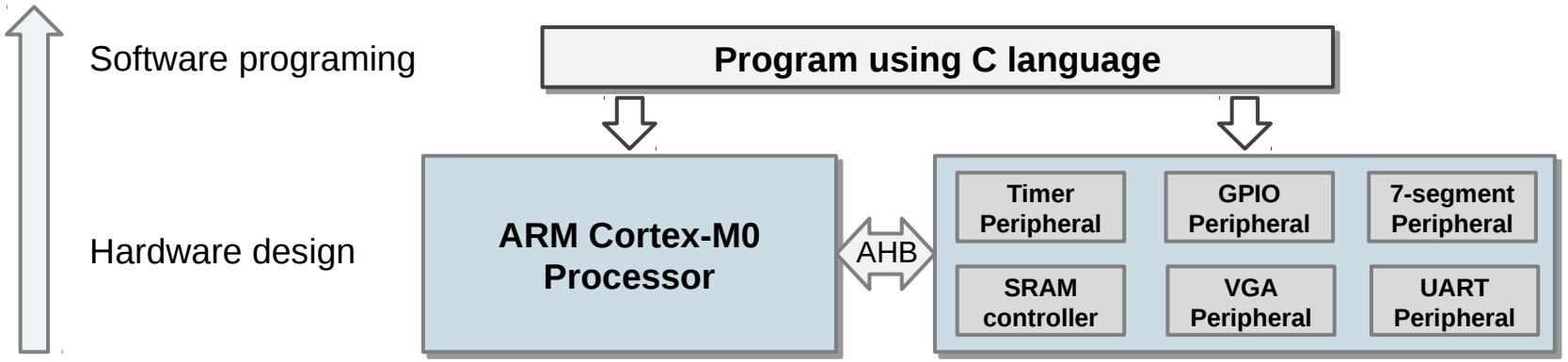
Program SoC using C Language

Module Overview

- General understanding of C, program compilation, program image, data storage, data type, and how to access peripherals using C language;
- Program SoC using C language;
- Mix assembly code to C code;
- Lab practice.



Module Syllabus

- C and Assembly Language Review
- Program SoC using C
- Program Compilation
- Program Image
- Data Storage
- Data Type
- Accessing Peripherals in C
- Lab Practice

