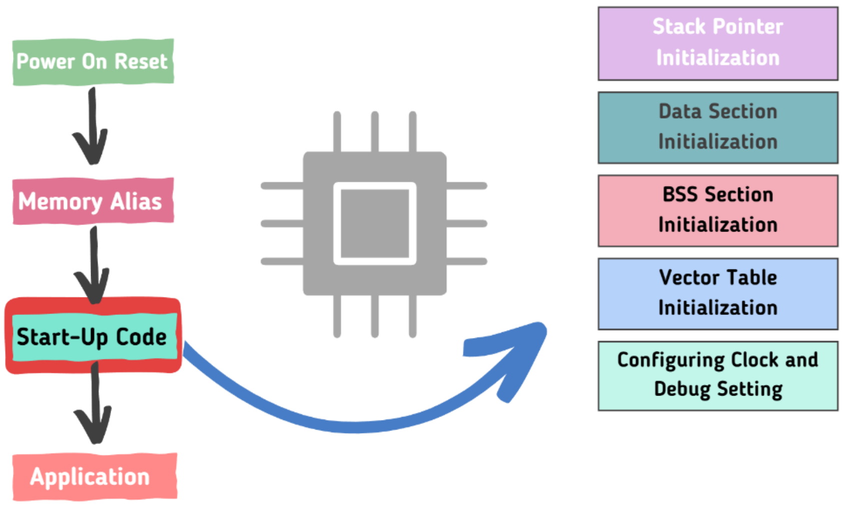
C STARTUP
IN EMBEDDED SYSTEM

In embedded systems, the startup code is responsible for initializing the hardware and software environment for the application code to run. This code is the first to execute immediately after a Power-On Reset (POR) in a microcontroller (MCU) and is considered fundamental for system operation. Startup code is typically lightweight and written in assembly language or low-level C code


Power On Reset
Memory Alias
Start-Up Code
Application
Stack Pointer Initialization
Data Section Initialization
BSS Section Initialization
Vector Table Initialization
Configuring Clock and Debug Setting

▶ **The startup code performs several essential tasks to prepare the system for running the target application:**

Stack Pointer Initialization:

- Sets the stack pointer to a known memory location in RAM, where it will be used to store function call stacks and local variables.

Data Section Initialization:

- Copies initialized global and static variables from ROM to RAM

Zeroing the BSS Section:

- Clears the BSS section, which contains uninitialized global and static variables, ensuring that all variables are initialized to zero.

Vector Table Initialization:

- Configures the vector table, which contains function pointers for interrupt callbacks, so that the designated interrupt service routines are executed when an interrupt occurs.

Enabling the Interrupt System:

- Initializes and enables the interrupt system to allow the application code to respond to external events such as timer interrupts or serial communication.

Configuring the Clock and Debug Settings:

- Sets up default or complete clock settings to ensure the application code runs at the correct speed and configures debug settings, such as enabling/disabling JTAG/SWD for debugging and UART for debug prints.

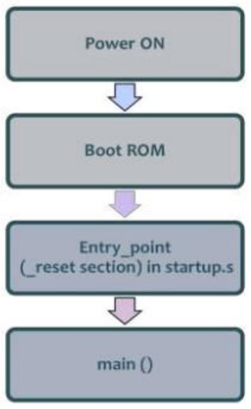
Jumping to the Main Function:

- After completing all initialization tasks, the startup code jumps to the main function, which is the entry point of the application code.

summary

Startup a small block of assembly language code that prepares the way for the execution the main() user code to run.

Startup code for C programs usually consists of the following series of actions:



1) Startup code have Vector Section
2) Disable all interrupts.
3) Copy any initialized data from ROM to RAM.
4) Zero the uninitialized data area.
5) Allocate space for and initialize the stack.
6) Initialize the processor's stack pointer.
7) Create and initialize the heap
8) Enable interrupts.
9) Call main.

the startup code will also include a few instructions after the call to main. These instructions will be executed only in the event that the high-level language program exits (i.e., the call to main returns).

Before transferring control to C code, the following have to be setup correctly.

1) Stack (r13"SP")
2) Global variables
   - Initialized .data
   - Uninitialized .bss
3) Read-only data .rodata
4) Then force the PC register to jump on the main functions

in a typical executable or object file

- memory is divided into several sections, each serving a different purpose.

Here’s a breakdown of common sections you might encounter:

1. Text Section (.text)

- **Storage**: Stored in the executable’s code segment in memory.
- **Characteristics**: This section is mapped into memory as read-only and executable to prevent modifications and ensure code security. It is loaded into the process's address space when the program starts.

2. Data Section (.data)

- **Storage**: Stored in the data segment of memory.
- **Characteristics**: This section contains initialized global and static variables. It is mapped into memory as read-write so that these variables can be modified during program execution.

3. BSS Section (.bss)

- **Storage**: Stored in the data segment of memory, but initialized to zero.
- **Characteristics**: This section holds uninitialized global and static variables. The memory is zeroed out by the loader when the program starts.

4. Read-Only Data Section (.rodata)

- **Storage**: Stored in a read-only data segment of memory.
- **Characteristics**: This section contains constant data and literals that should not be modified. It is mapped as read-only and is part of the process's address space.

