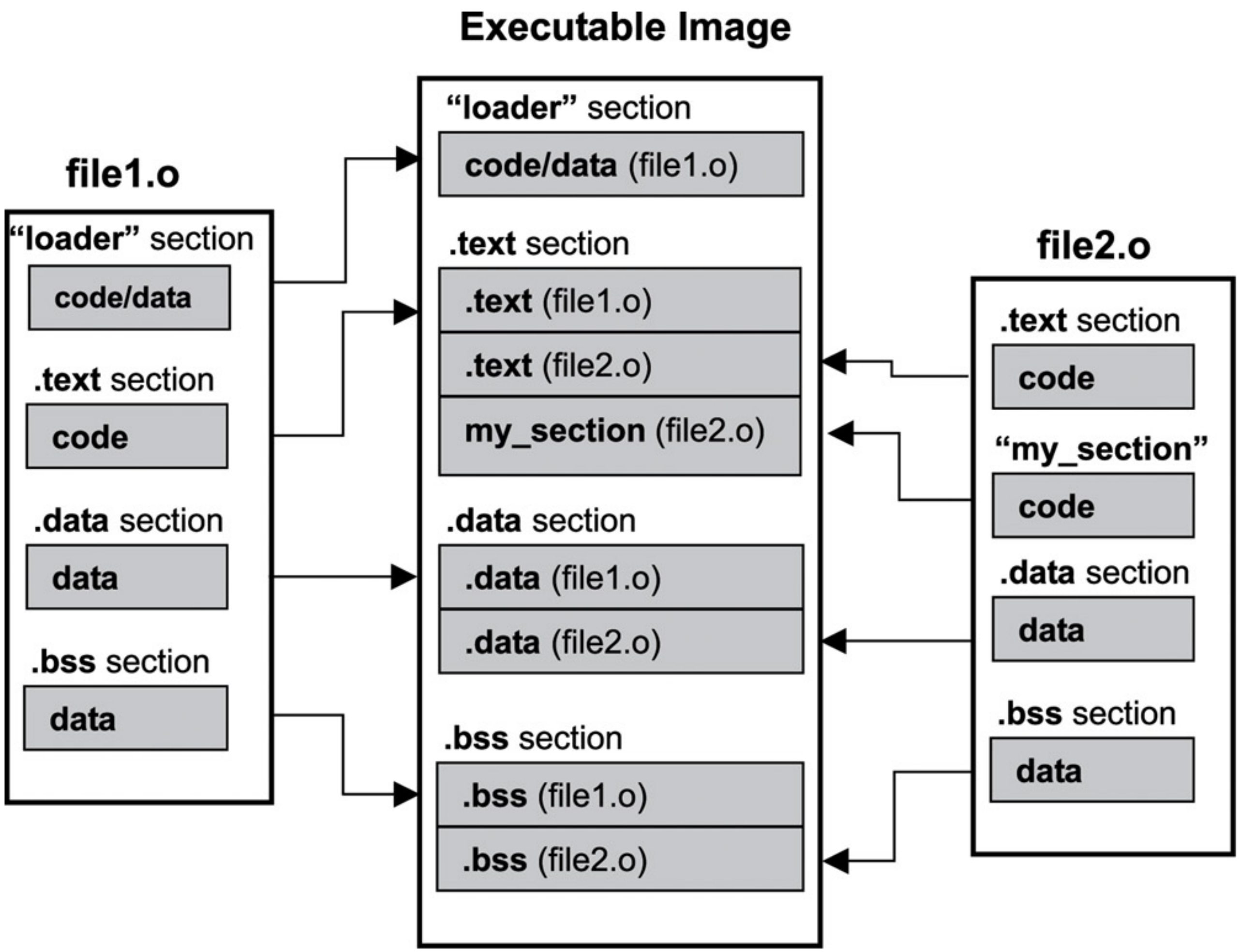
Object File Format

- Object file format
  - The manner in which the information is organized
- Two common object file formats
  - COFF: Common Object File Format
  - ELF: Executable and Linking Format
- Understanding the object file format
  - Allow embedded developers to map an executable image into the target embedded system for static storage, as well as for runtime loading and execution

Executable and Linking Format (ELF)

- An object file contains
  - General information about the object file
    - File size, binary code and data size, source file name
  - Machine-architecture-specific binary instructions and data
  - Symbol table and the symbol relocation table
  - Debug information for the debugger
- A compiler organizes the compiled program into sections
  - Default sections
  - Developer-specified sections
- Sections may contain
  - Binary instruction
  - Binary data
  - Symbol table
  - Relocation table
  - Debug information
  - Load address
  - Run address