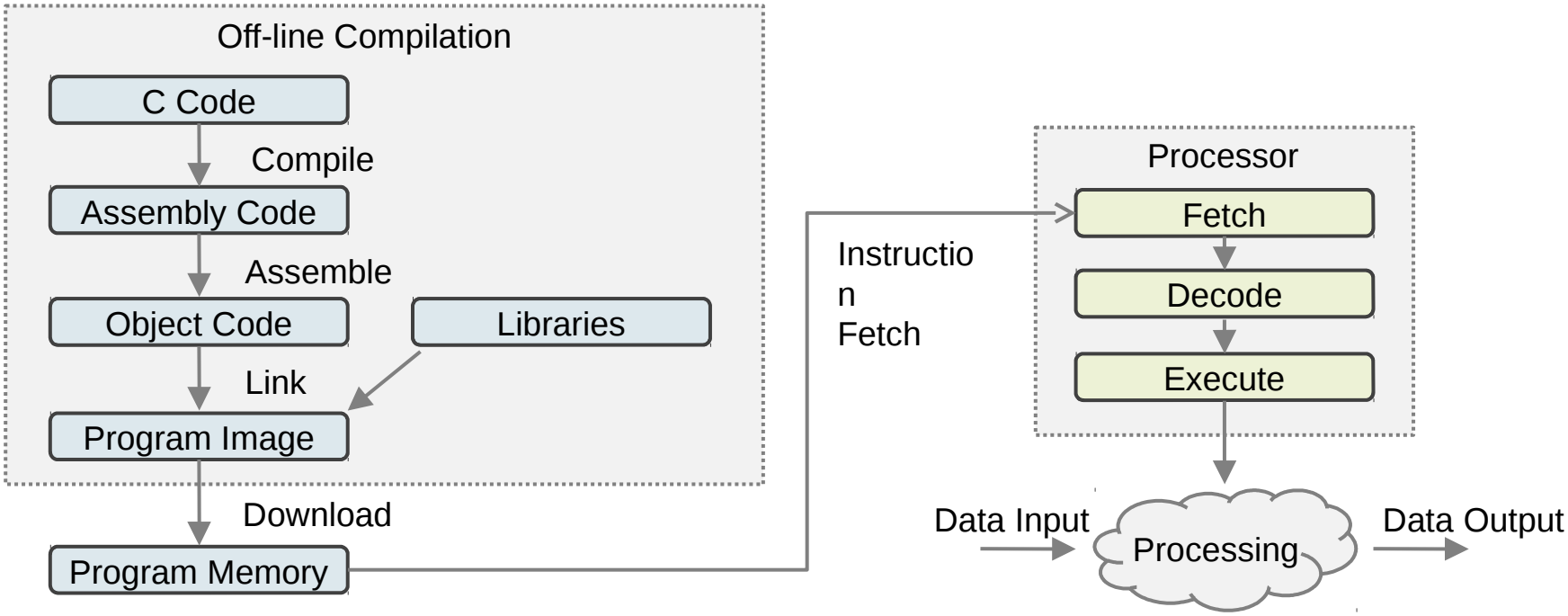
About C Language

C and Assembly Language Review

Language	Advantages	Disadvantage
C	Easy to learn	Limited or no direct access to core registers and stack
	Portable	No direct control over instruction sequence generation
	Easy handling of complex data structures	No direct control over stack usage
Assembly	Allow direct control to each instruction step and all memory	Take longer time to learn
	Allows direct access to instructions that cannot be generated with C	Difficult to manage data structure
		Less portable

