MSE | MASTER OF SCIENCE IN ENGINEERING

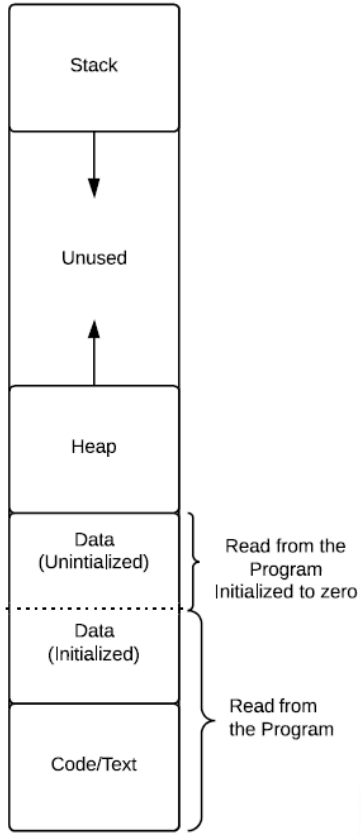
TSM_AdvEmbSof

Memory (Profiling, Optimisation)

Serge Ayer | 23.11.2023 | Cours MSE

Tasks need memory

- Computer memory is used for storing
  - program code
  - application data
  - modified/computed data
- Memory is usually organized in different sections
  - Code or Text
    - Binary instructions to be executed
    - Usually read-only
    - Program Counter (PC) points to the next instruction to be executed
  - Static Data
    - Global/constant/static variables – shared between tasks/threads
  - Heap
    - Dynamic allocated with malloc/free/new/delete
  - Stack (FILO)
    - Used for executing code, method/function calls and return
    - Position in Stack Pointer (SP)

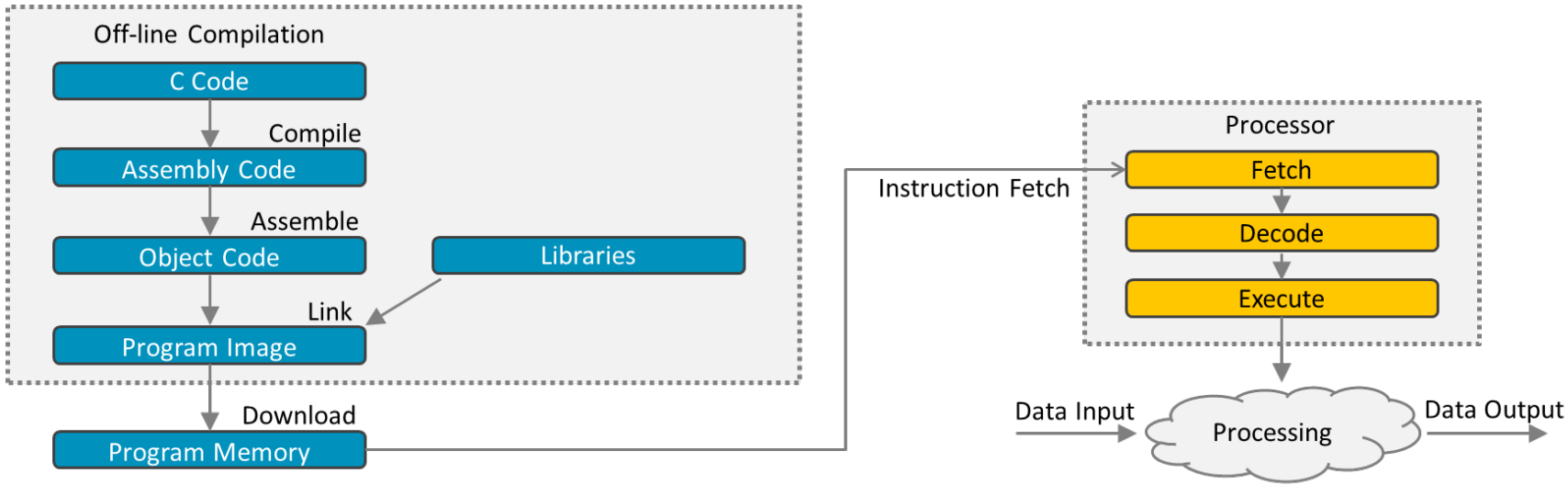


What Does Memory Management Do?