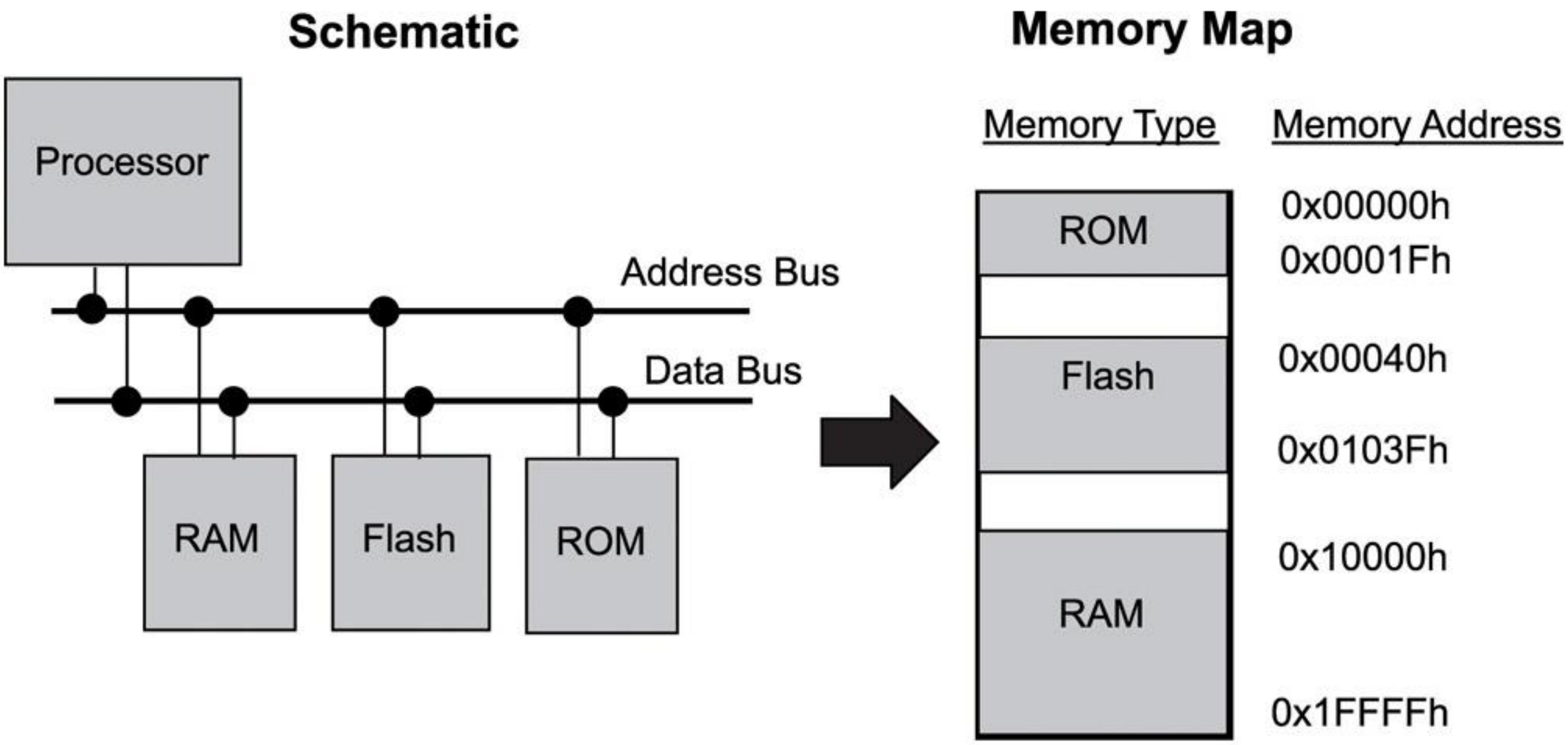
Combining Input Sections into an Executable Image