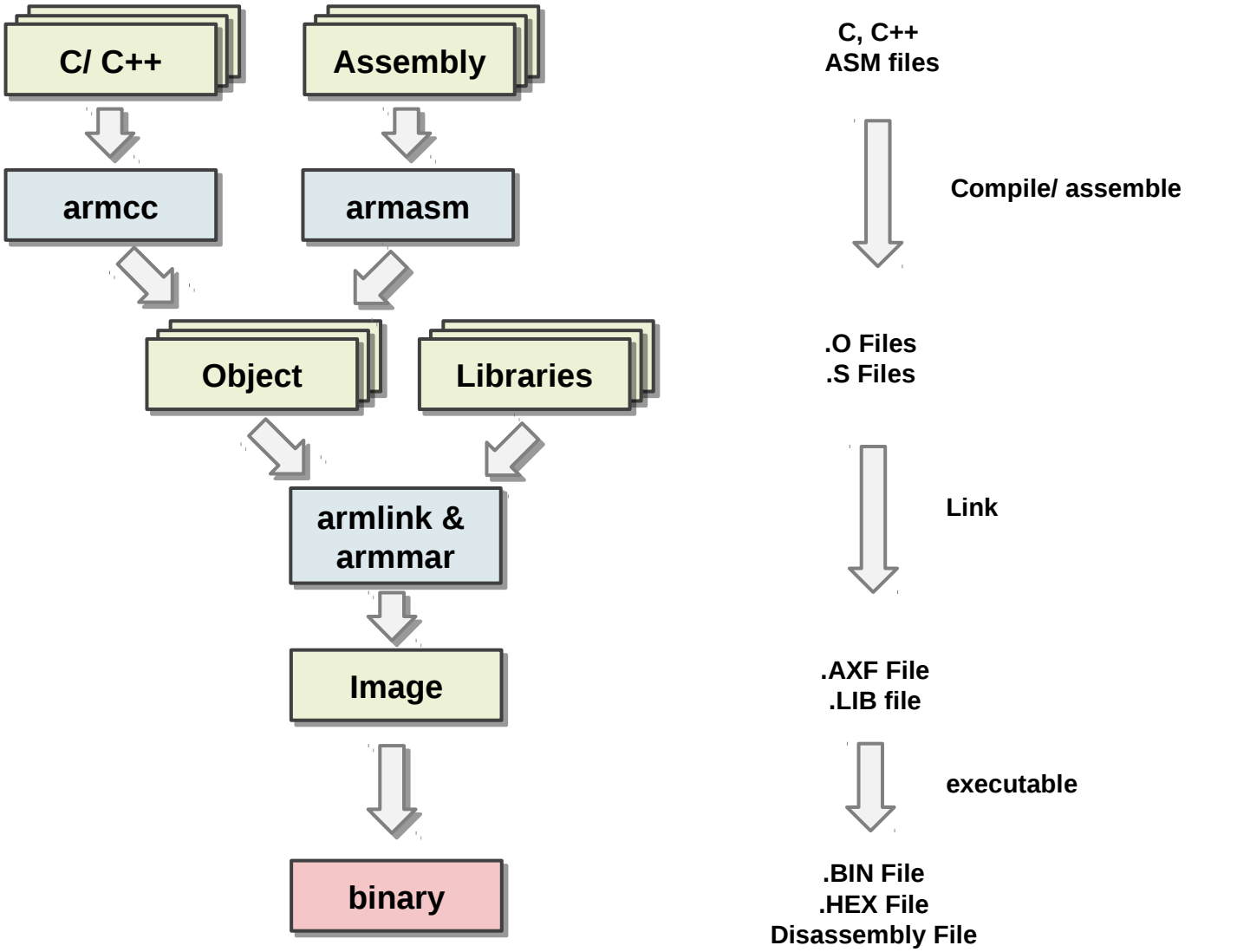
Typical Program-Generation Flow

- 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



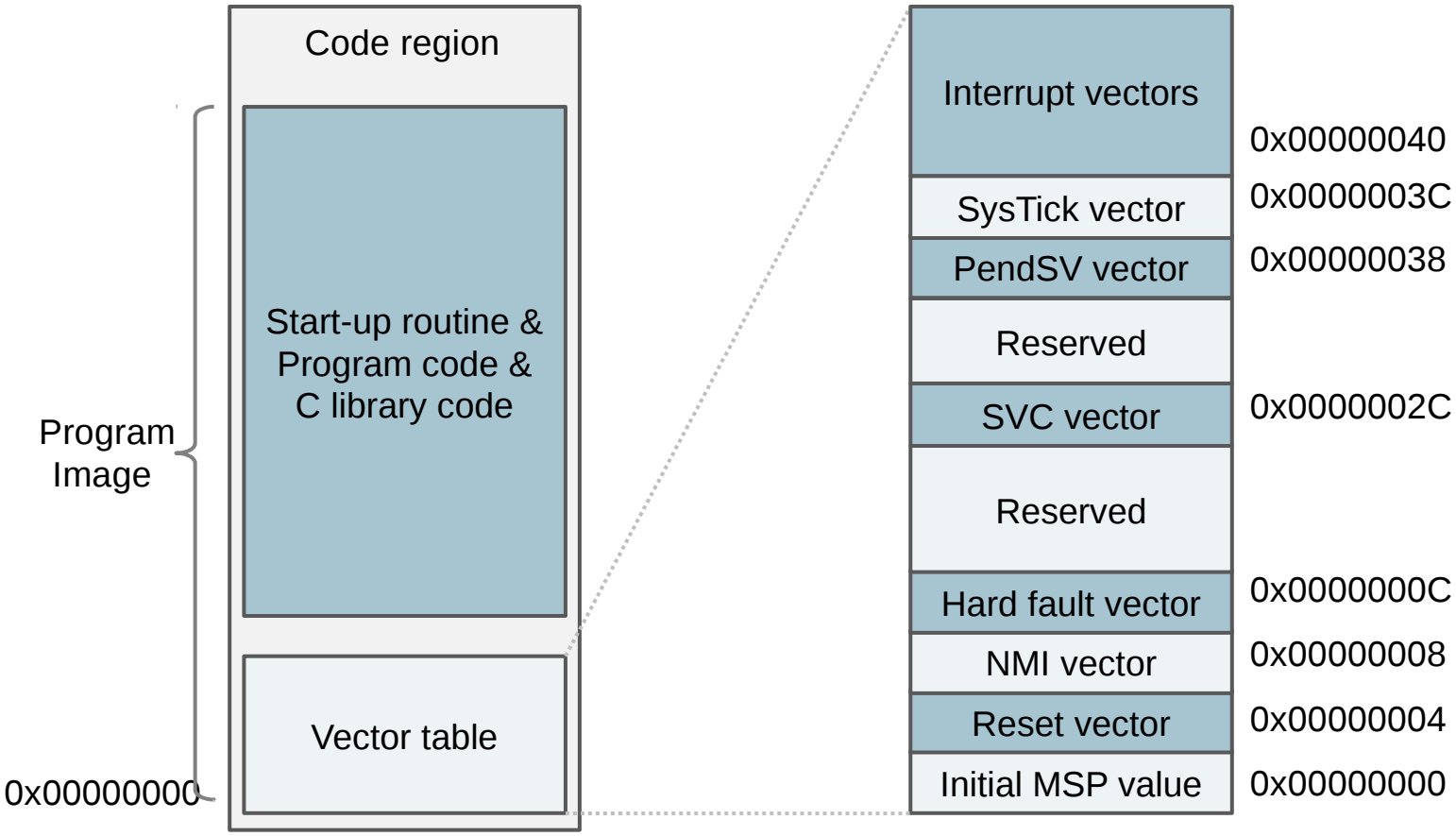
Compiler Stages

- 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

Program Image

- What is a program image
  - The program image (or sometimes referred as executable file) usually refers to a piece of fully integrated code that is ready to execute
- In Cortex-M0, 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 code – application code and data
