Unit-1

Cortex-M0 Programming model

Overview

- Operation Modes and States
- Registers and Special Registers
- Behaviors of the Application Program
- Memory system overview

Operation Modes and States

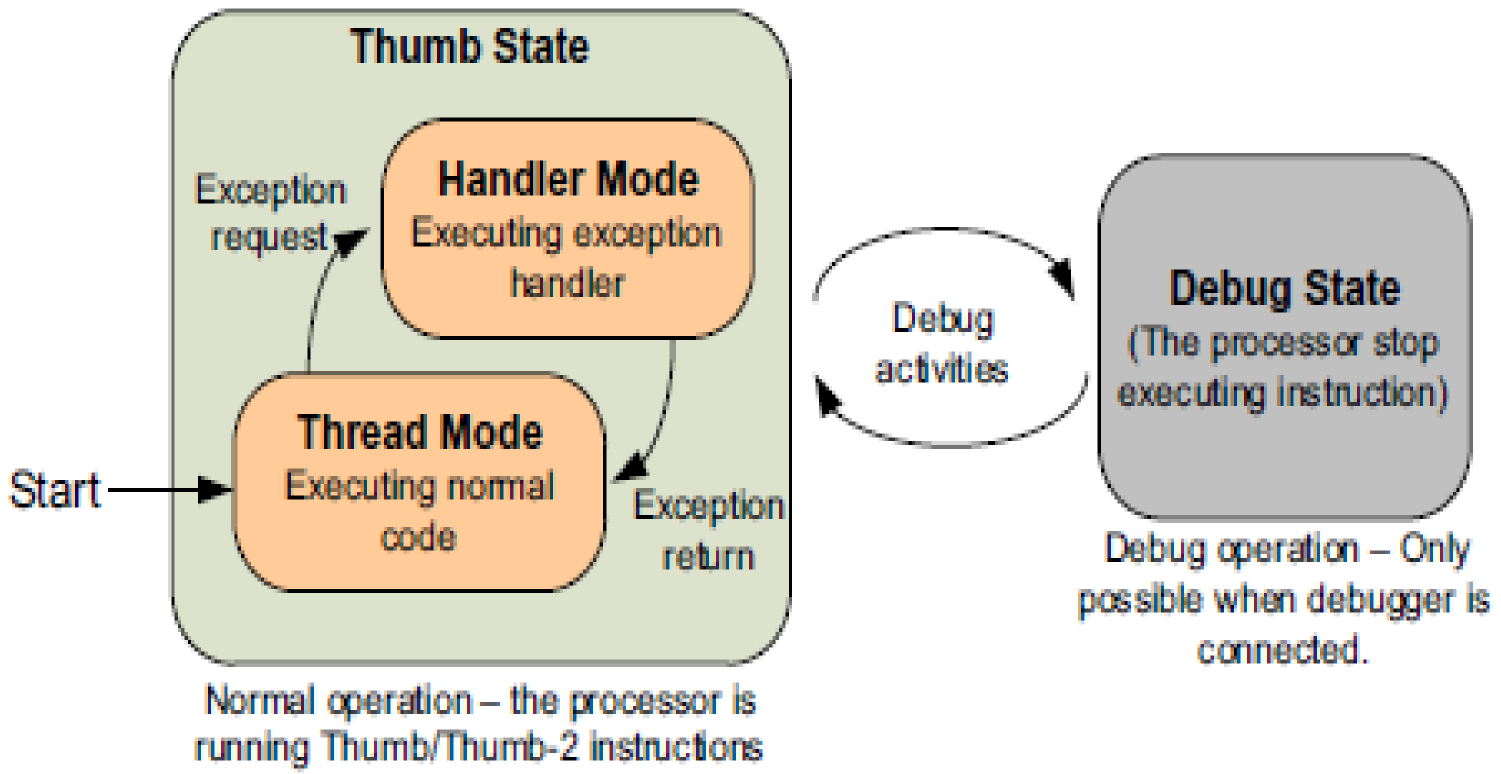
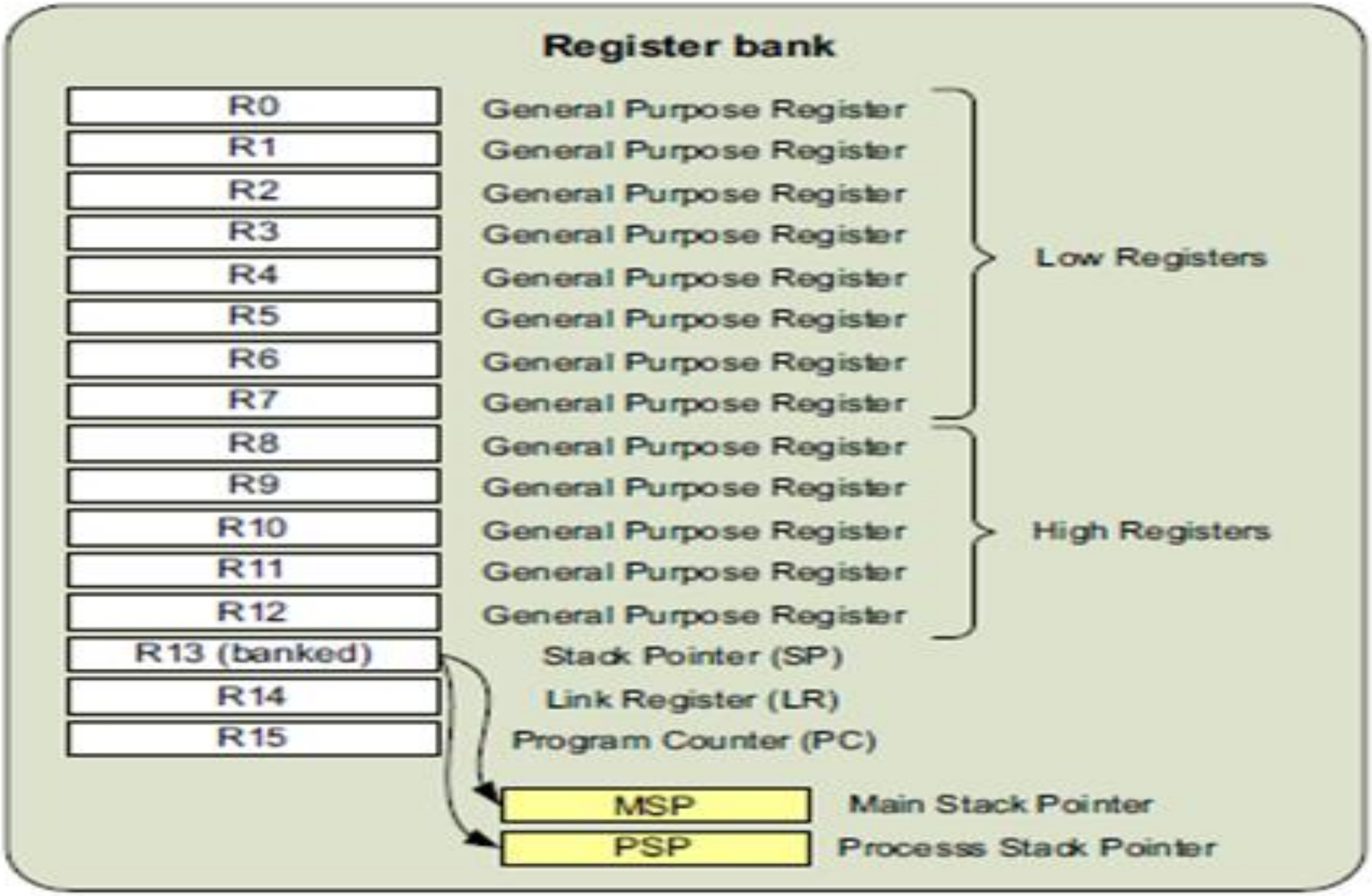
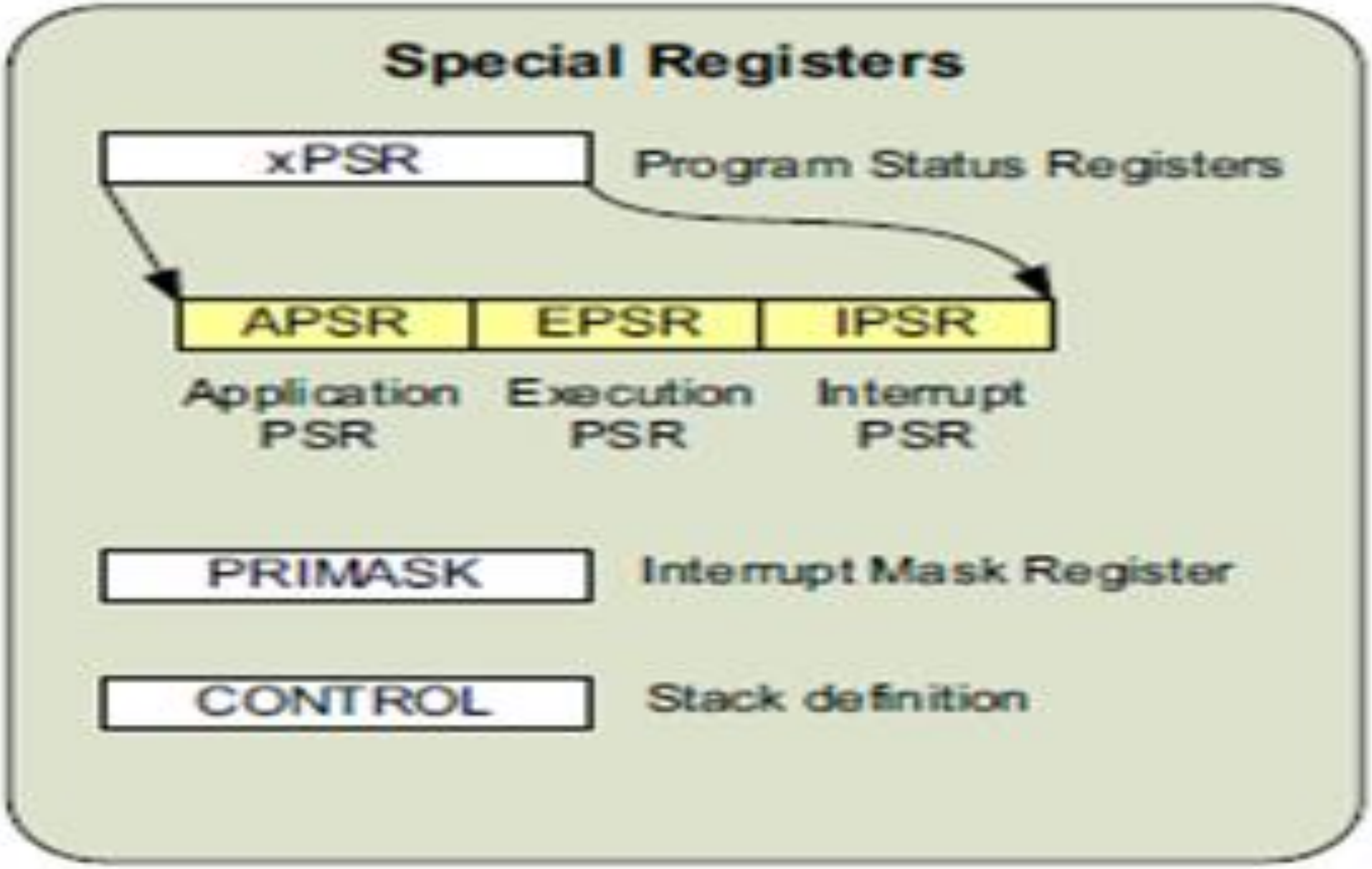

Thumb State
Exception request
Handler Mode
Executing exception handler
Thread Mode
Executing normal code
Start
Exception return
Normal operation – the processor is running Thumb/Thumb-2 instructions
Debug activities
Debug State
(The processor stop executing instruction)
Debug operation – Only possible when debugger is connected.

Figure 3.1:
Processor modes and states in the Cortex-M0 processor.