5. Heap

- **Storage**: Managed dynamically at runtime.
- **Characteristics**: The heap grows and shrinks as needed during the execution of the program. It is used for dynamic memory allocation and is typically located in the upper part of the address space.

6. Stack

- **Storage**: Managed dynamically at runtime.
- **Characteristics**: The stack is used for function call management and local variable storage. It grows and shrinks with function calls and returns, and is typically located in the lower part of the address space.

7. Symbol Table

- **Storage**: Not part of the executable memory but included in object files and executables.
- **Characteristics**: Contains metadata about symbols used in the program and is used by the linker and debugger. This information is typically stored in a separate section or file.

8. **Debug Information**

- ***Storage***: Included in object files or executables but not loaded into memory for program execution.
- ***Characteristics***: Contains additional information for debugging purposes and is used by debugging tools.

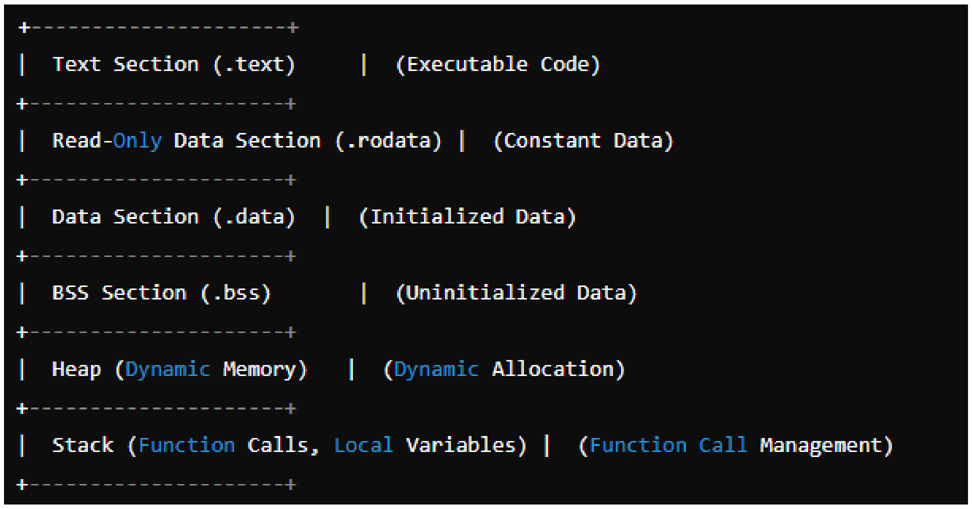
9. **Dynamic Linker Section**

- ***Storage***: Part of the executable file and loaded into memory by the dynamic linker.
- ***Characteristics***: Contains information about shared libraries and relocation. It is used at runtime to manage dynamic linking.

10. **Exception Handling Section**

- ***Storage***: Included in the executable file or object files.
- ***Characteristics***: Contains data and tables for exception handling mechanisms, used by runtime libraries and the operating system to manage exceptions.

Memory Layout Example



```
+---------------------+
|  Text Section (.text)     |  (Executable Code)
+---------------------+
|  Read-Only Data Section (.rodata) |  (Constant Data)
+---------------------+
|  Data Section (.data)  |  (Initialized Data)
+---------------------+
|  BSS Section (.bss)       |  (Uninitialized Data)
+---------------------+
|  Heap (Dynamic Memory)   |  (Dynamic Allocation)
+---------------------+
|  Stack (Function Calls, Local Variables) |  (Function Call Management)
+---------------------+
```

Sections that are not stored in flash memory

1. **Heap**

- **Purpose**: Used for dynamic memory allocation during program execution (e.g., using malloc, calloc, realloc, free in C).
- **Storage**: Managed in RAM. The heap grows and shrinks as needed during the program's execution.

2. **Stack**

- **Purpose**: Used for function call management, local variables, and return addresses.
- **Storage**: Managed in RAM. The stack dynamically grows and shrinks as functions are called and return.

3. **BSS Section (.bss)**

- **Purpose**: Contains uninitialized global and static variables.
- **Storage**: Although its size is defined in the executable file, the actual data is not stored in flash memory. Instead, it is allocated and zeroed out in RAM when the program is loaded.

4. **Dynamic Linker Information**

- **Purpose**: Contains metadata for dynamic linking of shared libraries (e.g., symbols, relocation information).
- **Storage**: Not typically stored in flash memory but loaded into RAM by the dynamic linker at runtime.

5. **Exception Handling Data**

- **Purpose**: Contains information related to exception handling and runtime error management.
- **Storage**: Typically stored in RAM during execution but may be included in the executable file as part of metadata.

6. **Debug Information**

- **Purpose**: Includes symbols, source code mapping, and other data used for debugging.
- **Storage**: Stored in the executable or object files but not in flash memory during runtime. It is usually omitted in release builds.