- Memory needs to be managed
  - Both the OS and user tasks need memory.
- Allocation/Partition
  - How to allocate memory sections specific to each use (code, static, dynamic).
  - Done at build and at run time.
- Relocation
  - Changing the memory space dynamically, ideally translation done by hardware.
- Protection
  - Illegal reference to other processes' memory should be detected and stopped at run time.
  - Cortex-M tasks may implement a Memory Protection Unit (MPU).
- Sharing
  - Several tasks/threads may access common parts of the memory

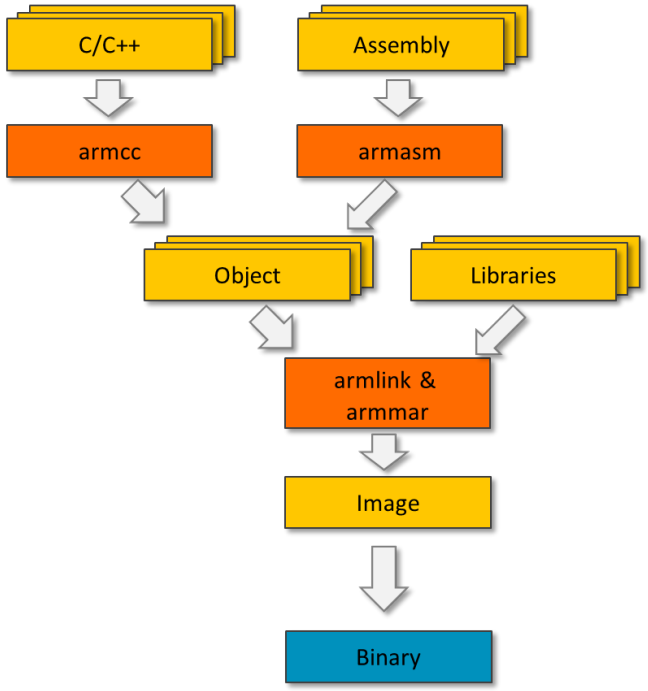
Typical Program-Generation Flow

- The generation of program follows a typical development flow:
  - Compile -> Assemble -> Link -> Download
  - The generated executable file (or program image) is stored in the program memory (normally an on-chip flash memory), to be fetched by the processor



Typical program-generation flow

Compilation using Arm-Based Tools



C, C++
ASM files

Compile/assemble

.O Files
.S Files

Link

.AXF File
.LIB file

Executable

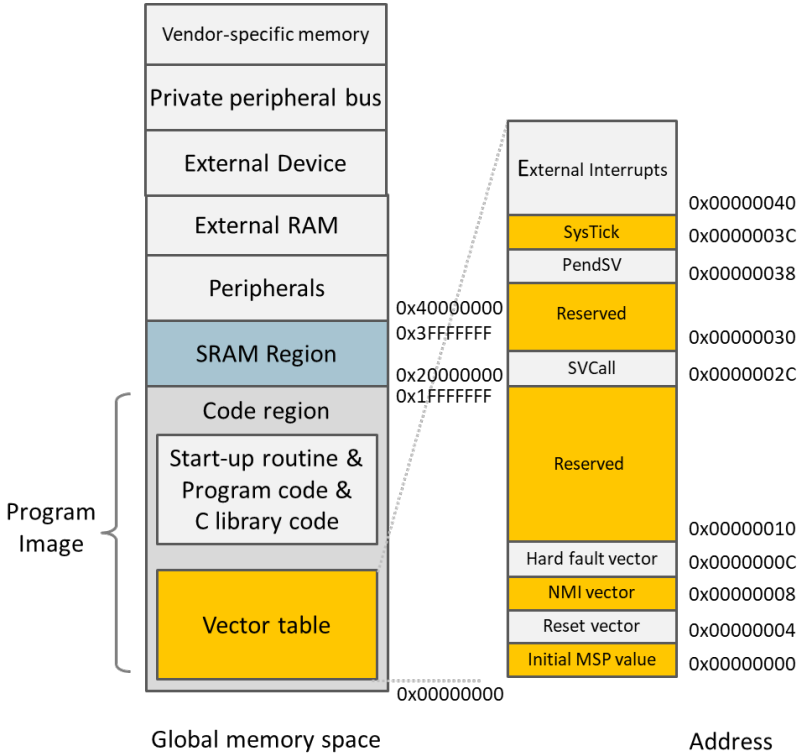
.BIN File
.HEX File
Disassembly File

Compiler Stages

- Pre-processing
  - Replaces macros, defined by an initial hash-tag (#) in the code
  - Merges all subfiles (.c/.cpp, .h) to one complete file
- Parser
  - Reads in C code
  - Checks for syntax errors
  - Forms intermediate code (tree representation)
- High-Level Optimizer: Modifies intermediate code (processor-independent)
- Code Generator
  - Creates assembly code step-by-step from each node of the intermediate code
  - Allocates variable uses to registers
- Low-Level Optimizer: Modifies assembly code (parts are processor-specific)
- Assembler: Creates object code (machine code)
- Linker/Loader: Creates executable image from object file

Cortex-M Program Image

- What is a program image?
  - The program image (sometimes also called the executable file) refers to a piece of fully integrated code that is ready to execute.
- In the Cortex-M, the program image includes:
  - Vector table: includes the starting addresses of exceptions (vectors) and the value of the main stack point (MSP)
  - C start-up routine
  - Program: application code and data
  - C library code: program codes for C library functions