Registers and Special Registers



Registers and Special Registers



xPSR, combined Program Status Register

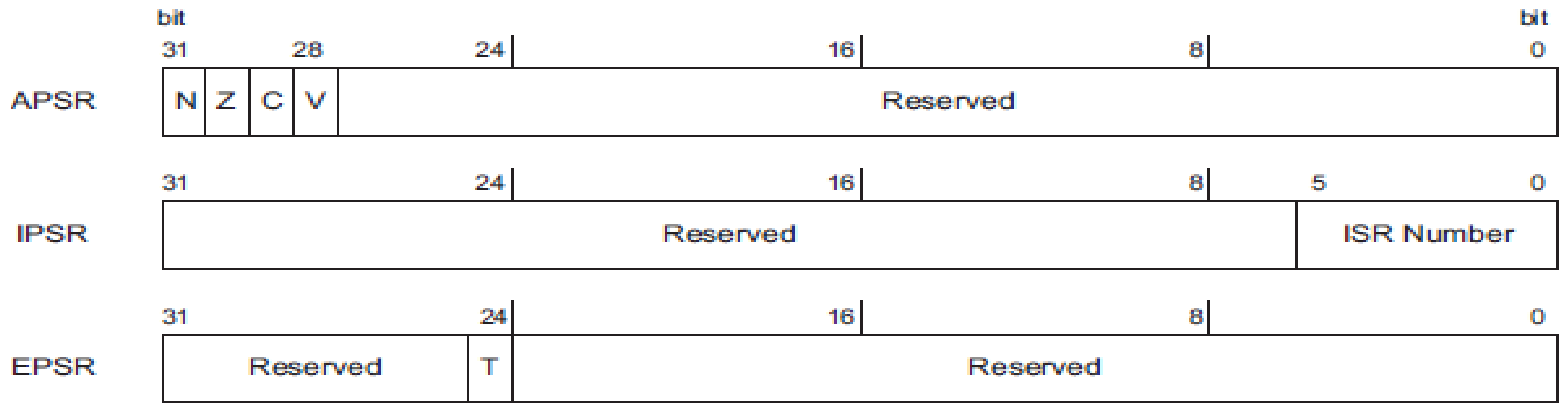


Figure 3.3:
APSR, IPSR, and EPSR.

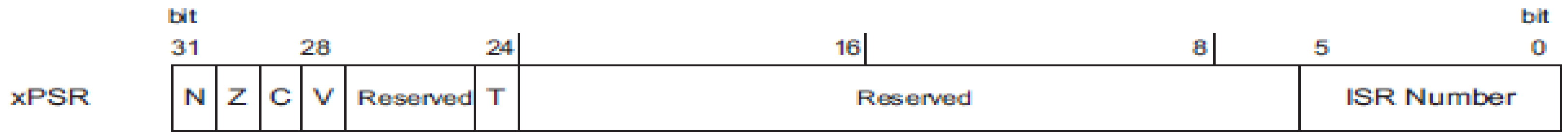
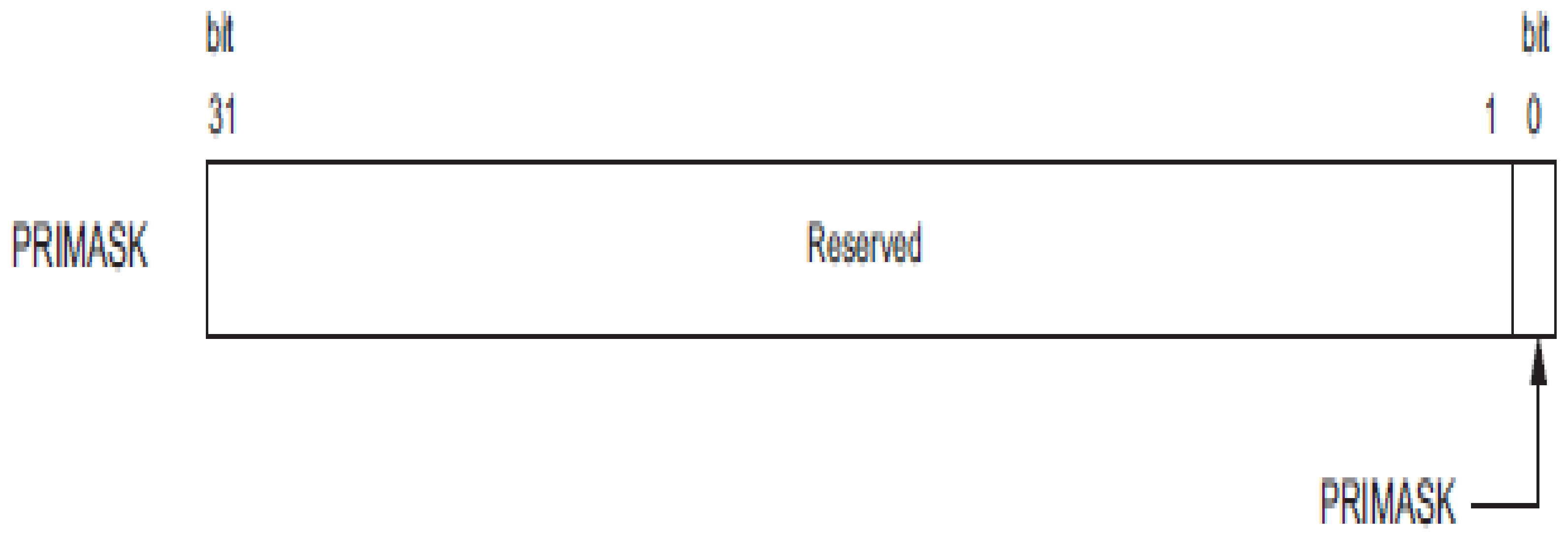
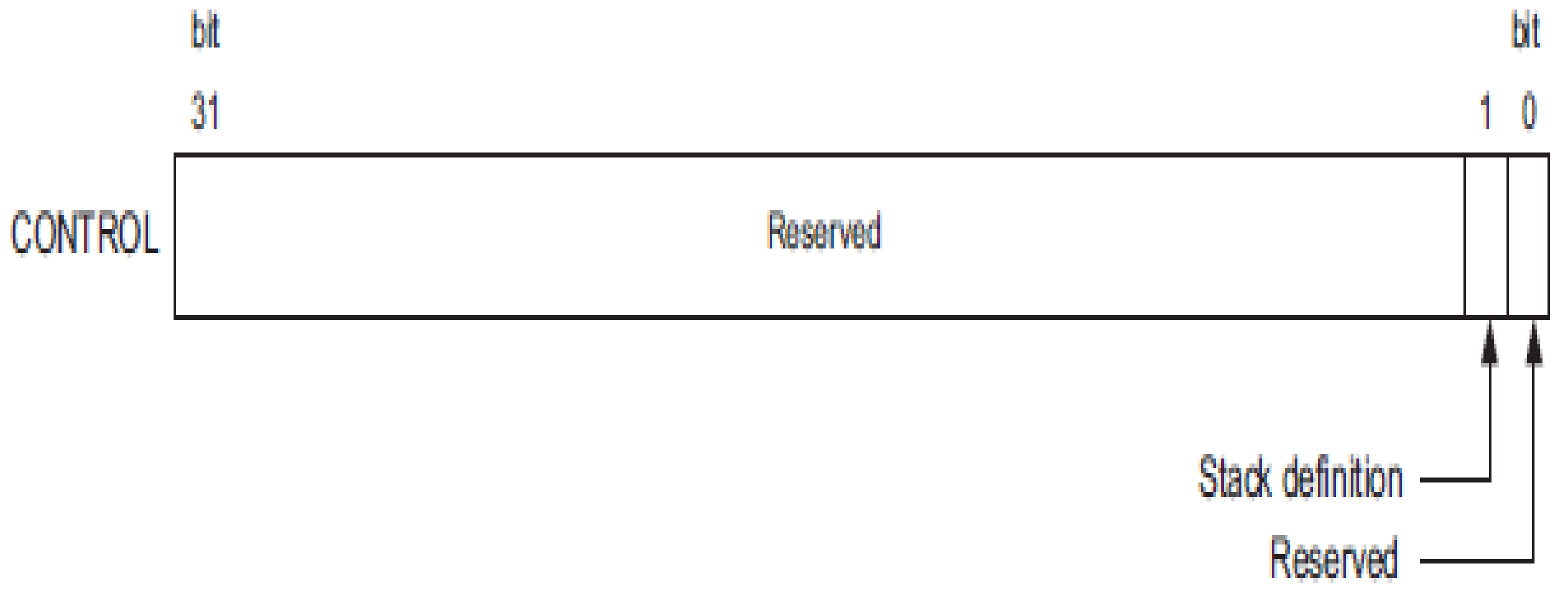


Figure 3.4:
xPSR.

PRIMASK: Interrupt Mask Special Register



CONTROL: Special Register



CONTROL: Special Register

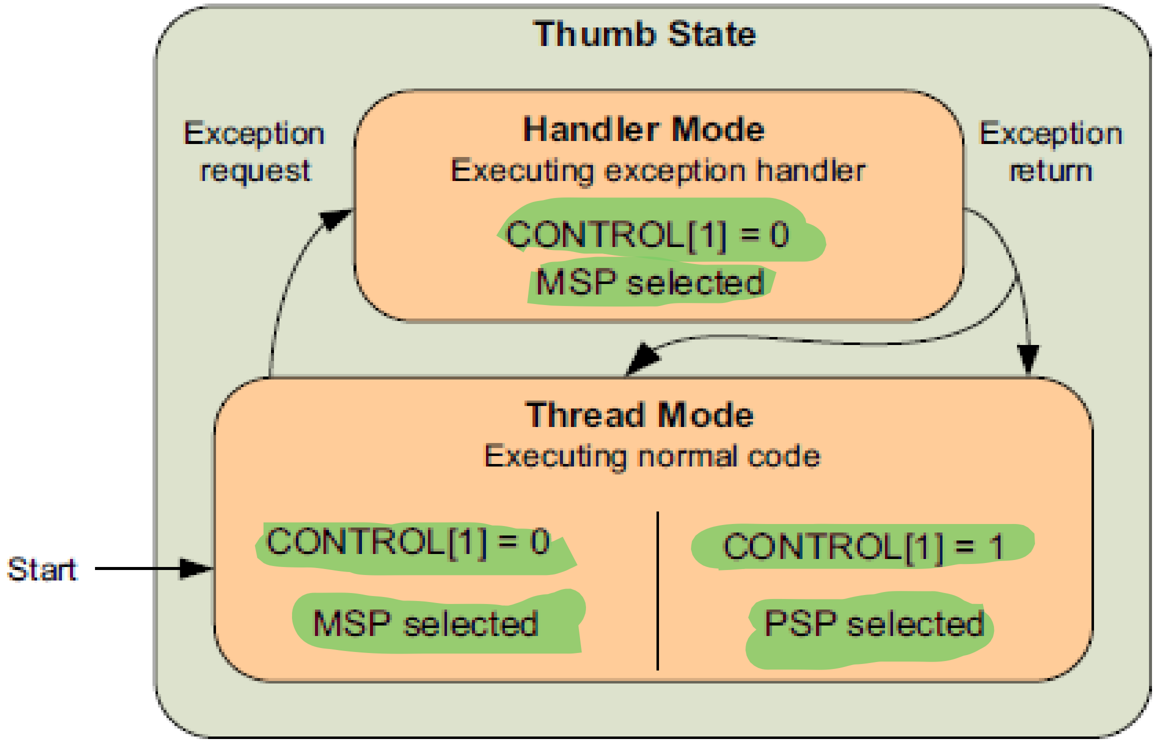
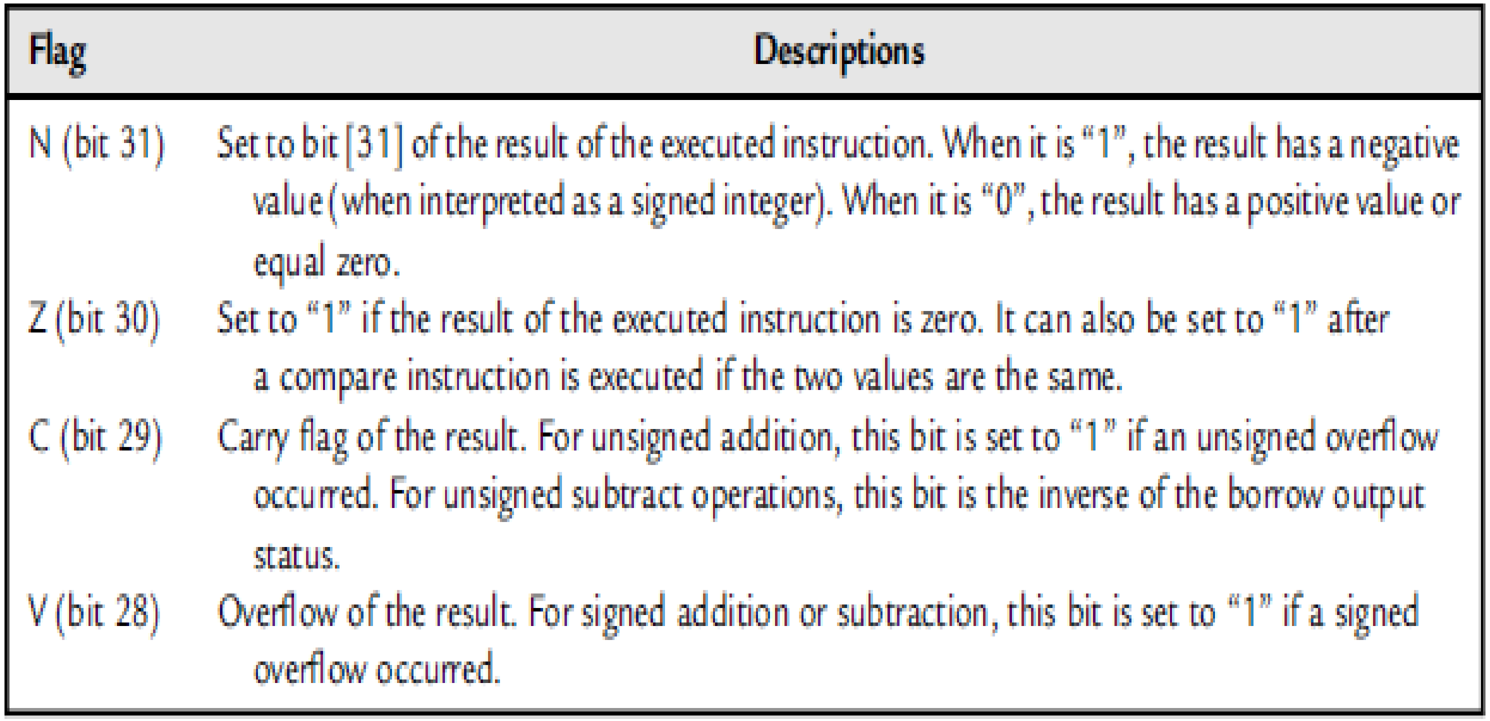


Figure 3.7:
Stack pointer selection.