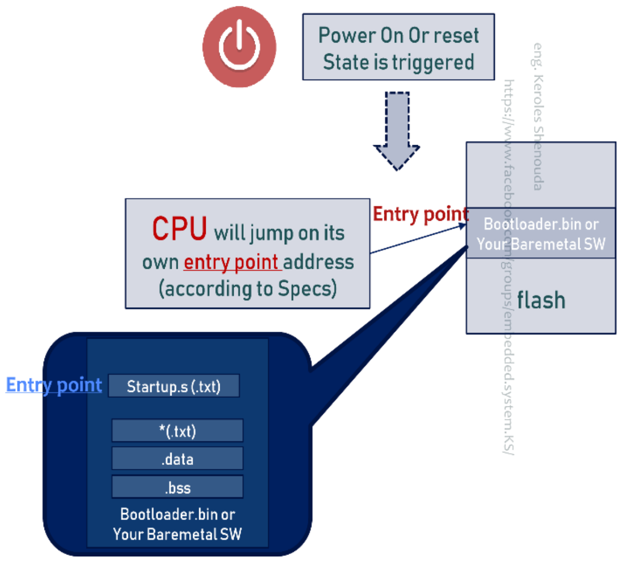
Booting Sequence

Most processors have a default address from which the first bytes of code are fetched upon application of power and release of reset.

Hardware designers use this information to arrange the layout of Flash memory on the board and to select which address range(s) the Flash memory responds to.

This way, when power is first applied, code is fetched from a well-known and predictable address, and software control can be established

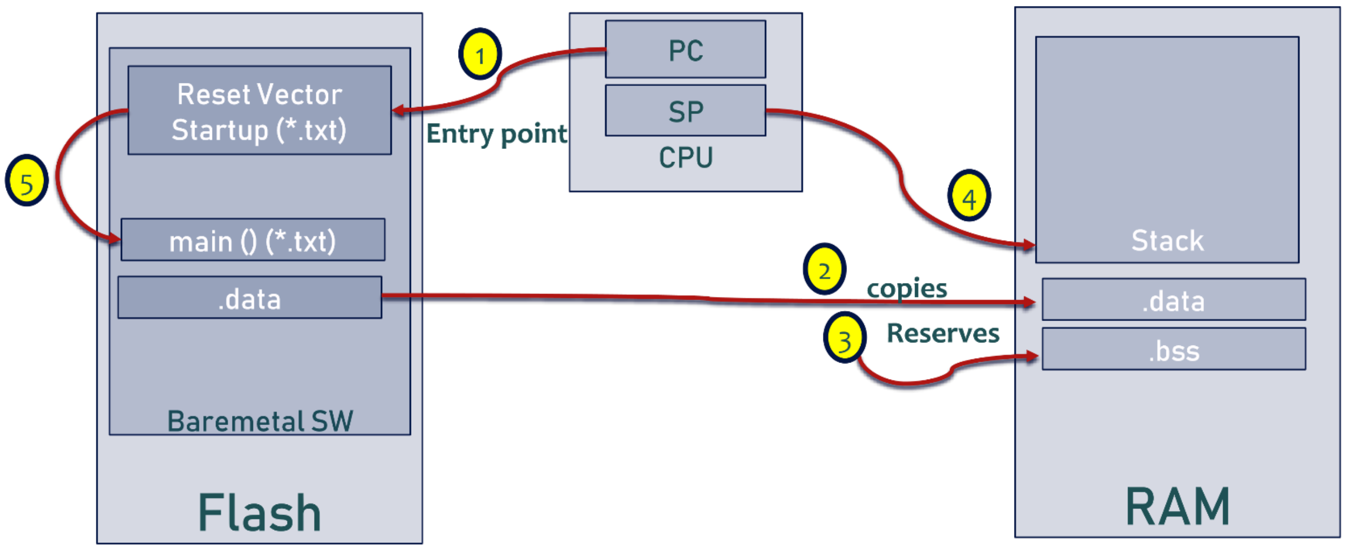
Default Address (Entry Point**)**: Most processors have a predefined address (known as the reset vector) where they start fetching instructions when power is first applied or a reset occurs. This address is specified in the processor's architecture and is where the processor expects to find the initial executable code.


Power On Or reset State is triggered
CPU will jump on its own entry point address (according to Specs)
Entry point
Bootloader.bin or Your Baremetal SW
flash
Entry point
Startup.s (.txt)
*(.txt)
.data
.bss
Bootloader.bin or Your Baremetal SW
eng. Keroles Shenouda
https://www.facebook.com/groups/embedded.system.KS/