Our Target Device Memory Map

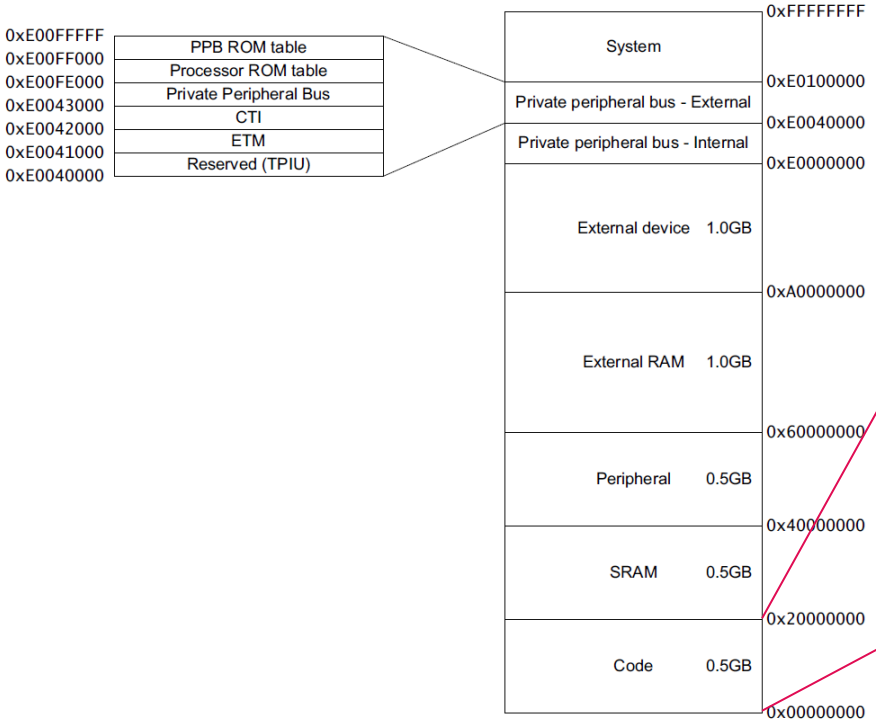


Figure 2-1 System address map

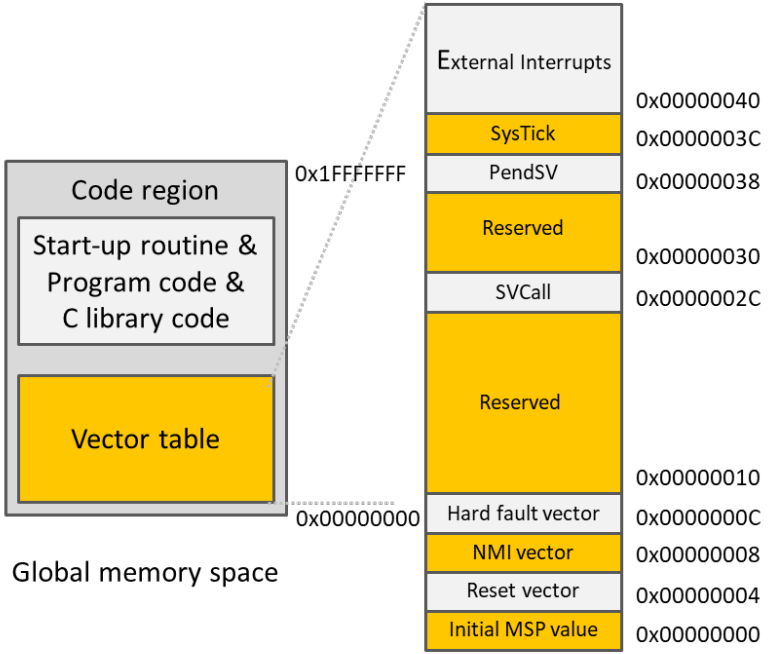
Table 6. Memory map and default device memory area attributes (continued)

Region	Boundary address	Arm® Cortex®-M7	Arm® Cortex®-M4	Type	Attributes	Execute never
Code	0x1FF20000 - 0x1FFFFFFF	Reserved		Normal	Write-through	N
	0x1FF00000 - 0x1FF1FFFF	System Memory	Reserved			
	0x10048000 - 0x1FEFFFFF	Reserved				
	0x10040000 - 0x10047FFF	SRAM3 (Alias)(1)				
	0x10020000 - 0x1003FFFF	SRAM2 (Alias)(1)				
	0x10000000 - 0x1001FFFF	SRAM1 (Alias)(1)				
	0x08200000 - 0x0FFFFFFF	Reserved				
	0x08100000 - 0x081FFFFF	Flash memory bank 2(2)				
	0x08000000 - 0x080FFFFF	Flash memory bank 1(3)				
	0x00010000 - 0x07FFFFFF	Reserved				
	0x00000000 - 0x0000FFFF	ITCM	VTOR REMAP(4)			

1. Alias to maintain Arm® Cortex®-M4 Harvard architecture.
2. Flash memory bank 2 boundary is limited to 0x08100000 - 0x0817FFFF on STM32H745xG/STM32H747xG.
3. Flash memory bank 1 boundary is limited to 0x08000000 - 0x0807FFFF on STM32H745xG/STM32H747xG.
4. Selectable boot memory alias.