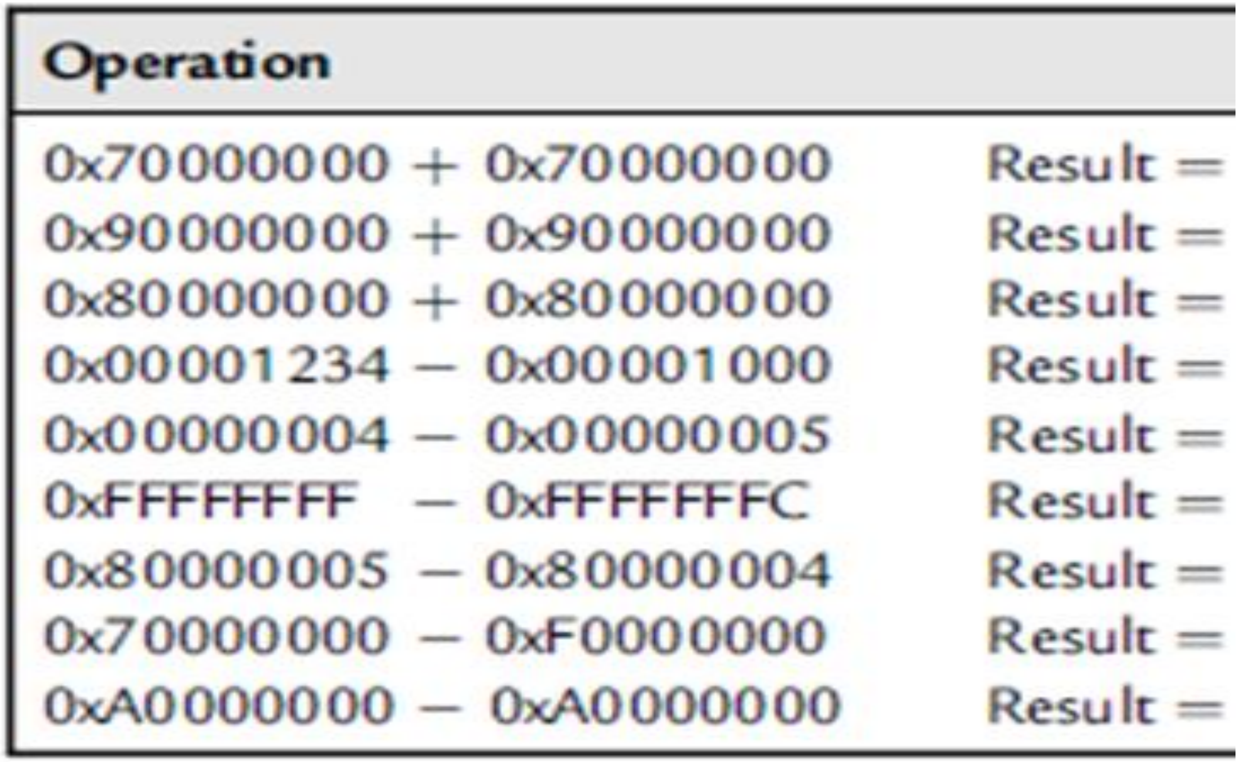
Behaviors of the Application Program Status Register (APSR)

Table 3.1: ALU Flags on the Cortex-M0 Processor



Flag	Descriptions
N (bit 31)	Set to bit [31] of the result of the executed instruction. When it is "1", the result has a negative value (when interpreted as a signed integer). When it is "0", the result has a positive value or equal zero.
Z (bit 30)	Set to "1" if the result of the executed instruction is zero. It can also be set to "1" after a compare instruction is executed if the two values are the same.
C (bit 29)	Carry flag of the result. For unsigned addition, this bit is set to "1" if an unsigned overflow occurred. For unsigned subtract operations, this bit is the inverse of the borrow output status.
V (bit 28)	Overflow of the result. For signed addition or subtraction, this bit is set to "1" if a signed overflow occurred.

Behaviors of the Application Program Status Register (APSR)



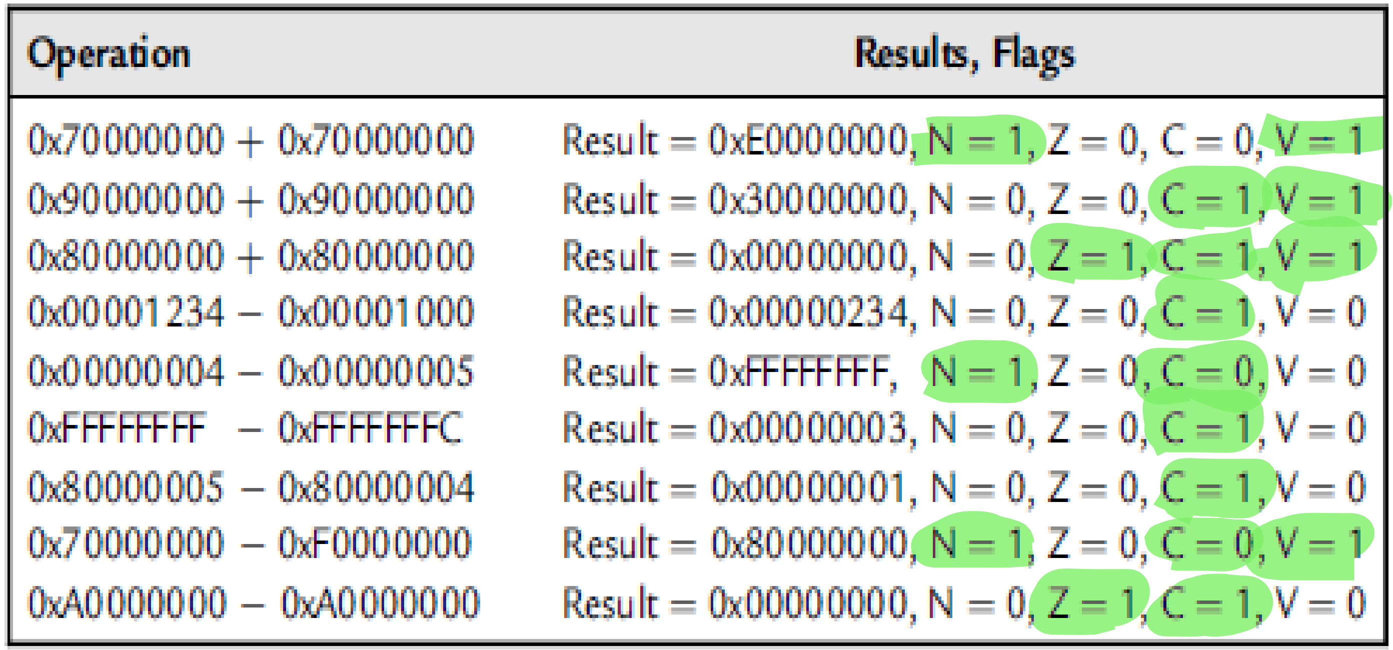
Operation			
0x70000000	+	0x70000000	Result =
0x90000000	+	0x90000000	Result =
0x80000000	+	0x80000000	Result =
0x00001234	−	0x00001000	Result =
0x00000004	−	0x00000005	Result =
0xFFFFFFFF	−	0xFFFFFFFC	Result =
0x80000005	−	0x80000004	Result =
0x70000000	−	0xF0000000	Result =
0xA0000000	−	0xA0000000	Result =

Behaviors of the Application Program Status Register (APSR)

N = Negative (Resultant is -ve or not)
Z = Zero (Result is 0 or not)
C = 1 (If the operation is addition and carrying was done or not done (0)
C = 0 (If operation was subtraction and borrowing was done or not done (1))
V = Overflow (If overflow happened)

A few examples of the ALU flag results are shown in Table 3.2.

Table 3.2: ALU Flags Example



Operation	Results, Flags
0x70000000 + 0x70000000	Result = 0xE0000000, N = 1, Z = 0, C = 0, V = 1
0x90000000 + 0x90000000	Result = 0x30000000, N = 0, Z = 0, C = 1, V = 1
0x80000000 + 0x80000000	Result = 0x00000000, N = 0, Z = 1, C = 1, V = 1
0x00001234 – 0x00001000	Result = 0x00000234, N = 0, Z = 0, C = 1, V = 0
0x00000004 – 0x00000005	Result = 0xFFFFFFFF, N = 1, Z = 0, C = 0, V = 0
0xFFFFFFFF – 0xFFFFFFFC	Result = 0x00000003, N = 0, Z = 0, C = 1, V = 0
0x80000005 – 0x80000004	Result = 0x00000001, N = 0, Z = 0, C = 1, V = 0
0x70000000 – 0xF0000000	Result = 0x80000000, N = 1, Z = 0, C = 0, V = 1
0xA0000000 – 0xA0000000	Result = 0x00000000, N = 0, Z = 1, C = 1, V = 0