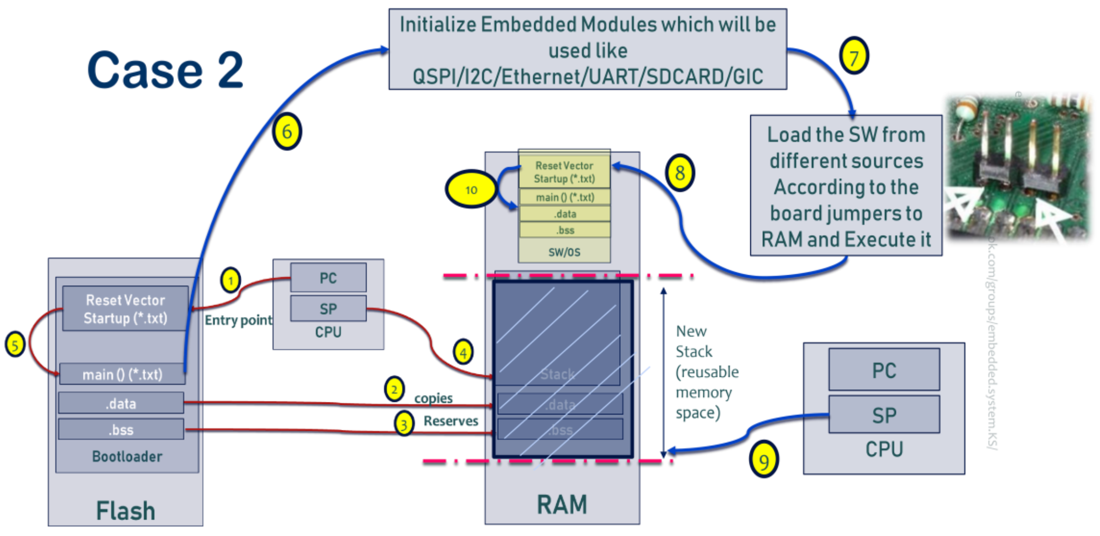
Boot Sequence (Case 1)

In Some SoC, you can burn your **baremetal SW** directly on the **Flash memory** and in this case you can put the reset section on the CPU entry address **(**Entry Point**)** based on the SoC DataSheet. You can do it(reset: in CPU entry point) by linker script.

Burn the Software: Once your linker script is configured correctly, compile and link your software, then burn it into the Flash memory of the SoC. The processor will fetch the initial code from the address specified by the entry point in the linker script.



Boot Sequence (Case 2)


Case 2
Initialize Embedded Modules which will be used like QSPI/I2C/Ethernet/UART/SDCARD/GIC
Load the SW from different sources According to the board jumpers to RAM and Execute it
Reset Vector Startup (*.txt)
main () (*.txt)
.data
.bss
SW/OS
Reset Vector Startup (*.txt)
main () (*.txt)
.data
.bss
Bootloader
Flash
Entry point
PC
SP
CPU
copies
Reserves
Stack
.data
.bss
RAM
New Stack (reusable memory space)
PC
SP
CPU
1
2
3
4
5
6
7
8
9
10

Workflow

1. **Power-Up/Reset**: When the SoC is powered on or reset, the processor begins execution from the BootROM code at the entry point.
2. **BootROM Execution**: The BootROM performs basic initialization and might also check for the presence of a bootloader.
3. **Bootloader Execution**: The BootROM hands control to the bootloader, which then completes hardware initialization and loads the operating system or application.
4. **OS Execution**: Finally, the bootloader transfers control to the operating system, which takes over the system's operation.

Bootloader

- **What is it?**: A bootloader is a program or code stored in modifiable memory, such as Flash memory. It prepares the system to run a full operating system or other software.
- **Function**: The bootloader handles more complex tasks after the BootROM phase, such as full hardware initialization, loading the operating system from persistent storage into RAM, and transferring control to the operating system.
- **Characteristics**: It can be modified and updated over time. There may be multiple stages of the bootloader, such as a "primary bootloader" and a "secondary bootloader."

Loading of an application binary, typically an operating system kernel, can be performed in various ways:

- **From Flash Storage**: Loading directly from onboard Flash memory.
- **From the Network**: Fetching the binary via network protocols.
- **From an SD Card or Other Non-Volatile Storage**: Loading from removable media or other types of persistent storage.
- **From a USB Client**: Fetching the binary through a USB connection.
- **Possibly Decompression**: Decompressing the application binary if it is stored in a compressed format.
- **Execution of the Application**: Transferring control to the loaded application to start its execution.

BootROM Code

- **What is it?**: BootROM code is fixed code located in a Read-Only Memory (ROM) that is embedded in the SoC (System on Chip). This code is automatically executed when the system is powered on or reset.
- **Function**: BootROM performs very basic tasks such as initializing the processor and essential I/O devices. It may also check for the presence of an external bootloader and initiate its loading if available.
- **Characteristics**: It is not easily changeable because it is stored in ROM.

Relationship Between BootROM and Bootloader

- **Boot Sequence**: When the system powers up, the processor begins by executing the BootROM code. BootROM performs initial system setup and may look for a bootloader.
- **Control Transfer**: After completing its basic tasks, the BootROM hands control over to the bootloader, which then completes system setup and loads the operating system or other required software.
- One main difference from bootrom is that it's usually in writable flash and can be replaced or upgraded.

By using BootROM and bootloaders, SoCs ensure a robust initialization process, allowing complex software to run smoothly after the initial hardware setup is complete.