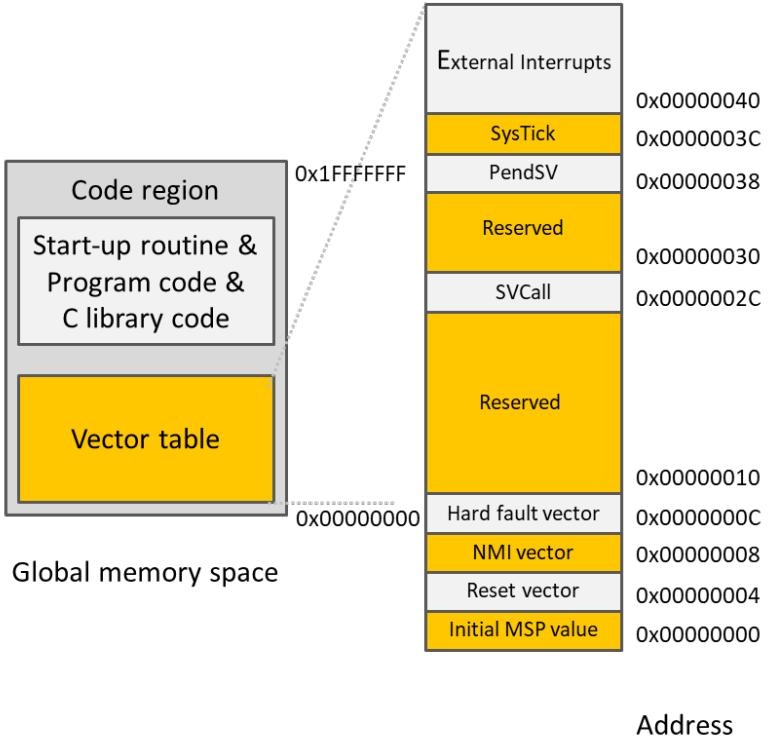
Cortex-M Program Image

- Vector table
  - Contains the starting addresses of exceptions (vectors) and the value of the main stack point (MSP)
- C Start-up code
  - Used to set up data memory and the initialization of values for global data variables
  - Is inserted by the compiler/linker automatically, labeled as '__main' by the Arm compiler, or '__start' by the GNU C compiler



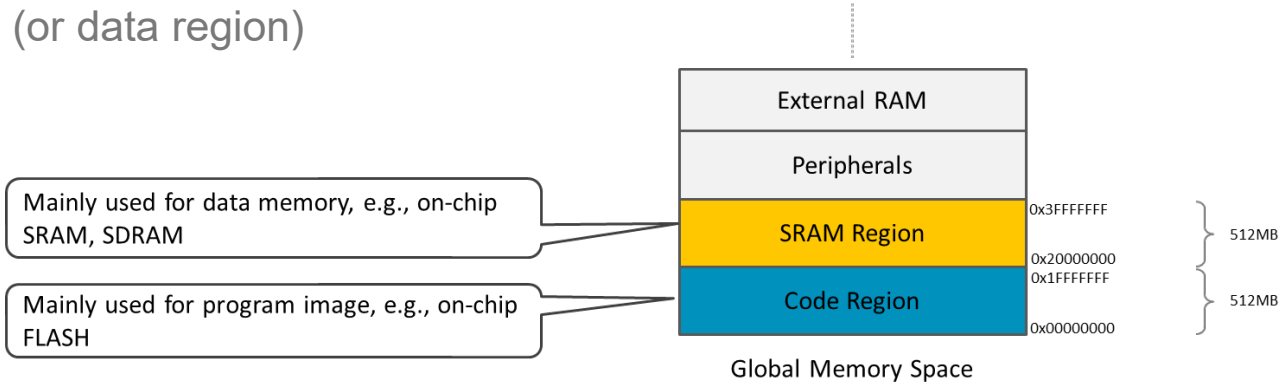
Cortex-M Program Image

- Program code
  - Program code refers to the instructions generated (application code) from the application program and the application data that includes:
    - Initial values of variables: the local variables that are initialized in functions or subroutines during program execution time
    - Constants: used in data values, address of peripherals, character strings, etc.
      - Sometimes stored together in data blocks called literal pools
      - Constant data such as lookup tables, graphics image data (e.g., bit map) can be merged into the program images
- C library code
  - Object codes inserted into the program image by linkers



Program Image in Global Memory

- The program image is stored in the code region in global memory
  - Up to 512 MB memory space range from 0x00000000 to 0x1FFFFFFF
    - On our target device 2 MB in two separate memory banks
  - Usually implemented on non-volatile memory, such as on-chip FLASH memory
  - Normally separated from program data, which is allocated in the SRAM region (or data region)