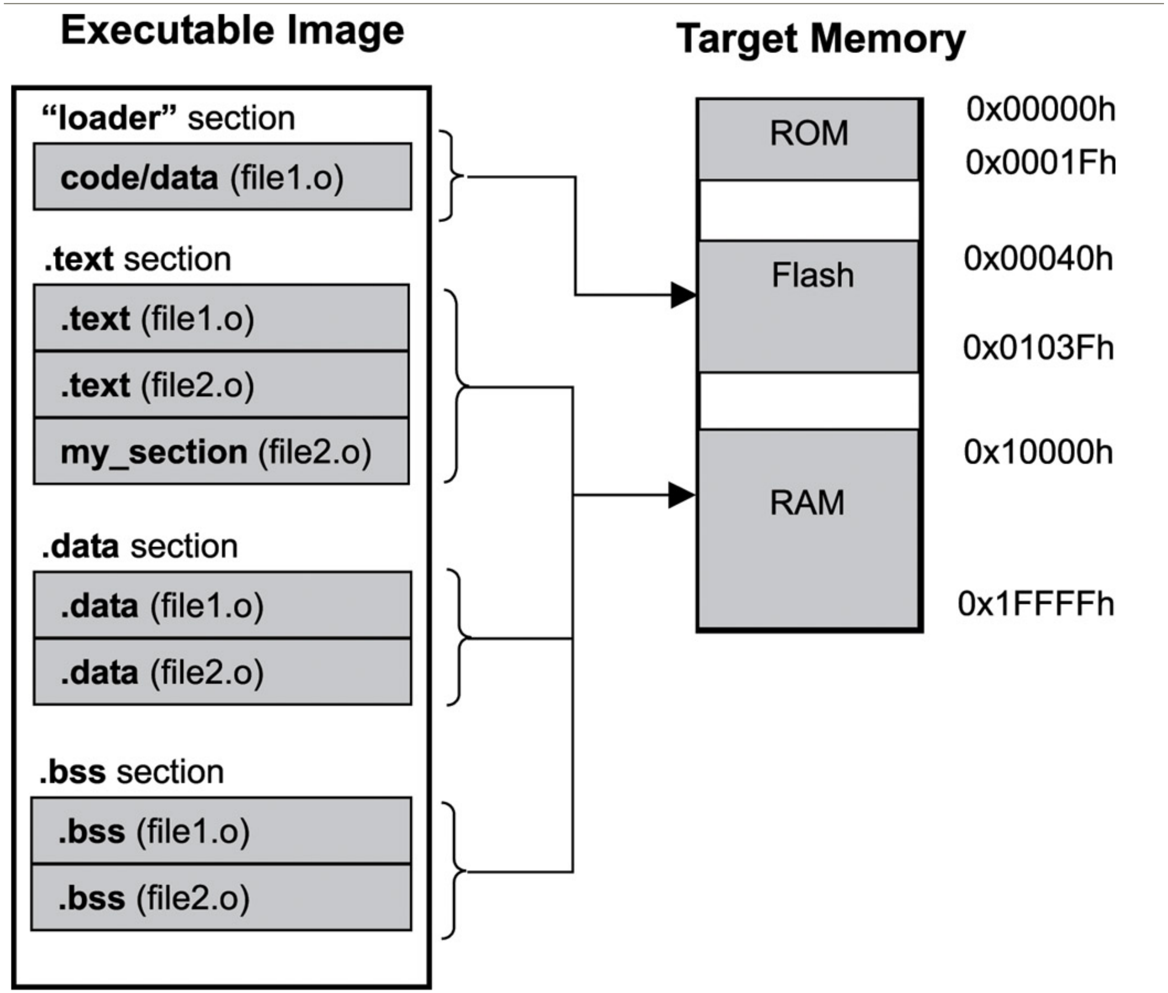
Load Address vs. Run Address

- An embedded software may be stored in ROM
- However, loader may copy the initialized data and code to the RAM
  - Modified data must reside in RAM
  - For faster execution speed, programs must execute out of RAM
- Load address: the address in ROM
- Run address: the location where the section is at the time of execution
  - Linker uses the run address for symbol resolution
- Load address may be the same as the run address
  - Embedded software are directly downloaded to the memory for immediate execution

Simplified Schematic and Memory Map for a Target System



Mapping an Executable Image into the Target System



A Case Study: ADS Scatter File

```
LOAD_ROM 0x0000 0x8000
{
        ROM 0x0000 0x8000
        {
                part1.o (+RO)
        }
        SRAM 0x8000 0x8000
        {
                part2.o (+RO)
        }
        DRAM 0x200000 0x400000
        {
                * (+RW, +ZI)
        }
}
```

Instructions are kept in ROM

Critical instructions are moved to SRAM

All data are moved to DRAM

Embedded System Initialization

Embedded System Initialization

- Image Transfer from the Host to the Target System
- Target System Tools
- Target Boot Process
- Target System Software Initialization Sequence

Image Transfer from the Host to the Target System (1/2)

- Loading Process: transfer an executable image from the host onto the target
  - Programming the image into EEPROM or flash
  - Downloading the image over a serial (RS-232) or network connection
    - Host: a data transfer utility
    - Target: a loader, a monitor or a debug agent
  - Download the image through either a JTAG (Joint Test Action Group) or BDM (Background Debug Mode) interface
- For the final product, the embedded software is stored in ROM or flash

Image Transfer from the Host to the Target System (2/2)

- If a system has both ROM and flash
  - Set jumpers to control which memory chip the processor uses to start its first set of instructions upon reboot
- However, the final product method is impractical during the development stage
  - Reprogramming the EEPROM or the flash memory is time consuming
- Solution
  - Transfer the image directly into the target system’s RAM memory
  - Achieved by
    - Serial or network connection
    - JTAG or BDM solution

Embedded System Initialization

- Image Transfer from the Host to the Target System
- Target System Tools
  - Embedded Loader
  - Embedded Monitor
  - Target Debug Agent
- Target Boot Process
- Target System Software Initialization Sequence

Embedded Loader (1/2)

- At the early development phase, a common approach is write a loader program for the target and use it to download the image from the host system
- Embedded Loader
  - Download the image from the host system to the target system
  - The loader is often programmed into ROM
- To communicate with the host system to download the image
  - Require a data transfer protocol between the host utility and the embedded loader
- Embedded loader may download the image to
  - RAM memory
  - Flash memory if the loader has the flash programming capability