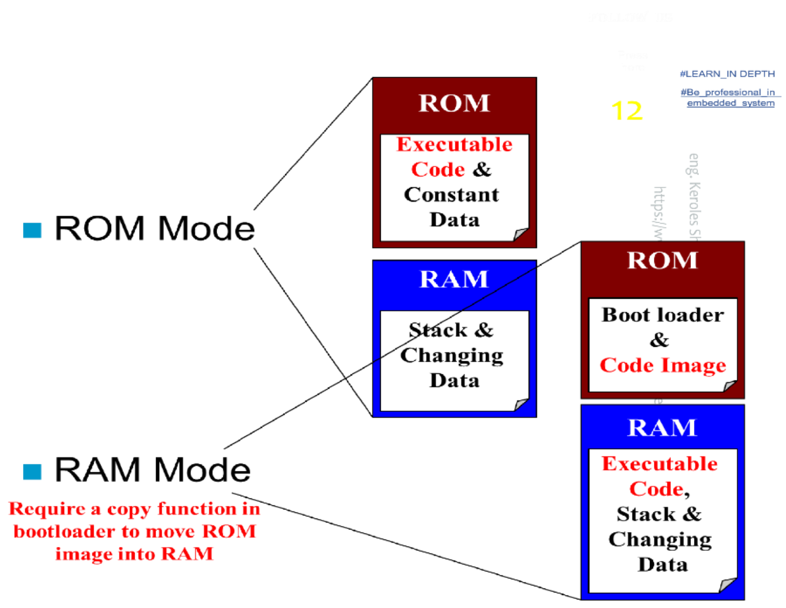
Running Mode

ROM Mode

- Simple
- Require smaller memory
- Fixed code address
- Relatively Small Code

RAM Mode

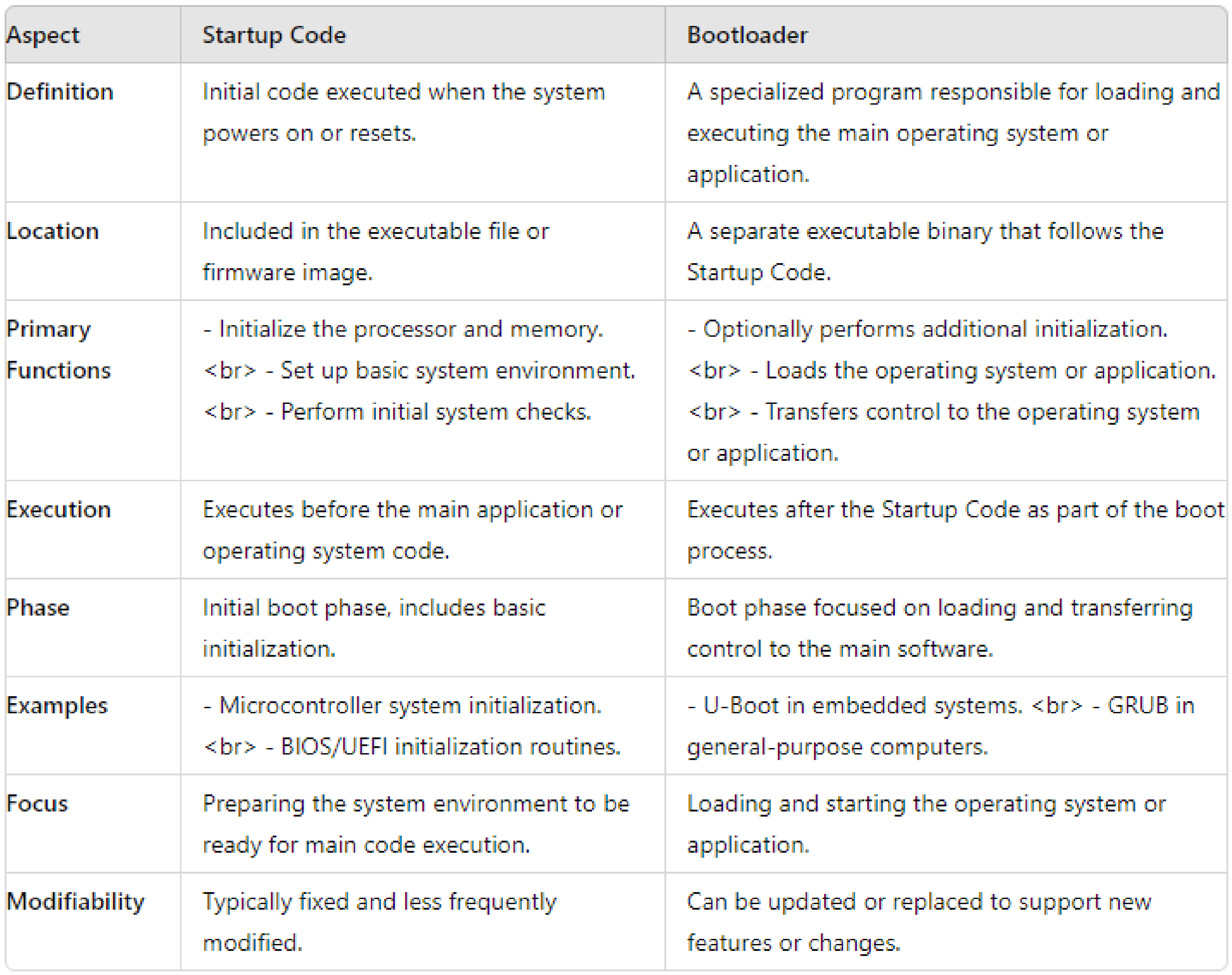
- Complex
- Re-locatable code
- Faster
- Large Code (SDRAM)


#LEARN_IN DEPTH
#Be_professional_in_embedded_system
ROM Mode
ROM
Executable Code & Constant Data
RAM
Stack & Changing Data
RAM Mode
Require a copy function in bootloader to move ROM image into RAM
ROM
Boot loader & Code Image
RAM
Executable Code, Stack & Changing Data

Is it possible to have an embedded system without BootROM code, and under what circumstances might this occur?