Codelab

- Understand your Bike Computer program image

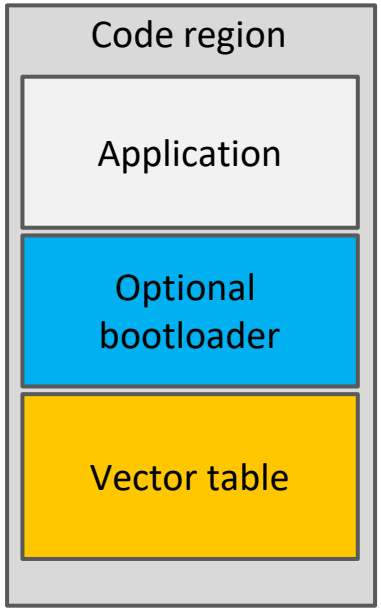
  The Bike Computer program image

- Understand the way a Mbed OS program is started and how memory is initialized

  The Boot Sequence and Memory Initialization

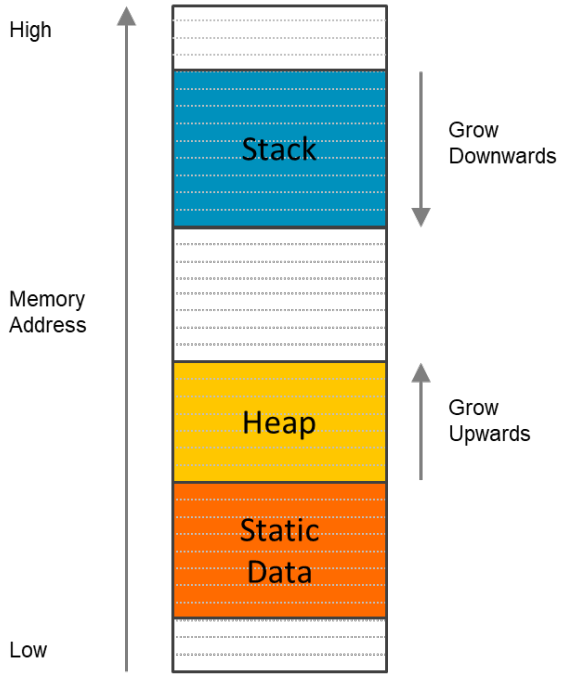
The Mbed Memory Model

- It follows the Cortex-M Memory Model
  - It includes an additional optional bootloader
  - A bootloader is a program that loads Mbed OS when a board is turned on.
  - Usually, the bootloader comes before the application in ROM and the application starts immediately after the bootloader
  - A boot sequence can have several stages of bootloaders, leading to an application.
    - The different stages (including the application) may need to evolve over time, to add features or bug-fixes.
  - Most boot sequences are composed of three stages:
    - Boot selector (also known as root bootloader or stage zero bootloader): does not get upgraded
    - Bootloader: upgradable, with several versions stored on the device.
    - Application: upgradable, with several versions stored on the device.



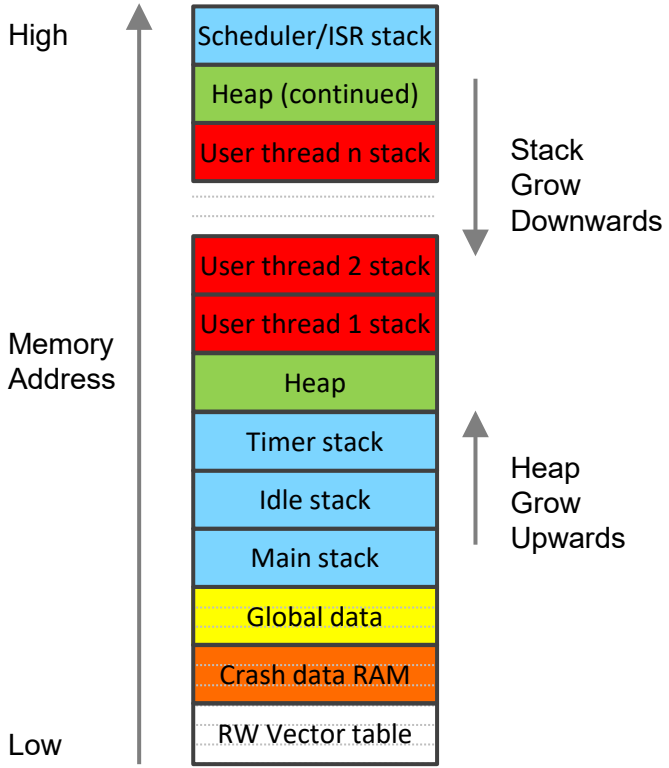
How is Data Stored in RAM?

- Typically, the data can be divided into three sections: static data, stack, and heap
  - Static data: contains global variables and static variables
  - Stack: contains the temporary data for local variables, parameter passing in function calls, registers saving during exceptions, etc.
  - Heap: contains the pieces of memory spaces that are dynamically reserved by calloc() malloc() or new calls.