  - C library code – program codes for C library functions

Program Image



Program Image

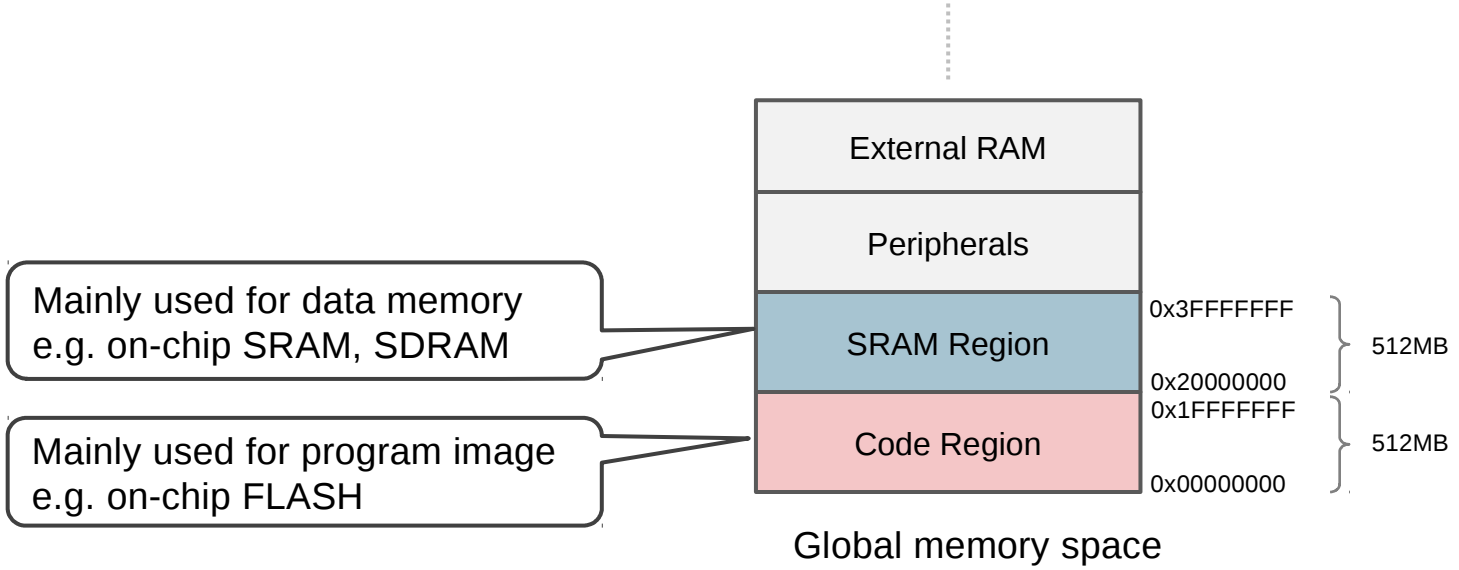
- Vector table
  - Contains the starting addresses of exceptions (vectors) and the value of the main stack point (MSP);
  - Can be programed in either C or assembly;
- C Start-up code
  - Used to set up data memory and the initialization values for global data variables;
  - Is inserted by compiler/ linker automatically, labelled as “__main” by ARM compiler, or “_start” by GNU C compiler;