Memory System Overview

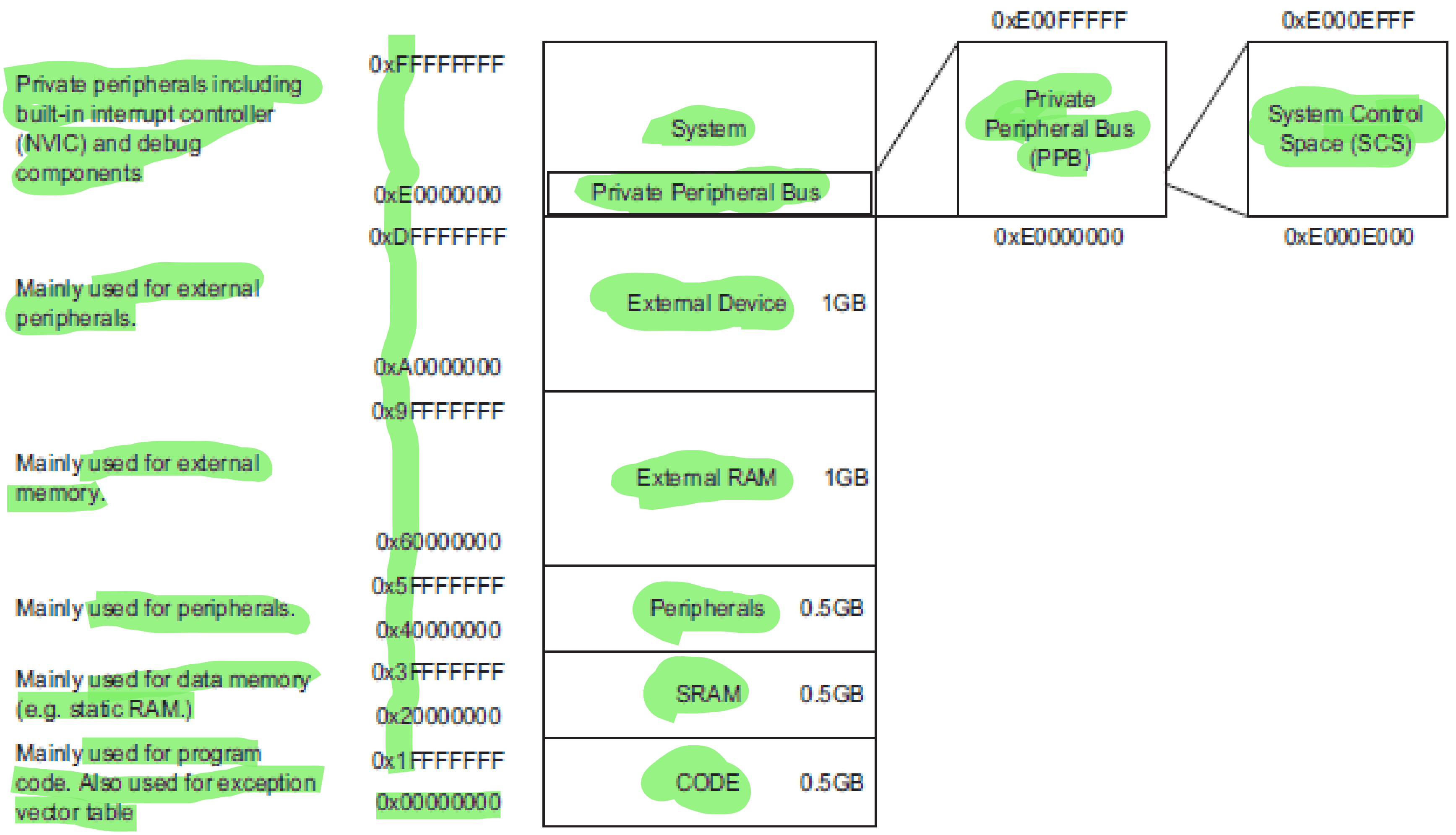

0xE00FFFFF
0xE000EFFF
Private peripherals including built-in interrupt controller (NVIC) and debug components
0xFFFFFFFF
System
Private Peripheral Bus
(PPB)
System Control Space (SCS)
0xE0000000
Private Peripheral Bus
0xE0000000
0xE000E000
0xDFFFFFFF
Mainly used for external peripherals.
External Device
1GB
0xA0000000
0x9FFFFFFF
Mainly used for external memory.
External RAM
1GB
0x60000000
0x5FFFFFFF
Mainly used for peripherals.
Peripherals
0.5GB
0x40000000
0x3FFFFFFF
Mainly used for data memory (e.g. static RAM.)
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Mainly used for program code. Also used for exception vector table
CODE
0.5GB
0x00000000

Figure 3.8:
Memory map.

Stack Memory Operations

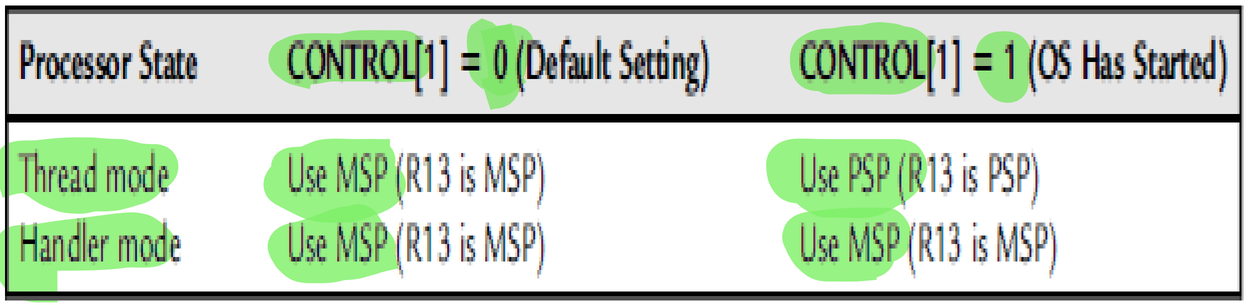
- Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in, last-out buffer.
- One of the essential elements of stack memory operation is a register called the stack pointer.
- The stack pointer is adjusted automatically each time a stack operation is carried out.
- In the Cortex-M0 processor, the stack pointer is register R13 in the register bank.
- Physically there are two stack pointers in the Cortex-M0 processor, but only one of them is used at one time, depending on the current value of the CONTROL register and the state of the processor

Stack Memory Operations

- In common terms, storing data to the stack is called pushing (using the PUSH instruction) and restoring data from the stack is called popping (using the POP instruction).
- Depending on processor architecture, some processors perform storing of new data to stack memory using incremental address indexing and some use decrement address indexing.
- In the Cortex-M0 processor, the stack operation is based on a “full-descending” stack model.
- PUSH and POP are commonly used at the beginning and end of a function or subroutine.

Stack Memory Operations

Table 3.3: Stack Pointer Usage Definition



Processor State	CONTROL[1] = 0 (Default Setting)	CONTROL[1] = 1 (OS Has Started)
Thread mode	Use MSP (R13 is MSP)	Use PSP (R13 is PSP)
Handler mode	Use MSP (R13 is MSP)	Use MSP (R13 is MSP)

Introduction to Cortex-M0 Programming

- Introduction to Cortex-M0 Programming
- Introduction to Embedded System Programming
- What happens when a microcontroller starts?
- Designing Embedded programs
- Input and outputs
- Development Flow

Introduction to Embedded System Programming

- All microcontrollers need program code to enable them to perform their intended tasks.
- Many embedded systems do not have any operating systems and do not have the same user interface as a personal computer.

What Happens When a Microcontroller Starts?

- Most modern microcontrollers have on-chip flash memory to hold the compiled program.
- The flash memory holds the program in binary machine code format, and therefore programs written in C must be compiled before programmed to the flash memory

What Happens When a Microcontroller Starts?

Reset button in MC

Boots the MC

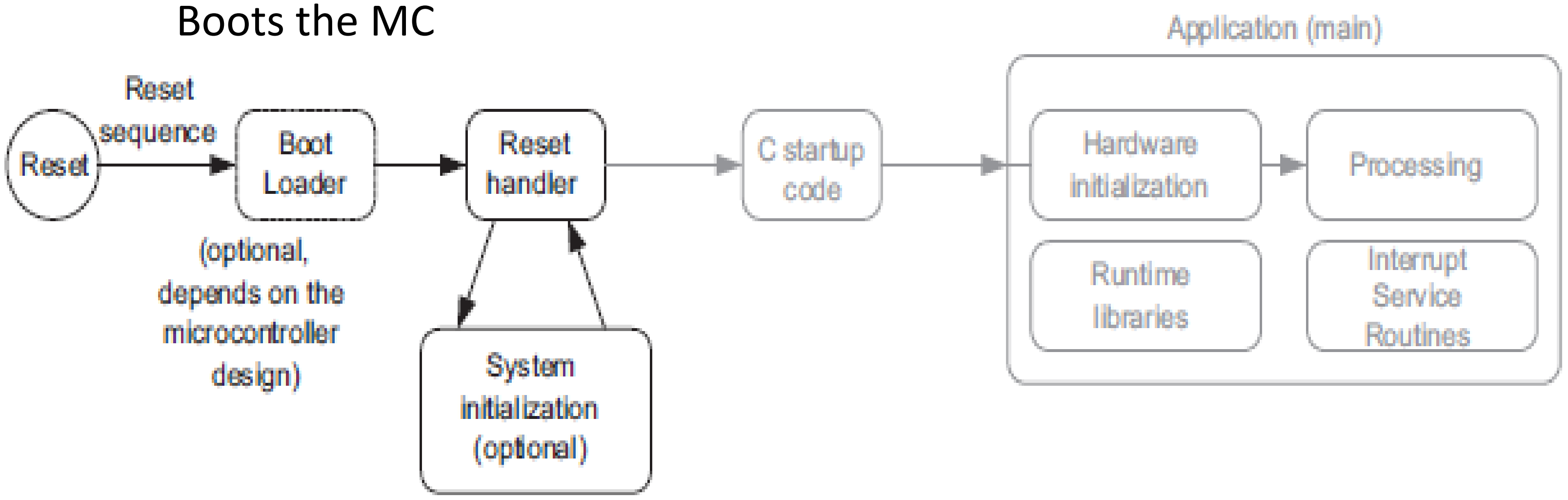


Figure 4.1:
What happens when a microcontroller starts—the Reset handler.

Boot Loader

- Some of these microcontrollers might also have a separate boot ROM, which contains a small boot loader program that is executed when the microcontroller starts, before executing the user program in the flash memory.
- In most cases, only the program code in the flash memory can be changed and the boot loader is fixed.
- After the flash memory (or other types of program memory) is programmed, the program is then accessible by the processor.

Reset Handler

- In the reset sequence, the processor obtains the initial MSP value and reset vector, and then it executes the reset handler.
- All of this required information is usually stored in a program file called startup code.
- The reset handler in the startup code might also perform system initialization (e.g., clock control circuitry and Phase Locked Loop [PLL]), although in some cases system initialization is carried out later when the C program “main()” starts.
- Example startup code can usually be found in the installation of the development suite or from software packages available from the microcontroller vendors.

- For example, if the Keil Microcontroller Development Kit (MDK) is used for development, the project creation wizard can optionally copy a default startup code file into your project that matches the microcontroller you selected

C start up code

- For applications developed in C, the C startup code is executed before entering the main application code.
- The C startup code initializes variables and memory used by the application and they are inserted to the program image by the C development suite.

Application

- The application program often contains the following elements:
- Initialization of hardware (e.g., clock, PLL, peripherals)
- The processing part of the application
- Interrupt service routines

Designing Embedded Programs

- There are many ways to structure the flow of the application processing. Here we will cover a few fundamental concepts