Embedded Loader (2/2)

- The downloading medium can be
  - Serial port
  - Network connection (Ethernet, FTP, TFTP protocols)
- However, before the loader can execute, there must be a boot image to initialize the target
- Boot image
  - Part of the ROM chip is occupied by the boot image
  - Consist of the code that executes when the system powers up
    - Initialize the required peripheral devices
    - Initialize the memory system for downloading the image
    - Initialize the interrupt controller and install default interrupt handler
  - Prepare the system to execute the loader

Embedded Monitor (1/2)

- An alternative to the boot image plus embedded loader is to use an embedded monitor
- Furthermore, embedded monitor enable developers to examine and debug the target system at run time
- Thus, an embedded monitor
  - Consists of boot image plus embedded loader
  - Adds the interactive debug capability

Embedded Monitor (2/2)

- How to help developer to examine and debug the target system at run time?
- Solution:
  - Embedded monitor defines a set of commands that can be accessible through a terminal emulation program over the serial line
    - Download the image
    - Read from and write to system memory locations
    - Read and write system registers
    - Set and clear different types of breakpoints
    - Single-step instructions
    - Reset the system

Target Debug Agent

- Target debug agent, or debug agent
  - Embedded monitor + visual source-level debug capability for the host debugger
- Thus, a target debug agent must provide enough information for the host debugger to provide visual source-level debug capability
- For example, a debug agent has built-in knowledge of the RTOS objects and services
  - Allow the developer to explore such object and services fully and visually

Department of Computer Science and Information Engineering, Chang Gung University