The Mbed Memory Model

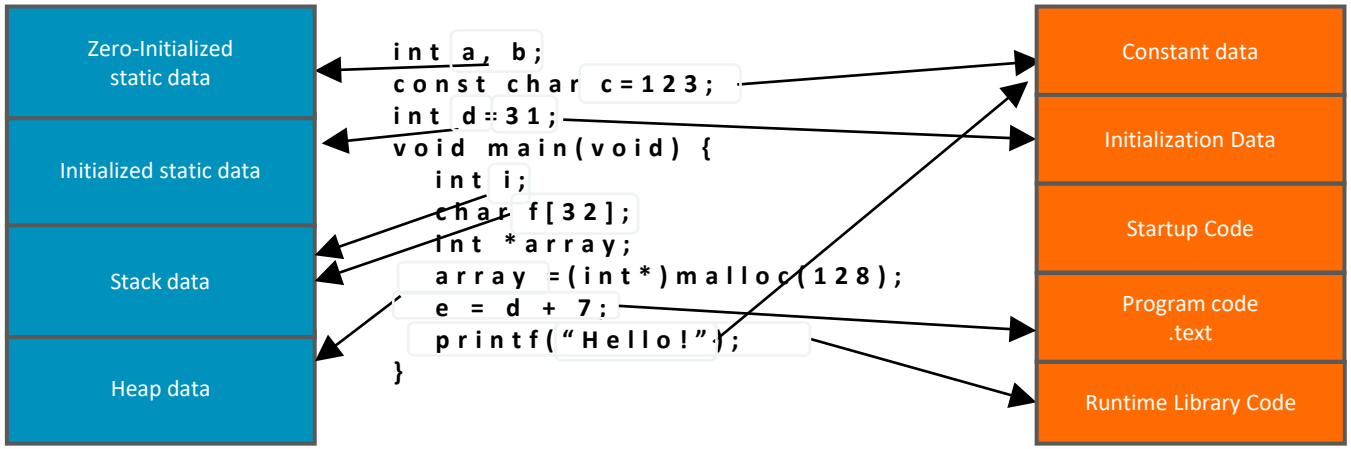
- Inside RAM, you can distinguish two logical types: static and dynamic memory.
- Static memory: allocated at compile time:
  - Vector table (read and write)
  - Crash data RAM
  - Global data
  - Static data
  - Stacks for default threads (main, timer, idle and scheduler/ISR).
- Dynamic memory is allocated at runtime:
  - Heap (dynamic data).
  - Stacks for user threads.
- Stack checking is turned on for all threads, and the kernel errors if it detects an overflow condition.



The Mbed Memory Model

- The stack and heap addresses and sizes are defined at build time
  - They can be defined in either C language (with linker file) or assembly language
- The linker also uses a scatter file that describes the location of the different memory regions. This file contains the definitions of
  - The Code Region (ER_IROM1)
  - The RAM Region (RW_IRAM1)
  - The Heap Region (ARM_LIB_HEAP)
  - The Stack Region (MBED_RAM_START)

Data Storage Through An Example



Usually stored in volatile memories, e.g. SRAM

Usually stored in non-volatile memories e.g. FLASH

What Memory Does a Program Need?

- Can the information change?
  - No: put it in read-only, nonvolatile memory for saving RAM
  - Yes: put it in read/write memory
- How long does the data need to exist?
  - Program scope: statically allocated
  - Function/method scope: automatically allocated on the stack
  - From explicit allocation to explicit deallocation: on the heap
  - Always define the most restrictive scope
  - Use dynamic allocation on the heap with care

Codelab

- Understand what memory goes where

  Static Memory Analysis Using memap
- Optimizing the memory usage of an application

  Reducing memory usage

Data And Memory

- A number of standard data types are supported by the C/C++ language
- However, their implementation depends on the processor architecture and C/C++ compiler
- In Arm programming, the data size is referred to as byte, half word, word, and double word:
  - Byte: 8-bit
  - Half word: 16-bit
  - Word: 32-bit
  - Double word: 64-bit
- The table in the next slide shows the implementation of different data types

Data Types

Data type	Size	Signed Range	Unsigned Range
char, int8_t, uint8_t	Byte	-128 to 127	0 to 255
short, int16_t, uint16_t	Half word	-32768 to 32767	0 to 65535
int, int32_t, uint32_t, long	Word	-2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	Double word	-2^{63} to $2^{63}-1$	0 to $2^{64}-1$
float	Word	$-3.4028234 \times 10^{38}$ to 3.4028234×10^{38}	
double, long double	Double word	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
pointers	Word	0x00 to 0xFFFFFFFF	
enum	Byte/ half word/ word	Smallest possible data type	
bool (C++), _bool(C)	Byte	True or false	
wchar_t	Half word	0 to 65535	