Knocking at each peripheral and verifying

Polling

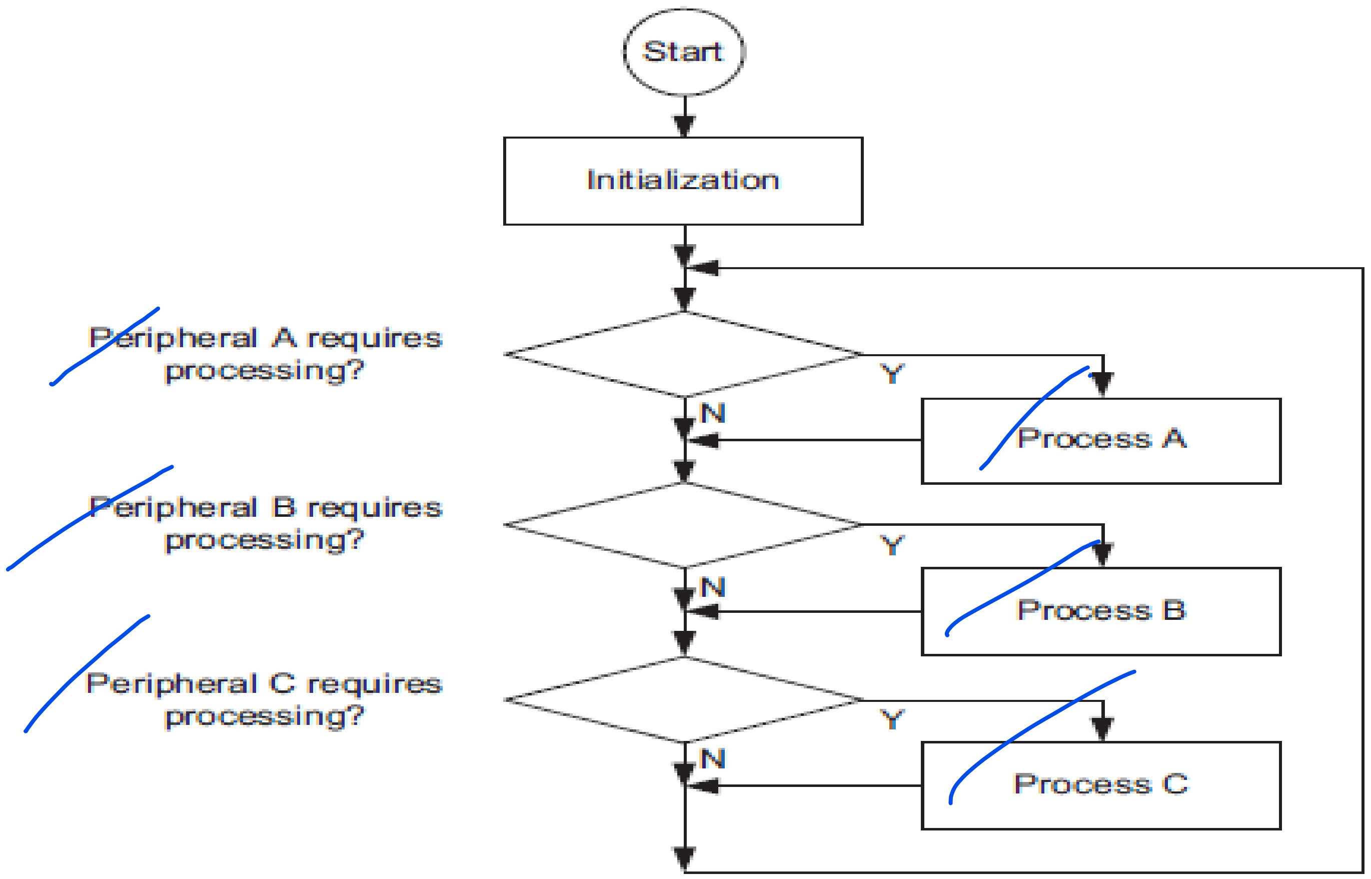
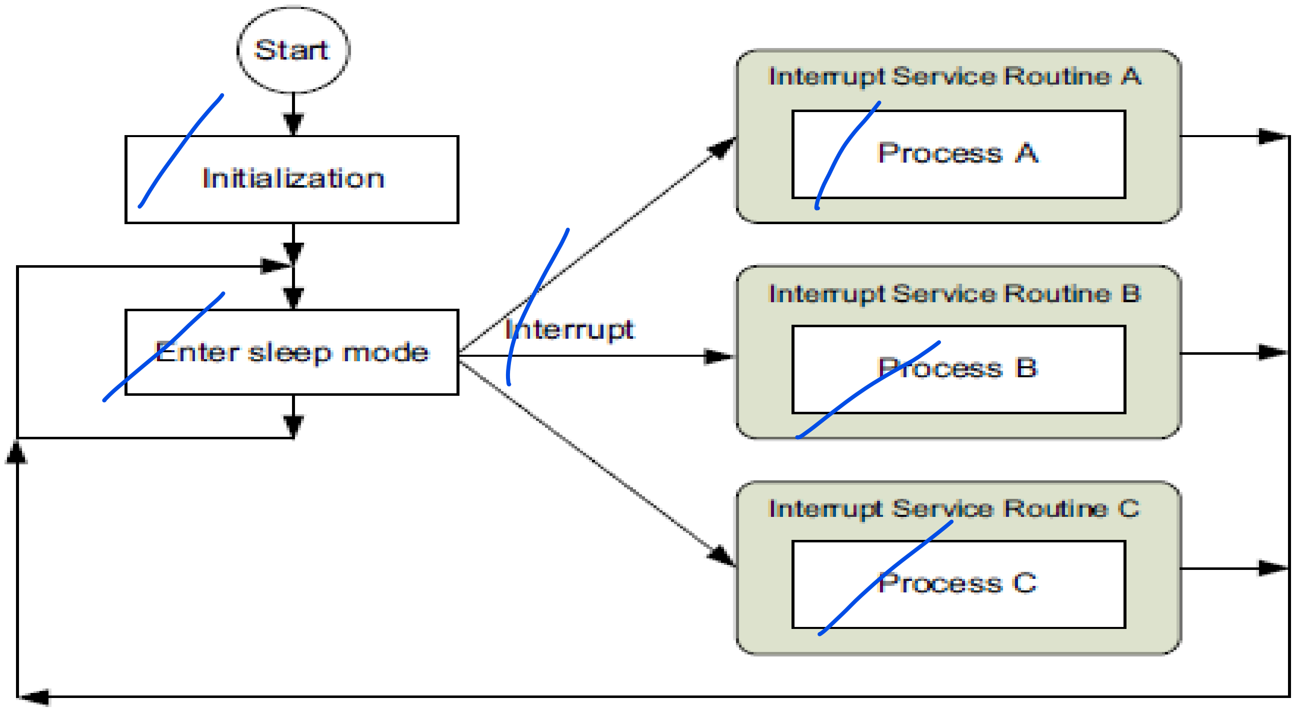

Start
Initialization
Peripheral A requires processing?
Y
N
Process A
Peripheral B requires processing?
Y
N
Process B
Peripheral C requires processing?
Y
N
Process C

Figure 4.4:
Polling method for simple application processing.

Service based on interrupts by an application program

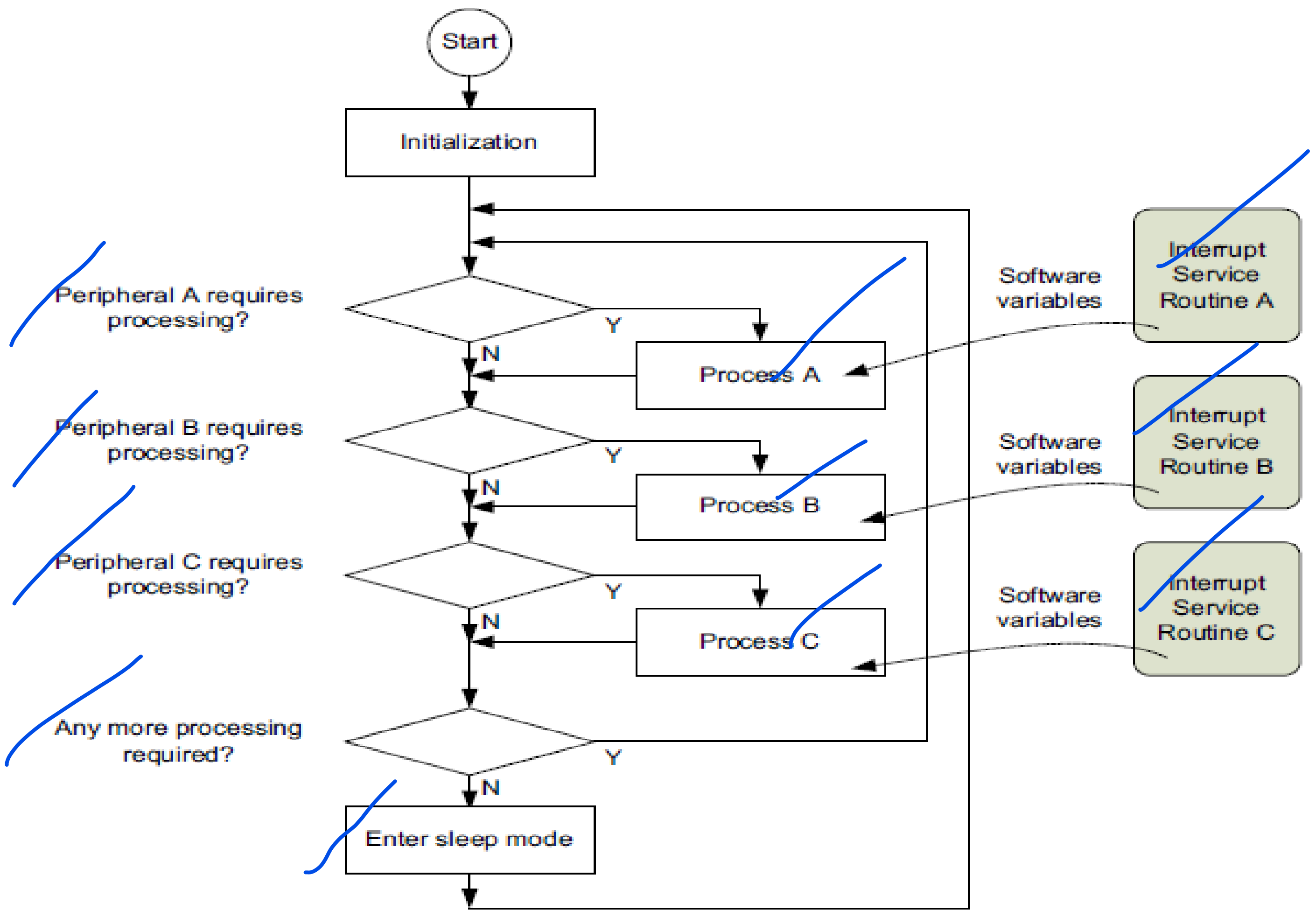
Interrupt Driven



Start
Initialization
Enter sleep mode
Interrupt
Interrupt Service Routine A
Process A
Interrupt Service Routine B
Process B
Interrupt Service Routine C
Process C

Figure 4.5:
An interrupt-driven application.

Combination of Polling and Interrupt Driven



Start
Initialization
Peripheral A requires processing?
Y
N
Process A
Peripheral B requires processing?
Y
N
Process B
Peripheral C requires processing?
Y
N
Process C
Any more processing required?
Y
N
Enter sleep mode
Software variables
Interrupt Service Routine A
Software variables
Interrupt Service Routine B
Software variables
Interrupt Service Routine C

Handling Concurrent Processes

1. Breaking down a long processing task to a sequence of states. Each time the process is accessed, only one state is executed.
2. Using a real-time operating system (RTOS) to manage multiple tasks.

Breaking down a long processing task to a sequence of states. Each time the process is accessed, only one state is executed.

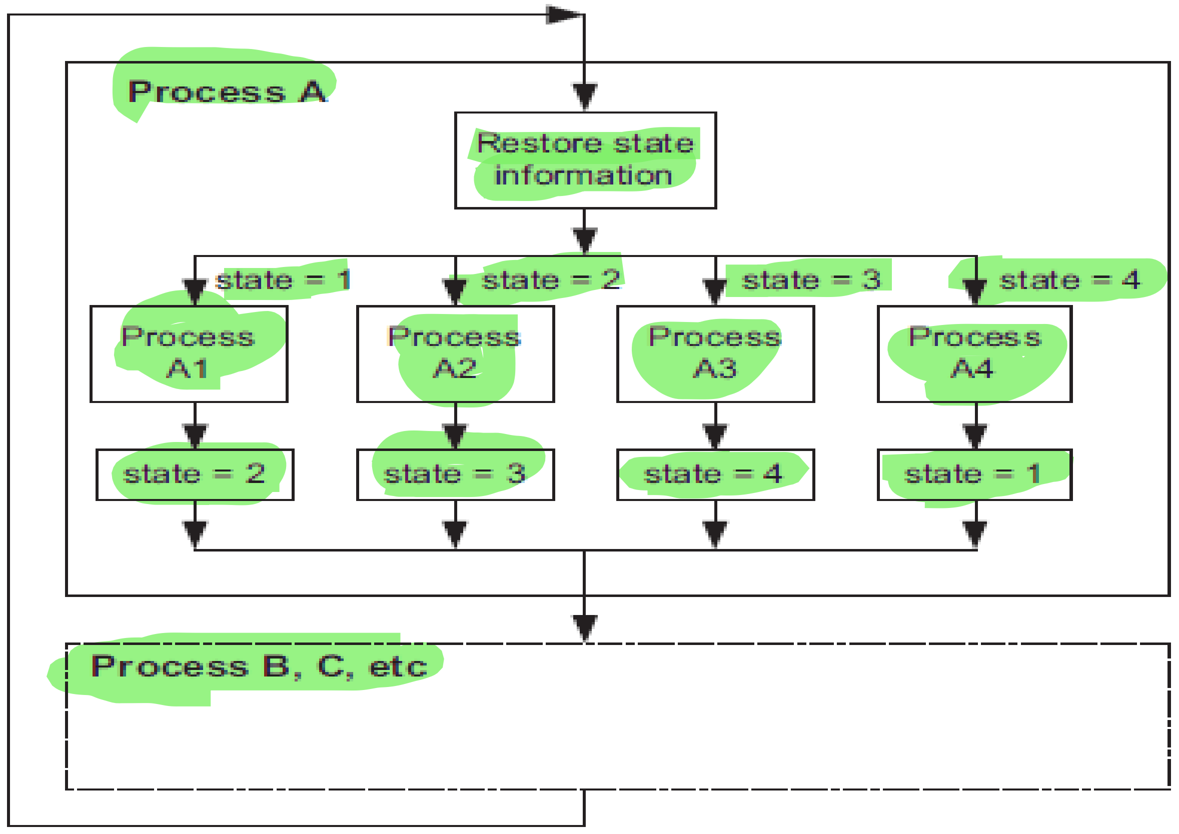


Figure 4.7:
Partitioning a process into multiple parts in the application loop.

Using a real-time operating system (RTOS) to manage multiple tasks.