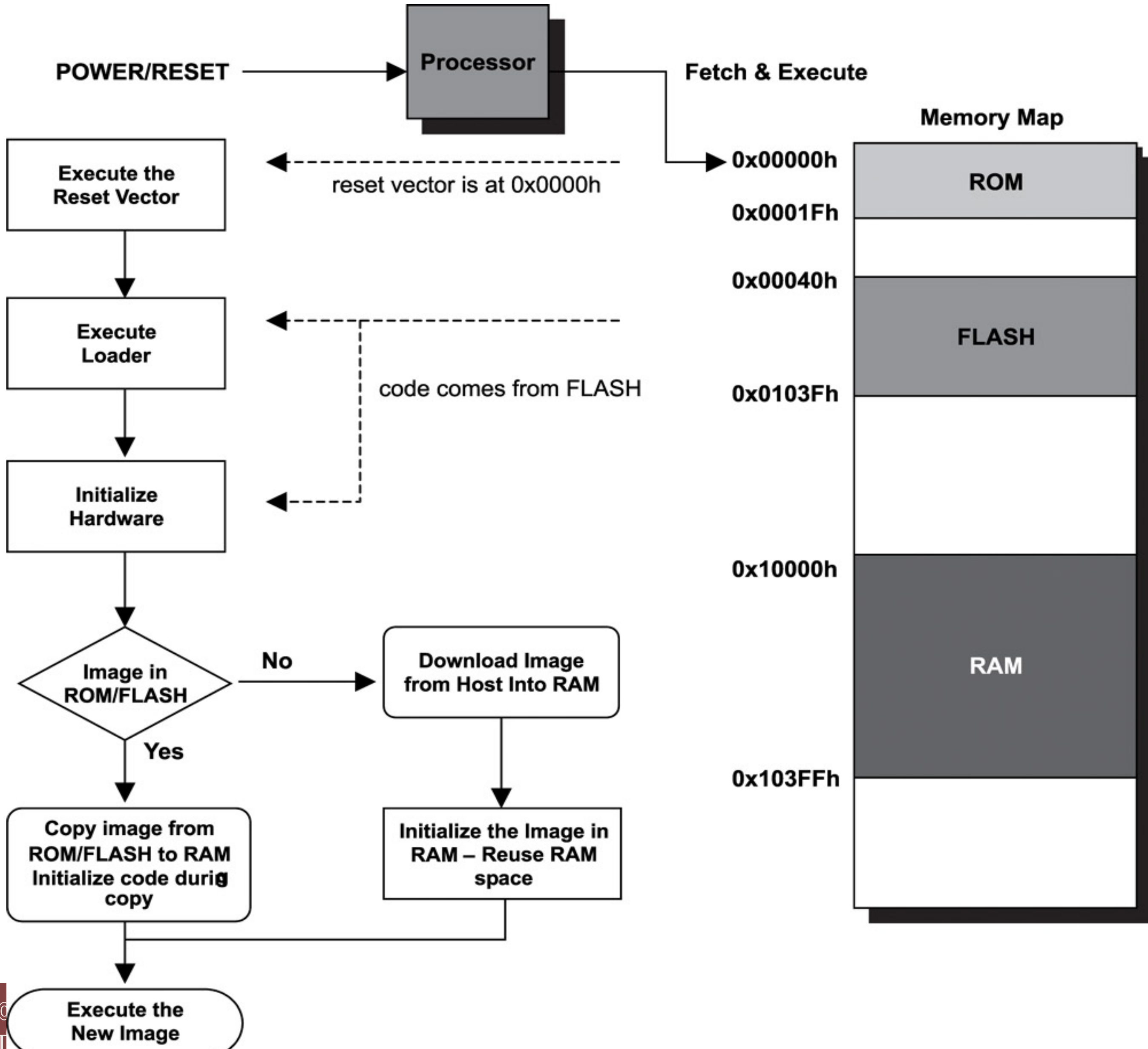
Embedded System Initialization

- Image Transfer from the Host to the Target System
- Target System Tools
- Target Boot Process
- Target System Software Initialization Sequence

Target Boot Process

- We give an example to show an embedded system boot process
    - Note that, each embedded system may have its own booting scenario
- Assume
    - The reset vector is contained in ROM and mapped to 0x00000h
        - The codes are executed when an embedded system powers on
        - Usually a jump into another part of memory space where the real initialization code is found
    - The loader is contained in flash and is mapped to 0x00040h
    - A loader performs
        - System bootstrapping
        - Image downloading
        - Initialization

Example: Bootstrap Overview



Steps of the Example Bootstrap Process (1/3)

- Power on or reset
  - Processor fetch and executes code from 0x00000h
    - Reset vector in ROM
- The code in reset vector is a jump instruction to 0x00040h
  - Loader in flash
- The code in loader first initialize hardware to put the system into a known state
  - Processor registers are set with default value
  - Disable interrupt
  - Initializes the main memory and the caches
  - Perform limited hardware diagnostics on those devices needed for its operation

Steps of the Example Bootstrap Process (2/3)

- Then, the loader optionally can copy itself from the flash memory into the RAM
  - Since RAM is faster than flash
- Besides, the loader must copy and reserve the initialized and uninitialized data sections of loader from flash to RAM
  - Copy the content of the initialized data section (.data) to RAM
  - Reserve spaces for unitialized data section (.bss) in RAM
- Keep .const section in flash or RAM

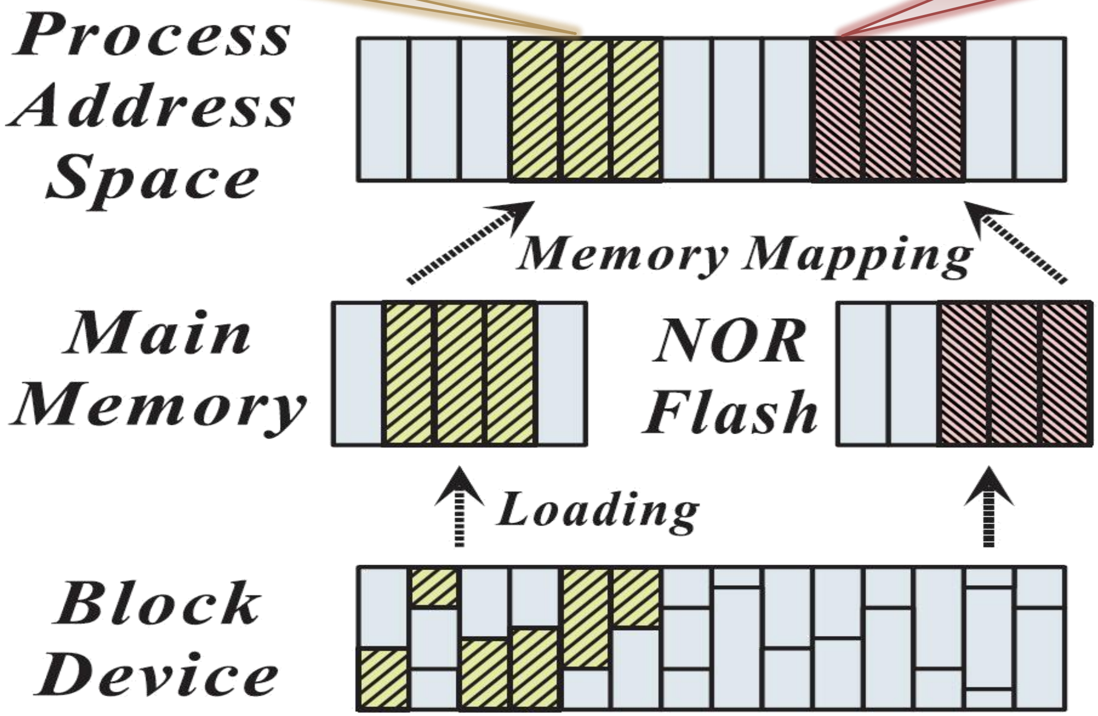
Steps of the Example Bootstrap Process (3/3)