Class Qualifiers

Const

- Never written by program, can be put in ROM to save RAM

Volatile

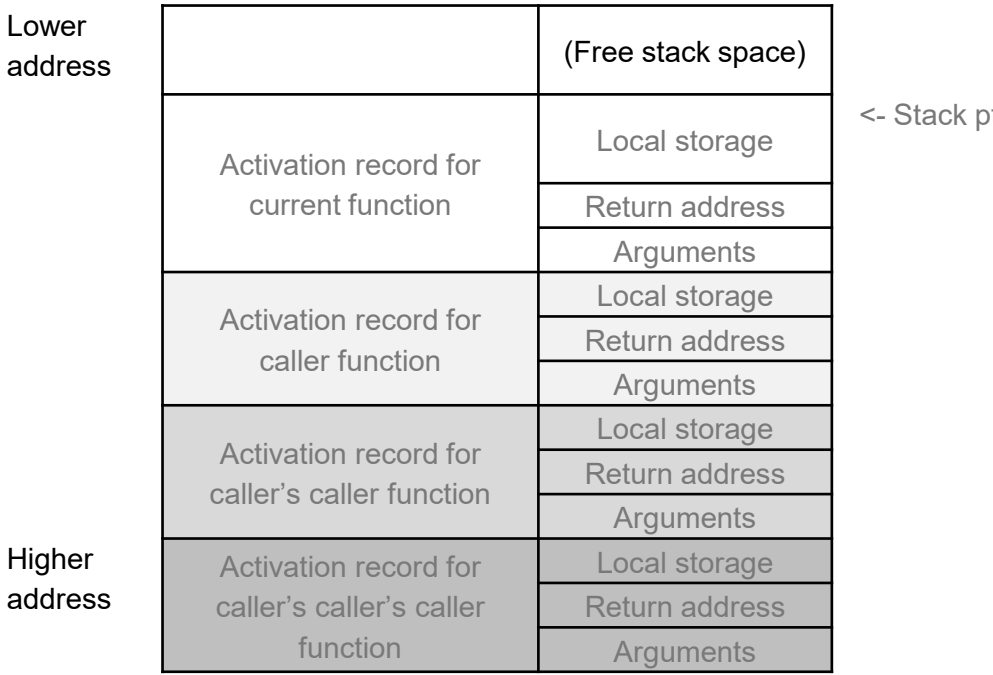
- Can be changed outside of normal program flow: ISR, hardware register
- Compiler must be careful with optimizations

Static

- Declared within function or method, retains value between function/method invocations
- Declared within classes: the field is instantiated once for all class instances and the value is retained for the program lifetime

Activation Record/Stack Frame

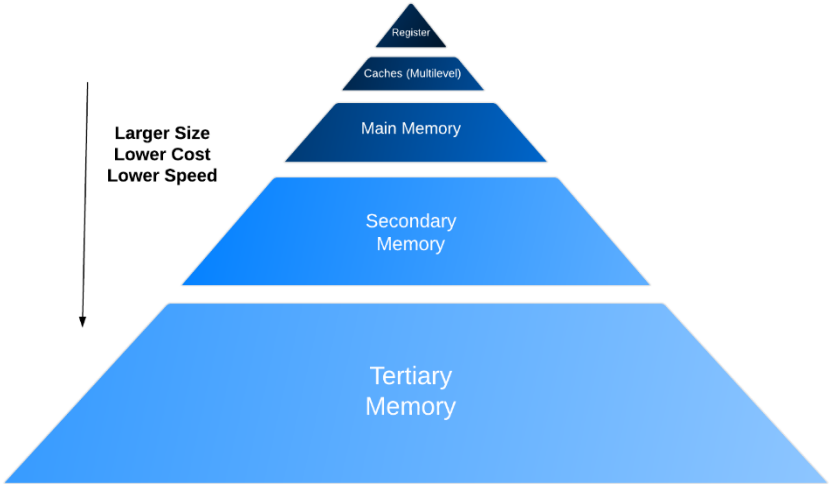
- Activation records are located on the stack
  - Calling a function creates an activation record
  - Returning from a function deletes the activation record
- Automatic variables and housekeeping information are stored in a function's activation record



Accessing Data

- What does it take to get a variable in memory
  - It depends on location, which depends on storage type (static, automatic, dynamic)
  - So the associated cost/time is variable

Memory Hierarchy



- Register: usually one CPU cycle to access
- Cache:
  - Static RAM
- Main Memory
  - Dynamic RAM
  - Volatile data
- Secondary Memory: Flash/Hard disk
- Tertiary Memory: Tape libraries