Yes, an embedded system can operate without BootROM if it uses alternative boot mechanisms, direct firmware execution, or external boot devices.

BootLoader Vs Startup



Aspect	Startup Code	Bootloader
Definition	Initial code executed when the system powers on or resets.	A specialized program responsible for loading and executing the main operating system or application.
Location	Included in the executable file or firmware image.	A separate executable binary that follows the Startup Code.
Primary Functions	- Initialize the processor and memory. - Set up basic system environment. - Perform initial system checks.	- Optionally performs additional initialization. - Loads the operating system or application. - Transfers control to the operating system or application.
Execution	Executes before the main application or operating system code.	Executes after the Startup Code as part of the boot process.
Phase	Initial boot phase, includes basic initialization.	Boot phase focused on loading and transferring control to the main software.
Examples	- Microcontroller system initialization. - BIOS/UEFI initialization routines.	- U-Boot in embedded systems. - GRUB in general-purpose computers.
Focus	Preparing the system environment to be ready for main code execution.	Loading and starting the operating system or application.
Modifiability	Typically fixed and less frequently modified.	Can be updated or replaced to support new features or changes.

Question:

Difference Between Bootloader and Bare-Metal Software

Bootloader: A program used for system initialization and starting the main application, with potential features for firmware updates and system settings management.

Bare-Metal Software: Programs that operate directly on hardware, handling all system functions without an operating system.

Is a bootloader required for every microcontroller project, and in what situations is a bootloader necessary or not?

Answer:

No, a bootloader is not always required. A bootloader should be used in the following situations:

- **Firmware Updates**: When updates need to be applied via external storage.
- **Configuration Management**: To load settings from non-volatile memory.
- **Security Enhancement**: To verify software signatures for integrity.
- **Additional Functions**: Such as debugging or handling multiple storage media.

A bootloader may be unnecessary in the following cases:

- **Static Applications**: Where no post-deployment updates are needed.
- **Resource-Limited Systems**: Where minimizing resource usage is critical.
- **Controlled Environments**: Where no frequent updates are required.
- **Applications Not Requiring Advanced Security**

Is startup code required for every microcontroller project, and in what situations is startup code necessary or not?

Answer:

Yes, startup code is typically required for every microcontroller project. It performs essential initialization tasks to prepare the microcontroller for running the application code. Startup code is necessary in the following situations:

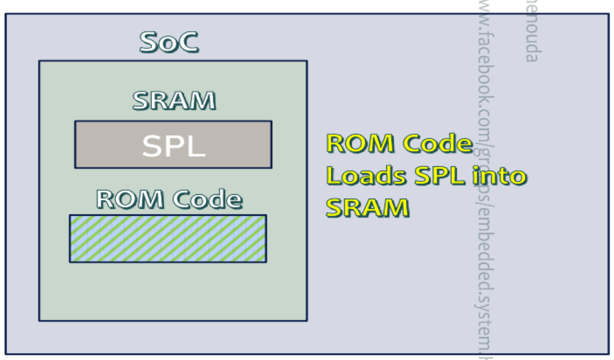
- **Initialization**: To set up the stack pointer, configure memory sections, and initialize global and static variables.
- **Interrupt Management**: To configure the interrupt system and set up the vector table.
- **Hardware Setup**: To configure critical hardware components like clocks and debugging interfaces.

Startup code may be less critical in the following cases:

- **Integrated Development Environments (IDEs)**: Some IDEs provide default startup code.
- **Operating Systems**: In complex systems where an operating system handles initialization and hardware configuration.
- **Preconfigured Systems**: Where the system is already configured and does not require additional setup.

3 phases boot sequence inside Bootloaders

Phase 1 – ROM code



- ROM code is the initial code executed immediately after a reset or power-on, stored on-chip in the SoC (System on Chip).

1) **Power-On or System Reset**:
   The system is powered on for the first time or reset.

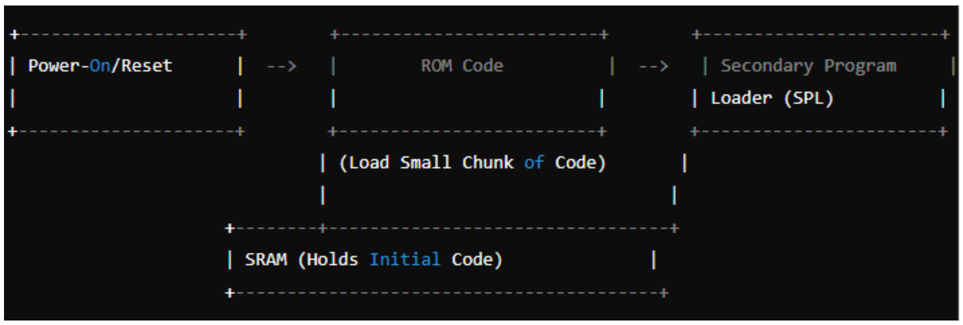
2) **Loading ROM Code**:
   The system starts executing the ROM code stored on the chip (SoC).
   Note: ROM code is fixed and non-modifiable, pre-loaded during chip manufacturing.