- The next step is to initialize the system devices
  - Only the necessary devices that the loader requires are initialized
    - For example, network controller if loader uses network to download image
  - Fully initialization is left until the downloaded image perform its system initialization
- Now, the loader can transfer the application image to the target system
  - Application image: kernel+ application code
  - Application image may come from
    - Read-only memory devices on the target
    - The host development system

XIP and SnD

SnD: Store and Download

XIP: eXection In Place



Target Image Execution Scenarios

- Three image execution scenarios
  - Execute from ROM while using RAM for data
  - Execute from RAM after being copied from ROM
  - Execute from RAM after being downloaded from a host system

Image Running on ROM (1/2)

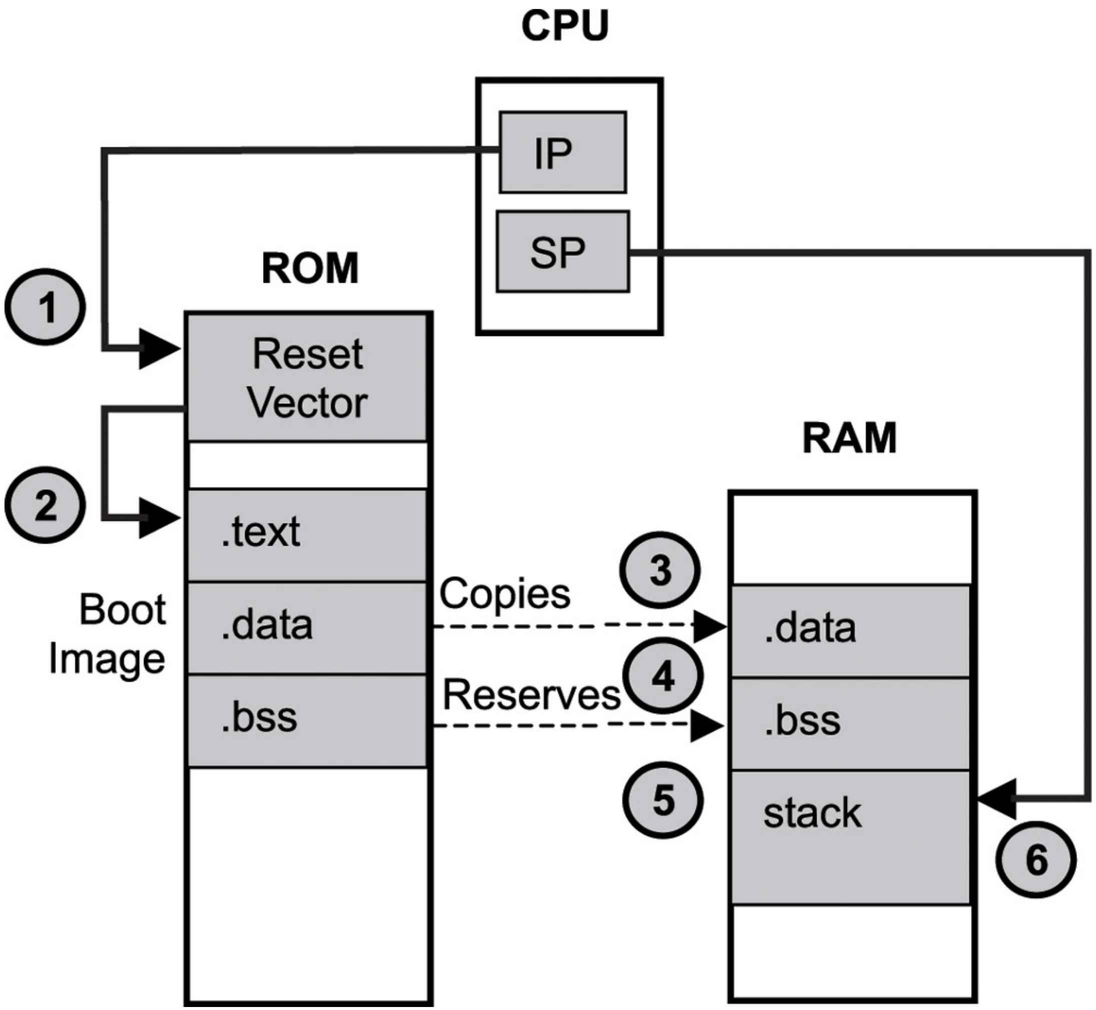


Image Running on ROM (2/2)

- Boot Sequence
  1. The CPU's IP register is hardwired to execute the first instruction in memory, i.e., the reset vector
  2. The reset vector jump to the first instruction of the .text section of boot image
     - Initialize the memory system (including the RAM)
  3. The .data section is copied to RAM
  4. Reserve space if RAM for the .bss section
  5. Reserve stack space in RAM
  6. Set SP register to the beginning of the newly created stack

Image Transferred from ROM Running on RAM (1/2)