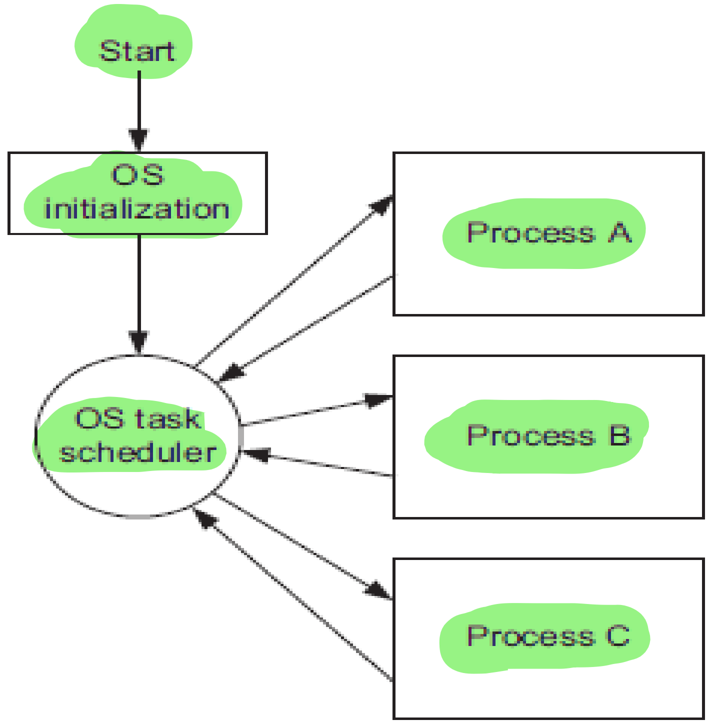


Figure 4.8:
Using an RTOS to handle multiple concurrent application processes.

Inputs and Outputs

- Typically, the initialization process for peripherals may consist of the following steps:
- **1. Programming the clock control circuitry** to enable the clock signal to the peripheral and the corresponding I/O pins if necessary. In many low-power microcontrollers, the clock signals reaching different parts of the chip can be turned on or off individually to save power. By default, most of the clock signals are usually turned off and need to be enabled before the peripherals are programmed. In some cases you also need to enable the clock signals for the peripherals bus system.
- **2. Programming of I/O configurations**. Most microcontrollers multiplex their I/O pins for multiple uses. For a peripheral interface to work correctly, the I/O pin assignment might need to be programmed. In addition, some microcontrollers also offer configurable electrical characteristics for the I/O pins. This can result in additional steps in I/O configurations

- **3. Peripheral configuration**. Most interface peripherals contain a number of programmable registers to control their operations, and therefore a programming sequence is usually needed to allow the peripheral to work correctly.
- **4. Interrupt configuration**. If a peripheral operation requires interrupt processing, additional steps are required for the interrupt controller (e.g., the NVIC in the Cortex-M0).

Development Flow

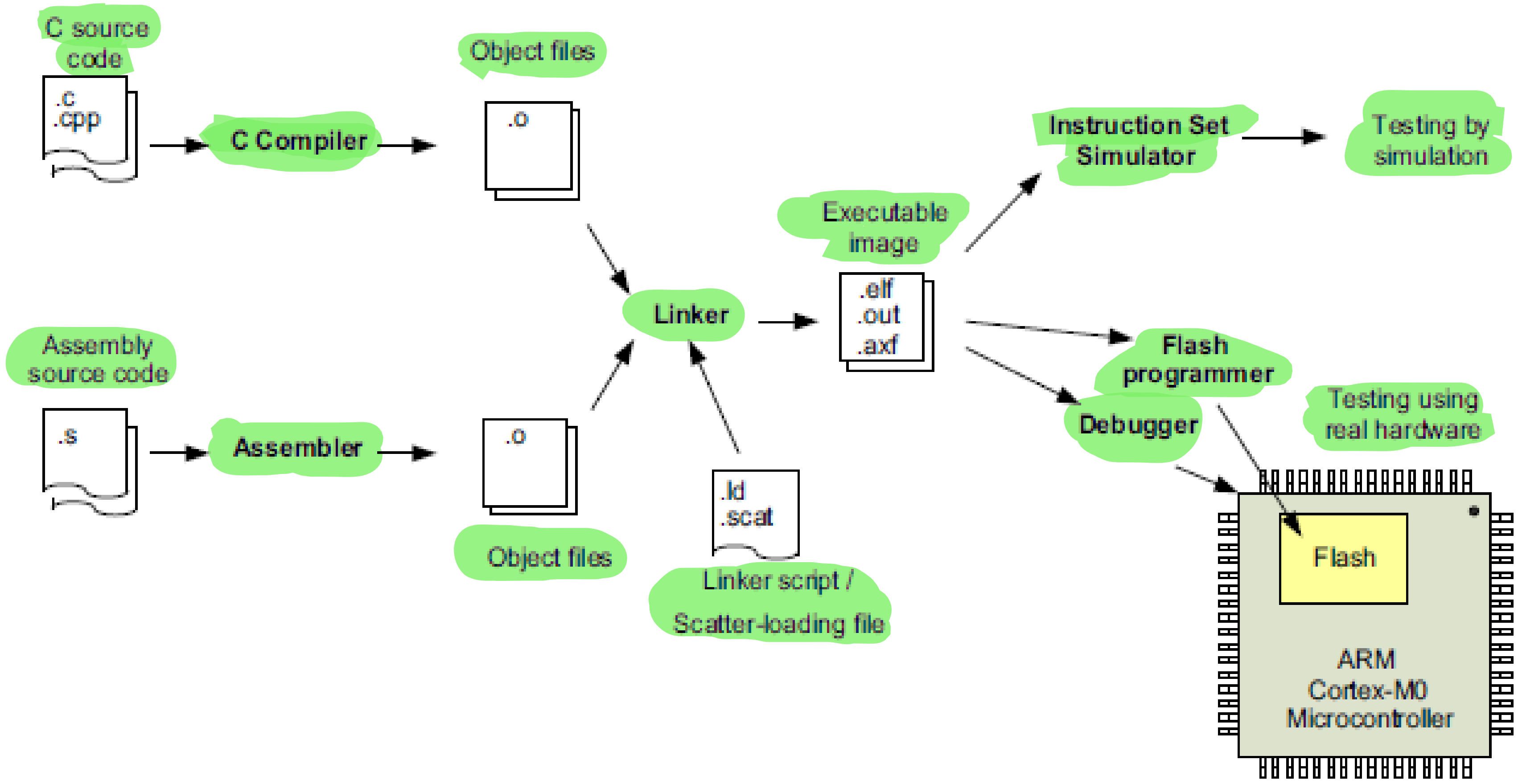


Figure 4.10:
Typical program-generation flow.

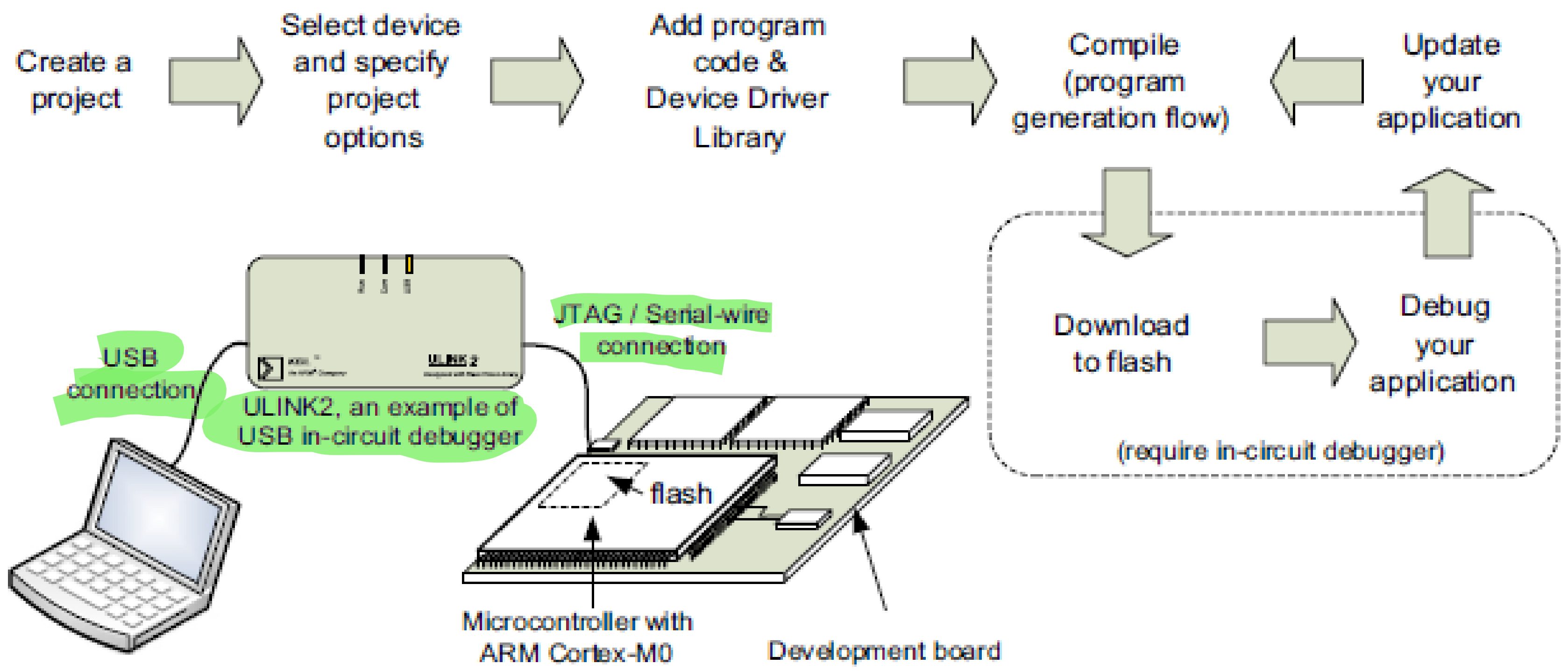

Create a project
Select device and specify project options
Add program code & Device Driver Library
Compile (program generation flow)
Update your application
Download to flash
Debug your application
(require in-circuit debugger)
USB connection
ULINK2, an example of USB in-circuit debugger
JTAG / Serial-wire connection
flash
Microcontroller with ARM Cortex-M0
Development board

Figure 4.11:
An example of development flow.

Table 4.1 : Comparison between C Programming and Assembly Language Programming

Language	Pros and Cons
C	**Pros**
	Easy to learn
	Portable
	Easy handling of complex data structures
	Cons
	Limited/no direct access to core register and stack
	No direct control over instruction sequence generation
	No direct control over stack usage
Assembly	**Pros**
	Allows direct control to each instruction step and all memory operations
	Allows direct access to instructions that cannot be generated with C
	Cons
	Take longer time to learn
	Difficult to manage data structure
	Less portable (syntax of assembly language in different tool chains can be different)

What Is in a Program Image?

A program image for the Cortex-M0 microcontroller often contains the following components:

- Vector table
- C startup routine
- Program code (application code and data)
- C library code (program codes for C library functions, inserted at link time)

Vector Table

The vector table can be programmed in either C language or assembly language. The exact details of the vector table code are tool chain dependent because vector table entries require symbols created by the compiler and linker. For example, the initial stack pointer value is linked to stack region address symbols generated by the linker, and the reset vector is linked to C startup code address symbols, which are compiler dependent

C Startup Code

- The C startup code is used to set up data memory such as global data variables. It also zero initializes part of the data memory for variables that are uninitialized at load time. For applications that use C functions like malloc(), the C startup code also needs to initialize the data variables controlling the heap memory. After this initialization, the C startup code branches to the beginning of the main() program.
- The C startup code is inserted by the compiler/linker automatically and is tool chain specific; it might not be present if you are writing a program purely in assembly. For ARM compilers, the C startup code is labeled as "__main," whereas the startup code generated by
- GNU C compilers is normally labeled as "_start."