3) **Memory Initialization**:
   ROM code typically does not include memory controller initialization (especially for DRAM) and instead works with SRAM.
   SRAM: Static Random-Access Memory on the chip.

4) **Loading Additional Code**:
   ROM code loads a small chunk of code into SRAM from pre-defined locations.
   Possible Sources:

   - NAND Flash memory.
   - Flash memory connected via SPI (Serial Peripheral Interface).
   - MMC device (eMMC or SD card).
   - File named MLO on the first partition of an MMC device.

5) **Executing SPL**:
   At the end of the ROM code phase, the SPL is present in the SRAM and the ROM code jumps to the beginning of that code.
   SPL takes over to perform complete system initialization, including loading the main bootloader.



```
+--------------------+       +----------------------+       +----------------------+
| Power-On/Reset     |  -->  |       ROM Code       |  -->  | Secondary Program    |
|                    |       |                      |       | Loader (SPL)         |
+--------------------+       +----------------------+       +----------------------+
                             | (Load Small Chunk of Code)    |
                             |                              |
                     +-------+------------------------------+
                     | SRAM (Holds Initial Code)            |
                     +--------------------------------------+
```

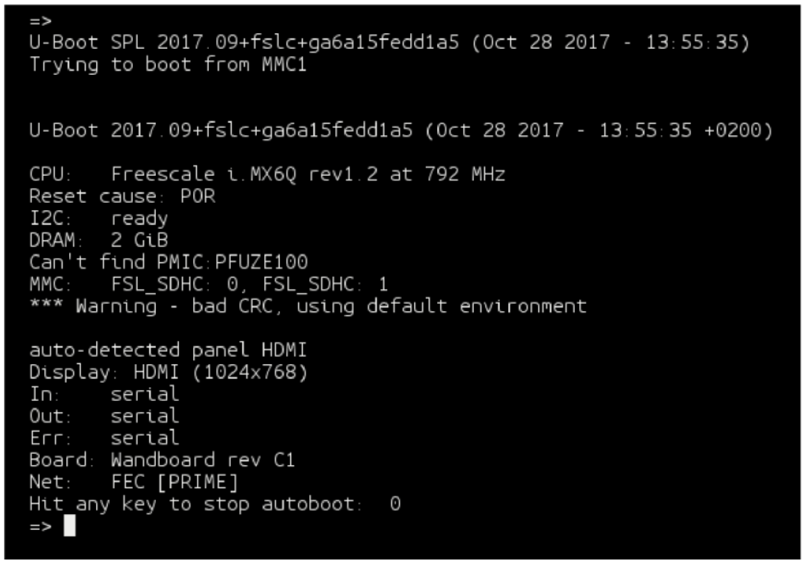
Phase 2 – Secondary Program Loader (SPL)

The SPL must set up the memory controller and other essential parts of the system preparatory to loading the uboot into DRAM.

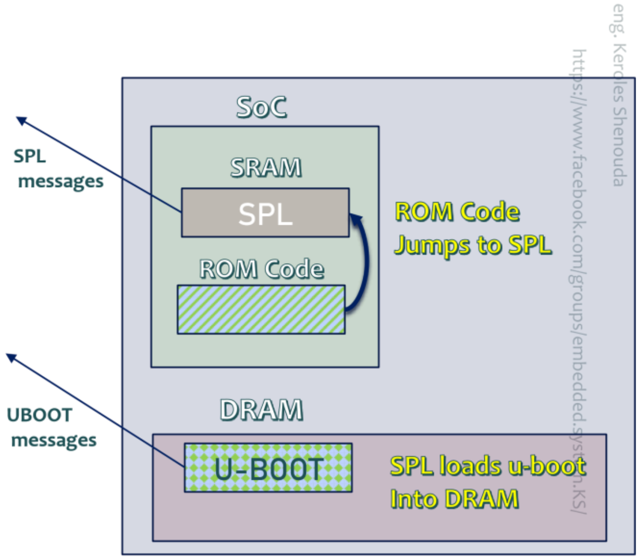
The functionality of the SPL is limited by the size of the SRAM. It can read a program from a list of storage devices.

system drivers built inside spl, it can read well known file names, such as u-boot.img from a disk partition.

The SPL usually doesn't allow for any user interaction, but it may print version information and progress messages, which you can see on the console