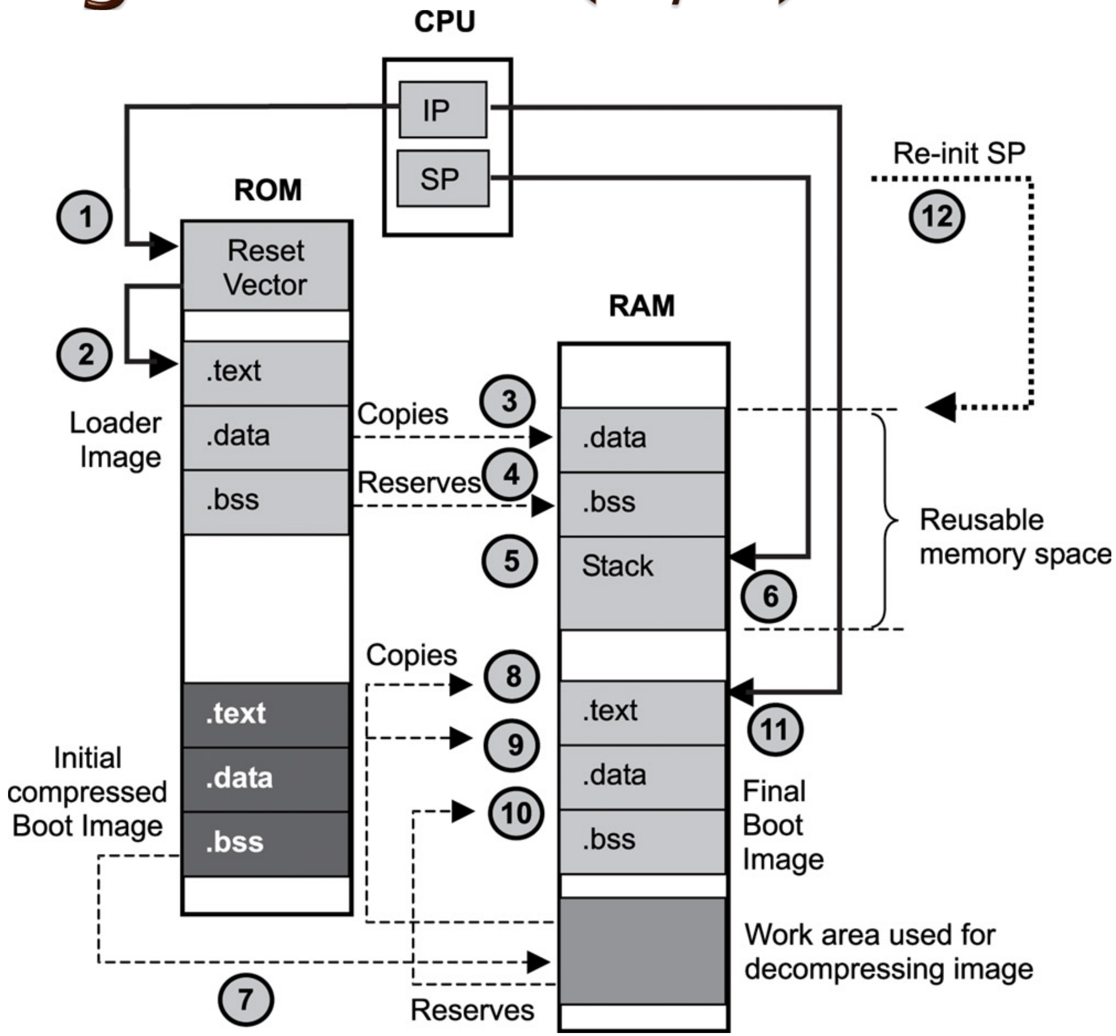


Image Transferred from ROM Running on RAM (2/2)

- The boot loader transfers an application image from ROM to RAM for execution
  - The application image is usually compressed in ROM to reduce the storage space required
- Boot sequence
  7. Copy the Compressed application image from ROM to RAM in a work area
  8. Decompress and initialize the application image(1)
  9. Decompress and initialize the application image(2)
  10. Decompress and initialize the application image(3)
  11. The loader transfers control to the image using a processor-specific jump instruction
  12. Recycle the memory area occupied by the loader and the work area
      - May also reinitialize the SP to point to the memory area occupied by the loader to use it as the stack space