Program Code

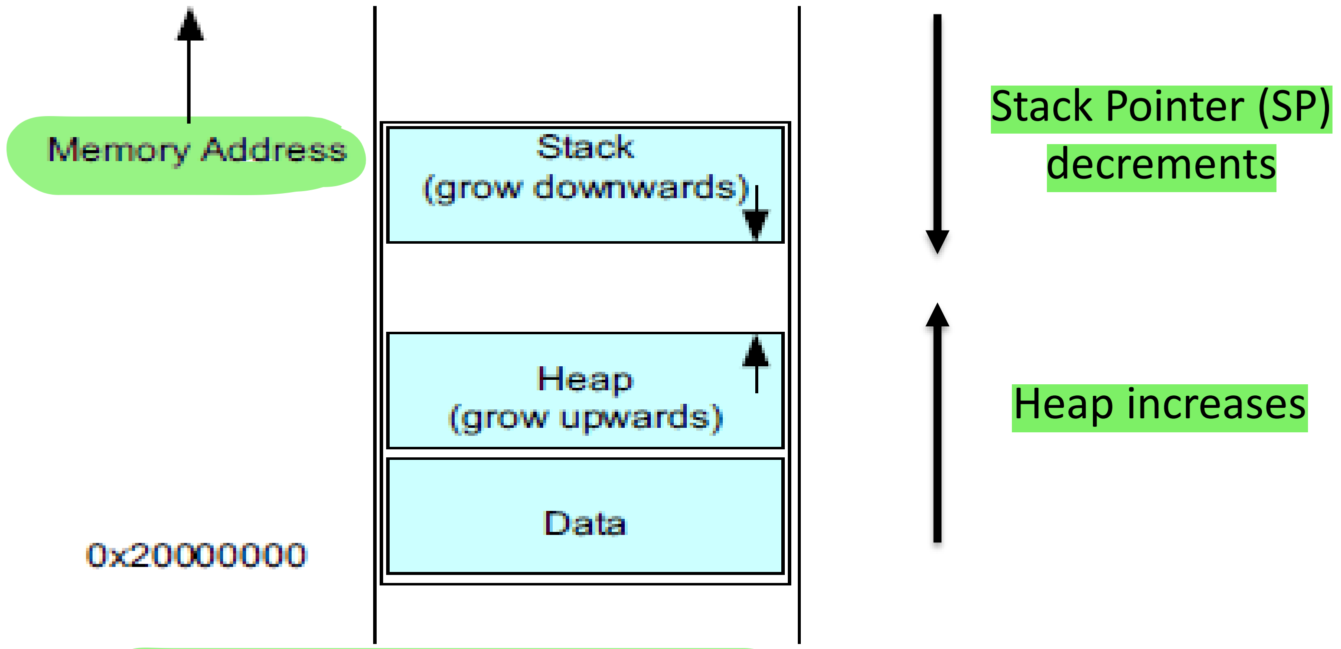
- The instructions generated from your application program code carry out the tasks you specify. Apart from the instruction sequence, there are also various types of data:
- Initial values of variables. Local variables in functions or subroutines need to be initialized,and these initial values are set up during program execution.
- Constants in program code. Constant data are used in application codes in many ways: data values, addresses of peripheral registers, constant strings, and so on. These data are sometimes grouped together within the program images as a number of data blocks called literal pools.
- Some applications can also contain additional constant data like lookup tables and graphics image data (e.g., bit map) that are merged into the program images.

C Library Code

C library code is injected in to the program image by the linker when certain C/C++ functions are used. In addition, C library code can also be included because of data processing tasks such as floating point operations and divide. The Cortex-M0 does not have a divide instruction, and this function typically needs to be carried out by a C library divide function.

- Some development tools offer various versions of C libraries for different purposes. For example, in Keil MDK or ARM RVDS there is an option to use a special version of C library called Microlib. The Microlib is targeted for microcontrollers and is very small, but it does not offer all features of the standard C library

Data in RAM



Example RAM usage in systems without OS

Figure 4.14:
Example of RAM usage in single task systems (without OS).

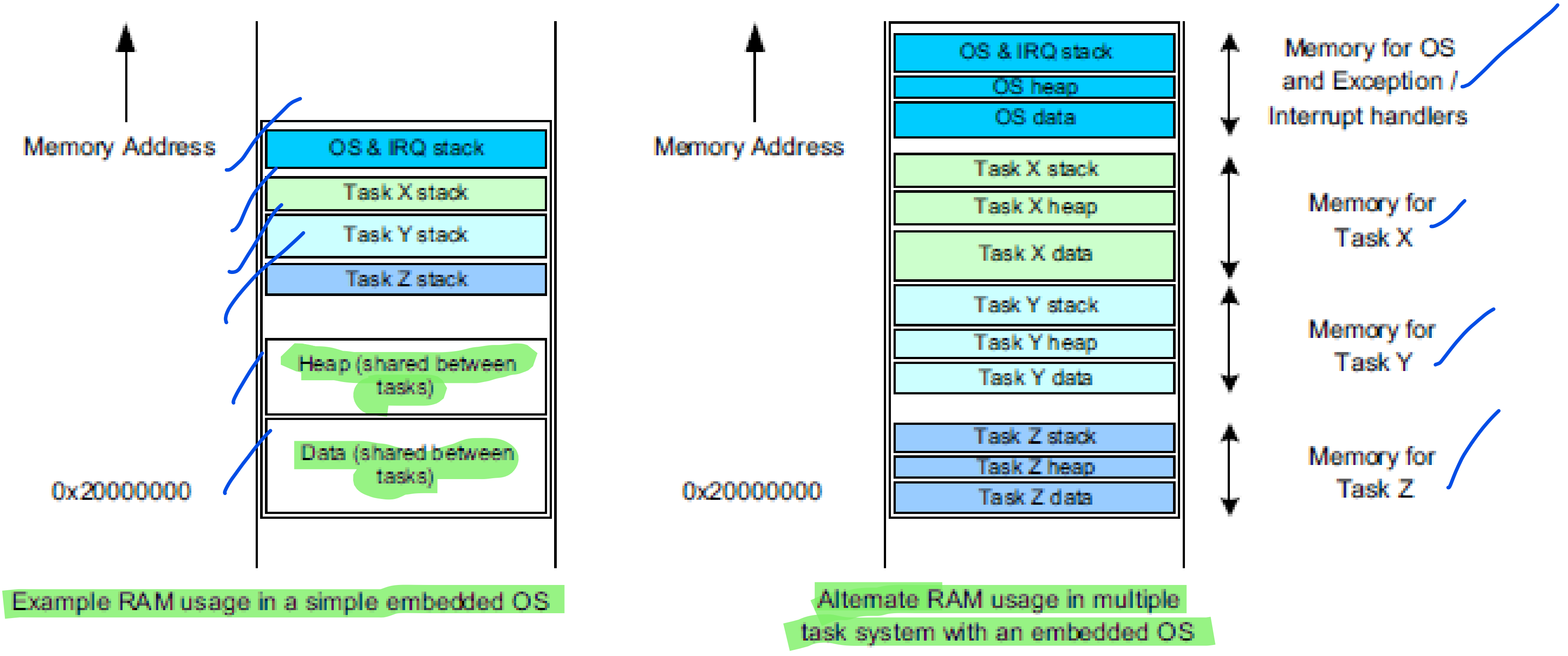
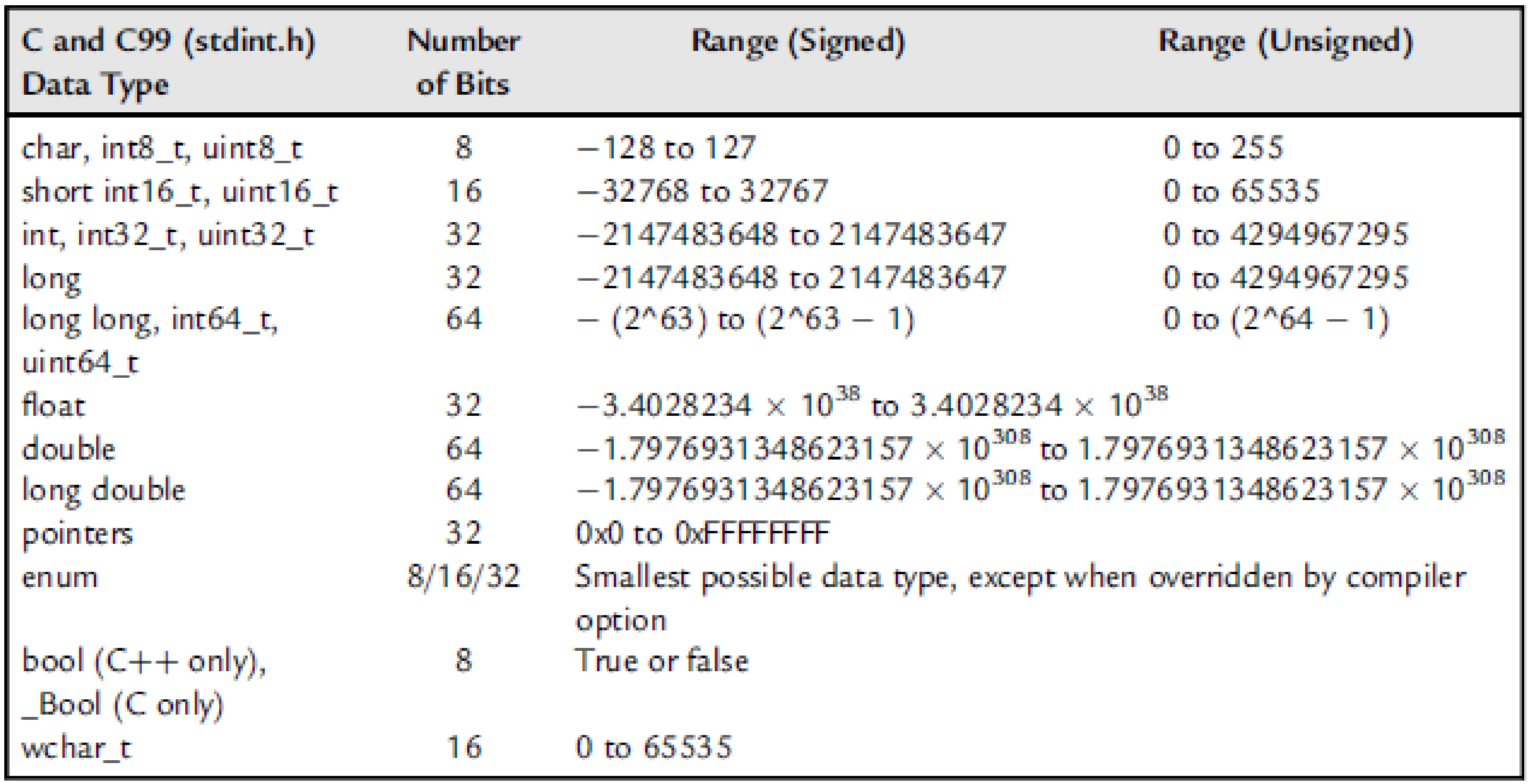


Figure 4.15:
Example of RAM usage in multiple task systems (with an OS).

C Programming: Data Types

Table 4.2: Size of Data Types in Cortex-M Processors



C and C99 (stdint.h) Data Type	Number of Bits	Range (Signed)	Range (Unsigned)
char, int8_t, uint8_t	8	−128 to 127	0 to 255
short int16_t, uint16_t	16	−32768 to 32767	0 to 65535
int, int32_t, uint32_t	32	−2147483648 to 2147483647	0 to 4294967295
long	32	−2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	64	− (2^63) to (2^63 − 1)	0 to (2^64 − 1)
float	32	$-3.4028234 \times 10^{38}$ to 3.4028234×10^{38}	
double	64	$-1.797693134862315 7 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
long double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
pointers	32	0x0 to 0xFFFFFFFF	
enum	8/16/32	Smallest possible data type, except when overridden by compiler option	
bool (C++ only), _Bool (C only)	8	True or false	
wchar_t	16	0 to 65535	

Cortex Microcontroller Software Interface Standard (CMSIS)

- As the complexity of embedded systems increase, the compatibility and reusability of software code becomes more important. Having reusable software often reduces development time for subsequent projects and hence speeds up time to market, and software compatibility helps the use of third-party software components. For example, an embedded system project might involve the following software components:
- Software from in-house software developers
- Software reused from other projects
- Device driver libraries from microcontroller vendors
- Embedded OS
- Other third-party software products like a communication protocol stack and codec (compressor/decompressor)