Program Image

- Program code
  - Program code refers the instructions generated from your application program. The type of data in the program code includes:
    - Initial values of variables – the local variables that initialized in functions or subroutines during program execution time;
    - Constants – used in data values, address of peripherals, character strings, etc…
      - Sometimes are stored together in data blocks called literal pools;
      - Additionally, constant data such as lookup tables, graphics image data (e.g. bit map) can also be merged into the program images;
- C library code
  - Object codes that inserted to the program image by the linkers;
  - For example, the divide function is normally imported from C library since Cortex-M0 does not have a divide instruction;

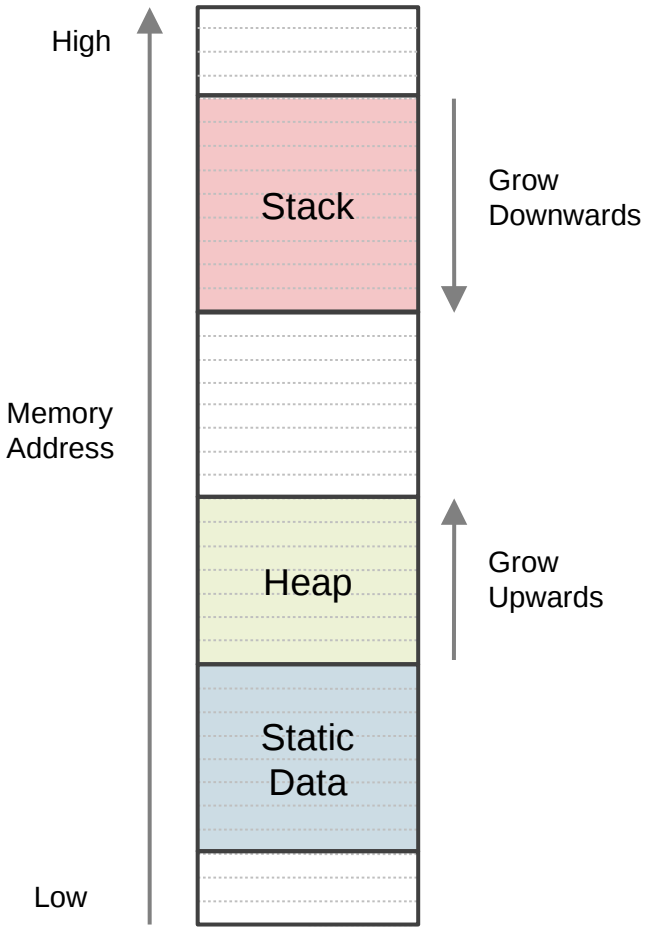
Program Image in Global Memory

- The program image is stored in the code region in the global memory
  - Up to 512 MB memory space range from 0x00000000 to 0x1FFFFFFF
  - 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)



How is Data Stored in RAM

- Typically, the data can be stored in three regions: 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 dynamically reserved by function calls, such as “alloc()”, “malloc()”.



Data Types

- A number of standard data types are supported by the C language.
- However, their implementation is depending on the processor architecture and C compiler.
- In ARM programming, the data size is referred as byte, half word, word, and double word:
  - Byte: 8-bit;
  - Half word: 16-bit;
  - Word: 32-bit
  - Double word: 64-bit
- The following table shows the implementation of different data types.