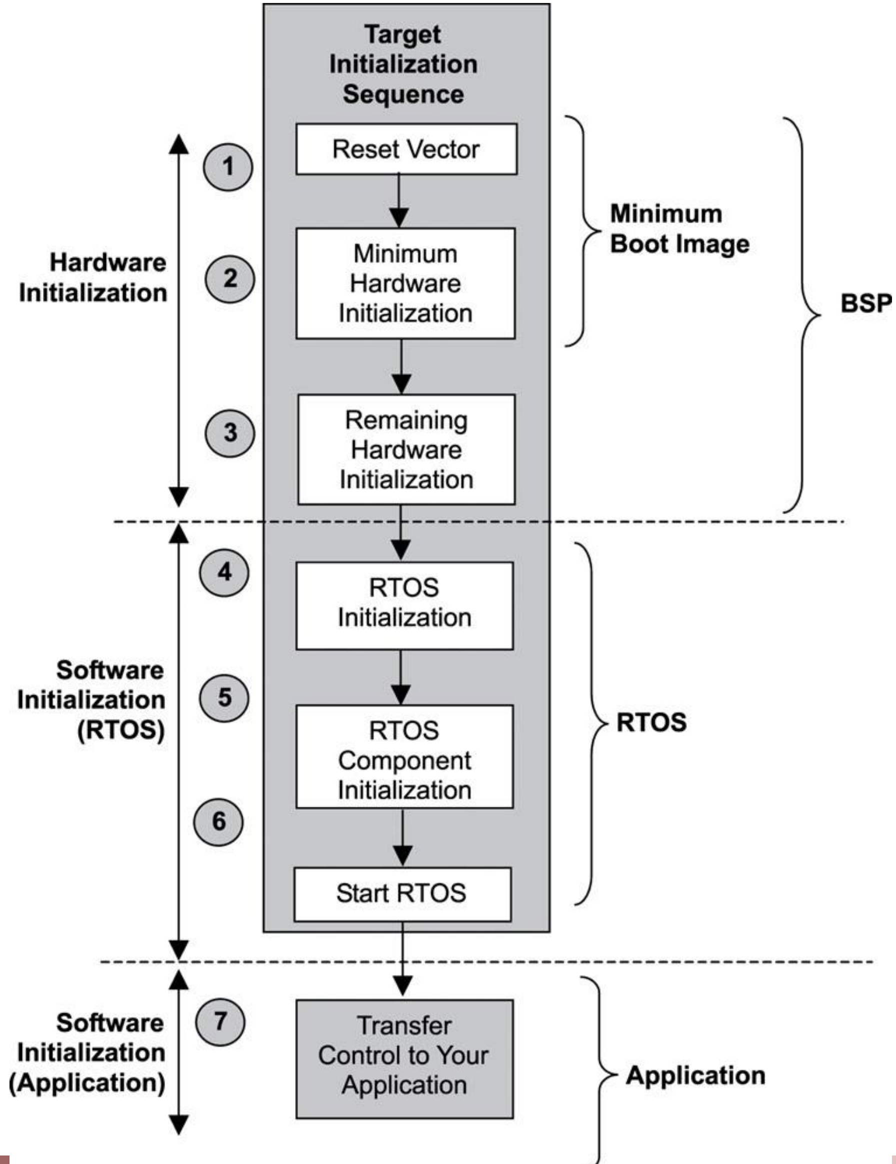
Embedded System Initialization

- Image Transfer from the Host to the Target System
- Target System Tools
- Target Boot Process
- Target System Software Initialization Sequence

Target System Software Initialization Sequence (1/2)

- Target image may consists of
  - Board support package (BSP)
    - A full spectrum of drivers for the hardware components/devices
  - RTOS
  - Other embedded modules
    - File system, networking, …
  - Applications
- Main steps to initialize the system
  - Hardware initialization
  - RTOS initialization
  - Application initialization

Software Initialization Process



Hardware Initialization (1/3)

- Power on-> reset vector -> boot image
  - Initialize the minimum hardware required to get the boot image to execute
  - Steps 1 and 2 in the previous slide
    - Starting execution at the reset vector
    - Putting the processor into a known state by setting the appropriate registers
      - Getting the processor type
      - Getting or setting the CPU's clock speed
    - Disabling interrupts and caches
    - Initializing memory controller, memory chips, and cache units
      - Getting the start address for memory
      - Getting the size of memory
      - Performing preliminary memory tests, if required

Hardware Initialization (2/3)

- Then
  - Boot sequence may copy and decompress the sections of code to RAM
  - It must copy and decompress its data to RAM
- Finally, initialize other hardware components
  - Step 3
    - Initializing interrupt handlers
    - Initializing bus interfaces, such as PCI, USB…
    - Initializing board peripherals such as serial, LAN and SCSI

Hardware Initialization (3/3)

- Initial Boot Sequence
  - Steps 1 and 2
  - Mainly initialize the CPU and memory subsystem
- BSP Initialization Phase
  - Also called hardware initialization
  - Steps 1 to 3

RTOS Initialization

- Initializing the RTOS
  - Steps 4 to 6
  - Initializing different RTOS objects and services
    - Task objects
    - Semaphore objects
    - Message-queue objects
    - Timer services
    - Interrupt services
    - Memory-management services
  - Creating necessary stack for RTOS
  - Initializing additional RTOS extensions
    - TCP/IP stack or file system
  - Starting the RTOS and its initial tasks

Application Software Initialization

- Finally, transfer control to the application
  - RTOS calls a predefined function implemented by the application
  - Then, the application software goes through its initialization
    - Declared and implemented necessary objects, services, data structures, variables, and other constructs