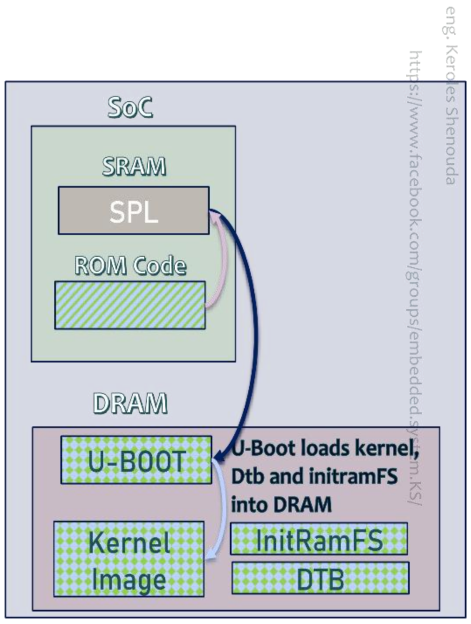
```
=>
U-Boot SPL 2017.09+fslc+ga6a15fedd1a5 (Oct 28 2017 - 13:55:35)
Trying to boot from MMC1


U-Boot 2017.09+fslc+ga6a15fedd1a5 (Oct 28 2017 - 13:55:35 +0200)

CPU:   Freescale i.MX6Q rev1.2 at 792 MHz
Reset cause: POR
I2C:   ready
DRAM:  2 GiB
Can't find PMIC:PFUZE100
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment

auto-detected panel HDMI
Display: HDMI (1024x768)
In:    serial
Out:   serial
Err:   serial
Board: Wandboard rev C1
Net:   FEC [PRIME]
Hit any key to stop autoboot:  0
=>
```



Phase 3 – U-BOOT

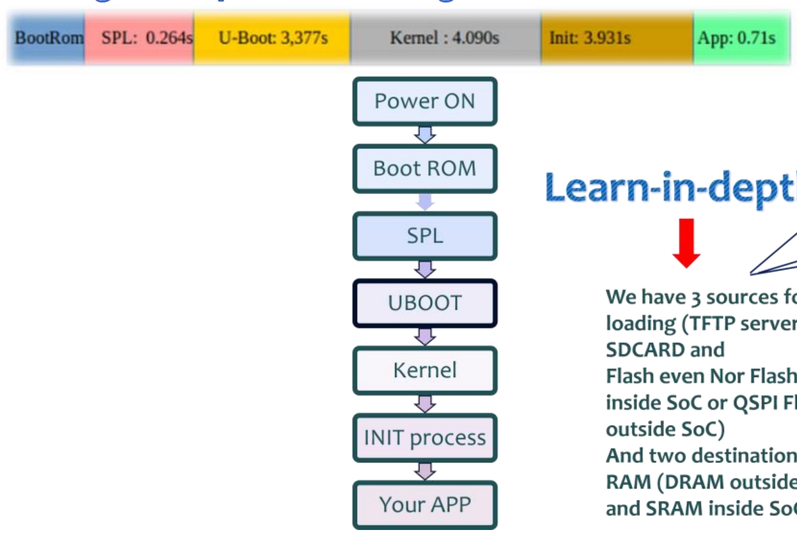
Now, at last, we are running a full bootloader, such as U-Boot

Usually, there is a simple command-line user interface that lets you perform maintenance tasks, such as loading new boot and kernel images into RAM


SoC
SRAM
SPL
ROM Code
DRAM
U-BOOT
U-Boot loads kernel, Dtb and initramFS into DRAM
Kernel Image
InitRamFS
DTB
eng. Kerolles Shenouda
https://www.facebook.com/groups/embedded.system.KS/

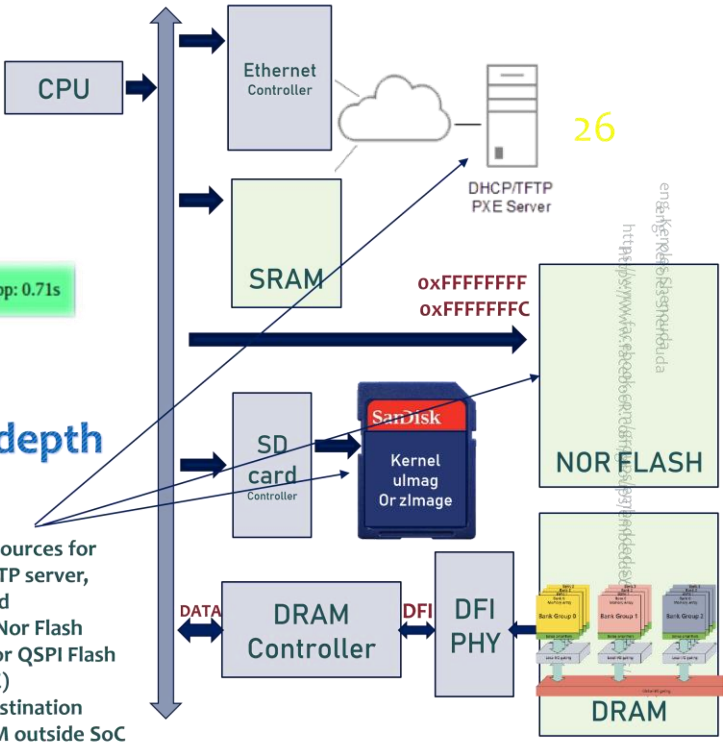
So the Booting Sequence

Average time spent on each stage:



Learn-in-depth

We have 3 sources for loading (TFTP server, SDCARD and Flash even Nor Flash inside SoC or QSPI Flash outside SoC)
And two destination RAM (DRAM outside SoC and SRAM inside SoC)



	Terminology 1	Terminology 2
Power ON		
Boot ROM	Primary program loader	N/A
SPL	Secondary program loader	1st stagebootloader
UBOOT	Secondary program loader	2st stagebootloader
Kernel		
INIT process		
Your APP		