CMSIS provides standardized access functions for processor features

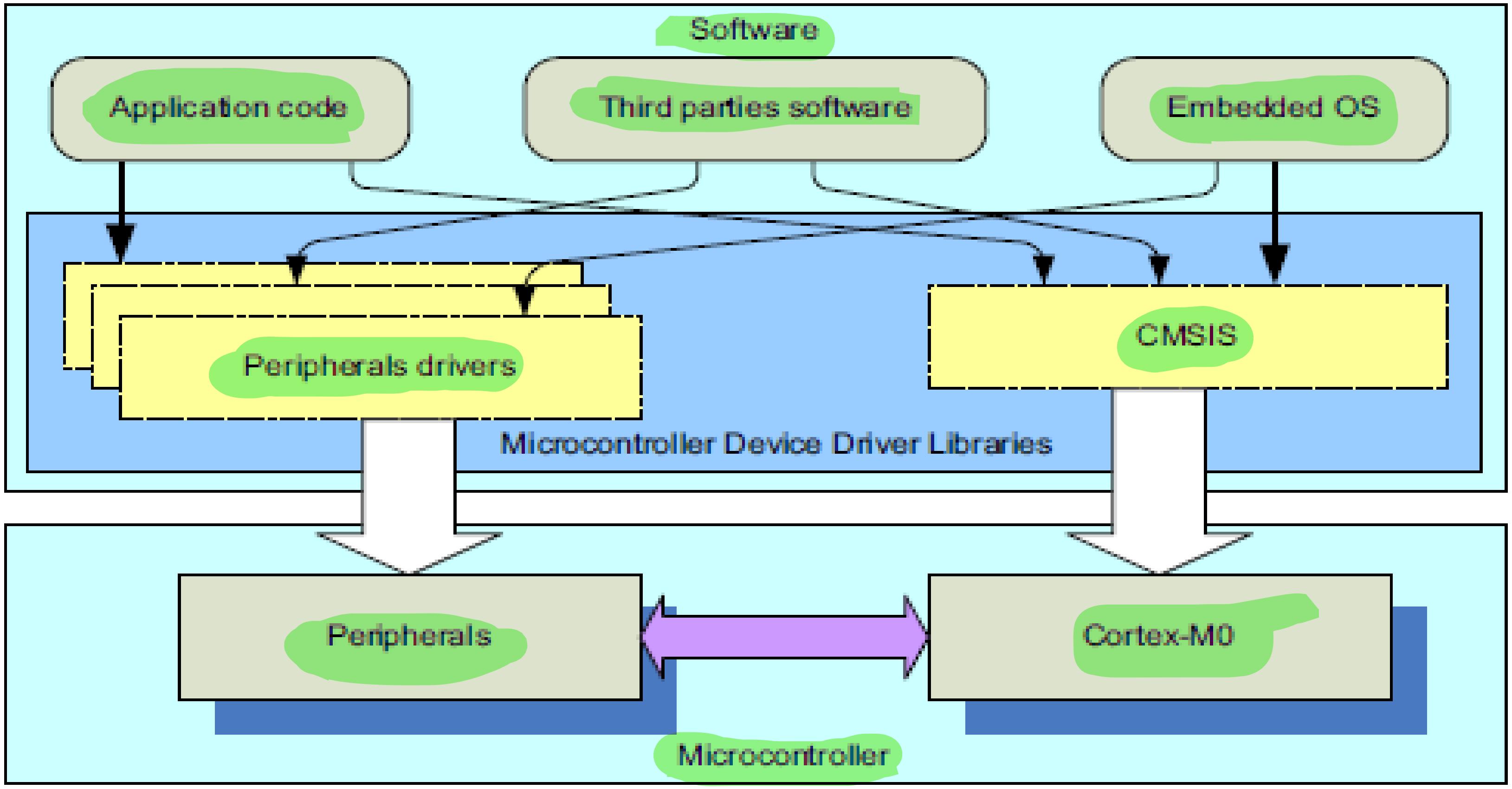


Figure 4.16:
CMSIS provides standardized access functions for processor features.

What is standardized in CMSIS?

- Standardized access functions for accessing NVIC, System Control Block (SCB), and System Tick timer (SysTick) such as interrupt control and SysTick initialization..
- Standardized register definitions for NVIC, SCB, and SysTick registers. For best software portability, we should use the standardized access functions. However, in some cases we need to directly access the registers in NVIC, SCB, or the SysTick. In such cases, the standardized register definitions help the software to be more portable.

- Standardized functions for accessing special instructions in Cortex-M microcontrollers. Some instructions on the Cortex-M microcontroller cannot be generated by normal C code. If they are needed, they can be generated by these functions provided in CMSIS. Otherwise, users will have to use intrinsic functions provided by the C compiler or embedded/inline assembly language, which are tool chain specific and less portable.
- Standardized names for system exceptions handlers. An embedded OS often requires system exceptions. By having standardized system exception handler names, supporting different device driver libraries in an embedded OS is much easier.
- Standardized name for the system initialization function. The common system initialization function “void SystemInit(void)” makes it easier for software developers to set up their system with minimum effort.

- Standardize variable for clock speed information. A standardized software variable called “SystemFreq” (CMSIS v1.00 to v1.20) or “SystemCoreClock” (CMSIS v1.30 or newer). This is used to determine the processor clock frequency.

Organization of CMSIS

The CMSIS is divided into multiple layers:

- **Core Peripheral Access Layer**

Name definitions, address definitions, and helper functions to access core registers and core peripherals like the NVIC, SCB, and SysTick

- **Middleware Access Layer (work in progress)**

Common method to access peripherals for typical embedded systems

Targeted at communication interfaces including UART, Ethernet, and SPI

Allows embedded software to be used on any Cortex microcontrollers that support the required communication interface

- **Device Peripheral Access Layer (MCU specific)**
- Register name definitions, address definitions, and device driver code to access peripherals
- Access Functions for Peripherals (MCU specific)
- Optional helper functions for peripherals

Organization of CMSIS

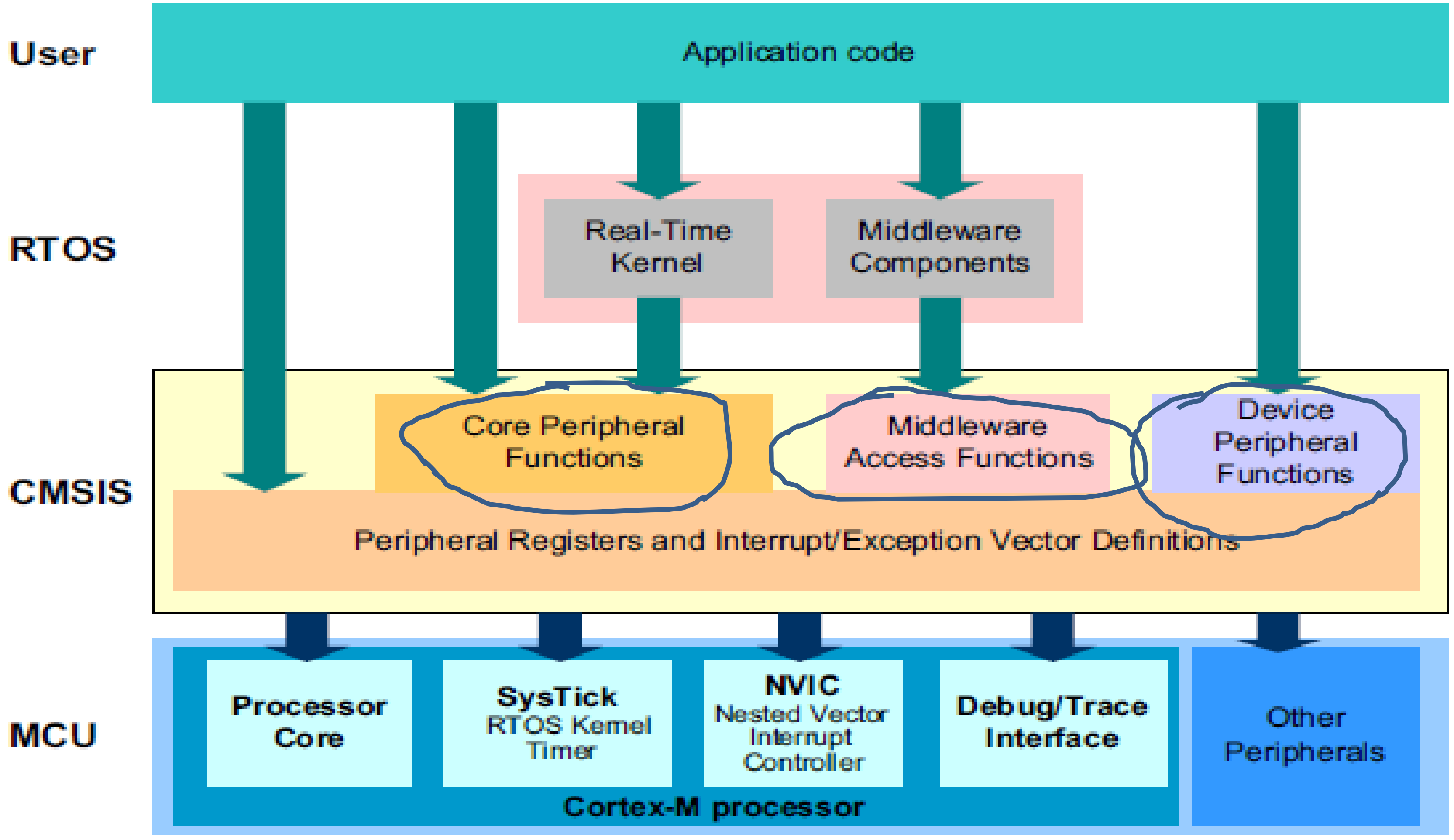

User
Application code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick RTOS Kernel Timer
NVIC Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals
Cortex-M processor

Figure 4.17:
CMSIS structure.

Using CMSIS

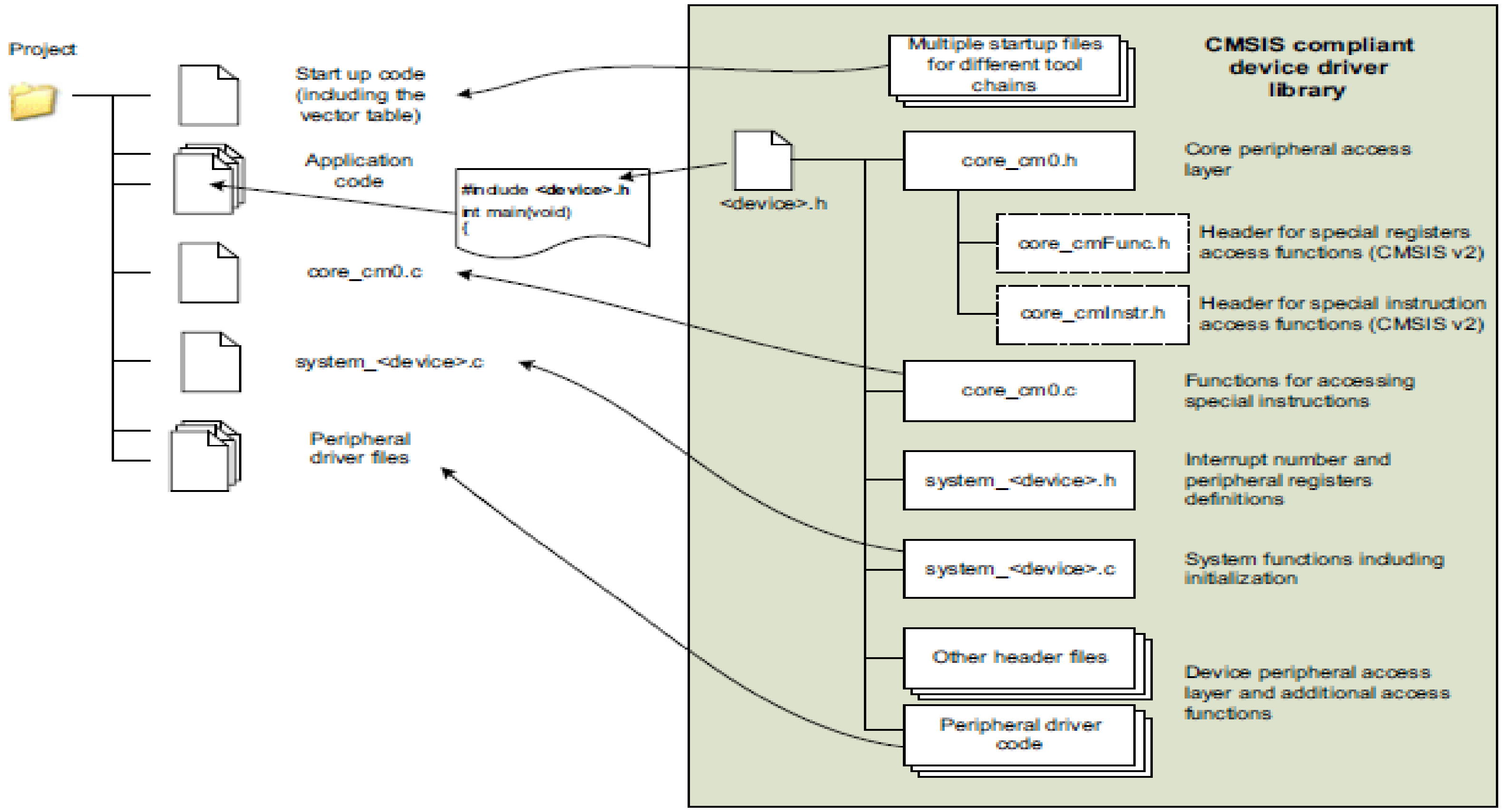

Project
Start up code (including the vector table)
Application code
core_cm0.c
system_<device>.c
Peripheral driver files
#include <device>.h
int main(void)
{
<device>.h
CMSIS compliant device driver library
Multiple startup files for different tool chains
core_cm0.h
Core peripheral access layer
core_cmFunc.h
Header for special registers access functions (CMSIS v2)
core_cmInstr.h
Header for special instruction access functions (CMSIS v2)
core_cm0.c
Functions for accessing special instructions
system_<device>.h
Interrupt number and peripheral registers definitions
system_<device>.c
System functions including initialization
Other header files
Peripheral driver code
Device peripheral access layer and additional access functions

Figure 4.18:
Using CMSIS in a project.