- Temporal locality
- Spatial locality

- Memory Hierarchy – to exploit the memory locality

Codelab

- Understanding dynamic memory usage

  Runtime memory tracing
- Understanding most common memory usage mistakes

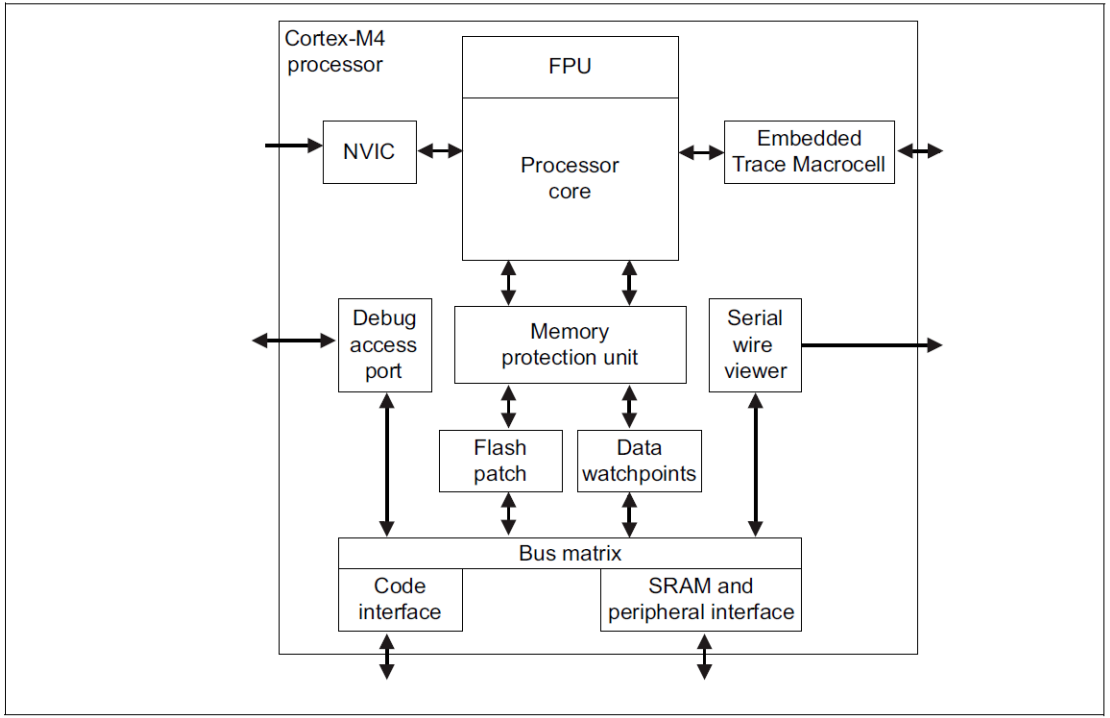
  Hunting for memory bugs

Memory Protection Unit (MPU)

- What is the Memory Protection Unit (MPU) for ARM ?
  - Programmable unit
  - Allows privileged software such as OS kernels, to define memory access permissions.
  - Monitors transactions, including instruction fetches and data accesses
  - Triggers a fault exception when an access violation is detected.
- The privileged software/OS kernel
  - Defines memory regions
  - Assigns memory access permission and memory attributes to each of them.

Memory Protection Unit (Cortex-M4)

Figure 1. STM32 Cortex-M4 implementation



Memory Protection Unit (MPU)

- It is a powerful component of a system for improving the system security
  - It can disallow the user mode/application software (i.e. the software running in unprivileged mode) to access the critical regions of the memory.
- As example, the OS may do the following:
  - Define a region of the memory, say from 0x4000_0000 to 0x4000_FFFF
  - Make this region accessible only while the processor code is running in privileged mode
  - Make this region as read-only
  - Make this region as Execute-Never

MPU Programming

- Through MPU registers, that can be read/written only while the processor is at privileged access level.
- 8 regions of memories are permitted, identified by base address and size.
- Each region can have different 'access rights‘ and MPU can be enabled/disabled for each region.
- Each transaction from the processor is checked against the MPU configuration.
  - If the transaction's attribute matches the 'access rights' of the region, the transaction is successful, and is produced at the processor's interface
  - In case of a mismatch, an exception is generated, and the processor jumps to the exceptional handler.

MPU on Mbed OS

- Memory protection for Mbed OS is enabled automatically for devices that support the MPU API.
- The MPU management functions provided in Mbed OS is limited to turning off the memory protections if necessary
  - Through Mbed MPU API (Mbed MPU Management)
- The memory protection in Mbed OS does the following:
  - It prevents execution from RAM
  - It prevents writing to ROM.

MPU on Mbed OS

- Mbed OS handles MPU management automatically in the following situations:
  - Memory protection is enabled as part of the boot sequence.
  - Memory protection is disabled when starting a new application.
  - Memory protection is disabled while flash programming.
- RAM execute lock (ScopedRamExecutionLock)
  - After boot, execution from RAM is not allowed.
  - Applications/libraries requiring the ability to execute from RAM can enable this by acquiring the RAM execution lock.
- ROM write lock (ScopedRomWriteLock)
  - After boot, writing to ROM is not allowed.
  - Application/libraries requiring the ability to writing to ROM can enable this by acquiring the ROM write lock.