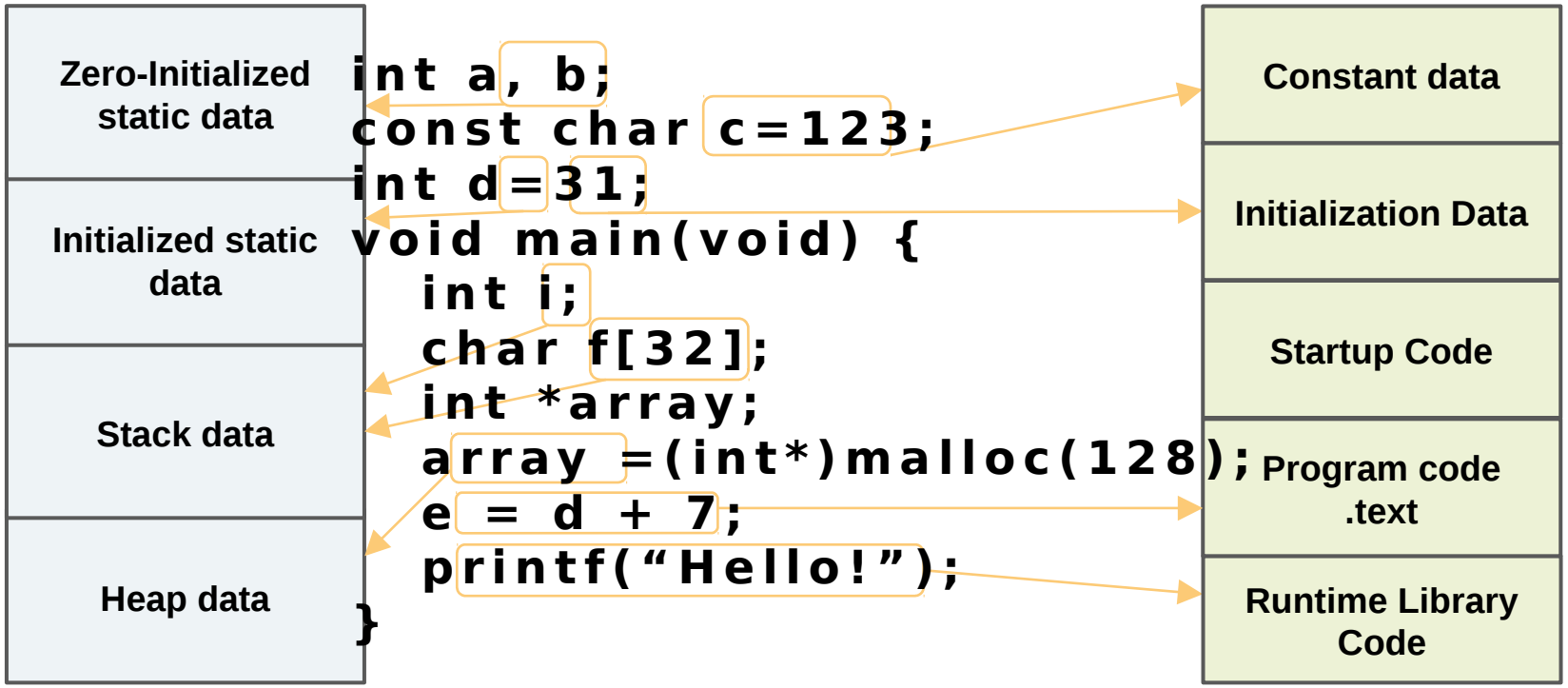
Data Types

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	

Data Types and 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, retains value between function invocations;
  - Scope is limited to function.

Example of Data Storage



Usually stored in volatile memories, e.g. SRAM

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

Program SoC using C

Define Interrupt Vector in C

- The interrupt vector can be defined in either C language or assembly language, for example in C:

```
typedef void(* const ExecFuncPtr)(void) __irq;

#pragma arm section rodata="exceptions_area”

ExecFuncPtr exception_table[] = {
    (ExecFuncPtr)&Image$$ARM_LIB_STACK$$ZI$$Limit, /* Initial SP */
    (ExecFuncPtr)__main,              /* Initial PC */
    NMIException,
    HardFaultException,
    MemManageException,
    BusFaultException,
    UsageFaultException,
    0, 0, 0, 0,               /* Reserved */
    SVCHandler,
    DebugMonitor,
    0,                    /* Reserved */
    PendSVC,
    SysTickHandler
    /* Configurable interrupts start here...*/
};
#pragma arm section
```

Define Stack and Heap

- The stack and heap can be defined in either C language (with linker file) or assembly language, for example in C:

```
/* Set stack and heap parameters */

#define  STACK_BASE     0x10020000        //stack start address
#define  STACK_SIZE       0x5000      //length of the stack
#define   HEAP_BASE      0x10001000   //heap starts address
#define   HEAP_SIZE        0x10000 – 0x6000    //heap length

/* inker generated stack base addresses */

extern unsigned int Image$$ARM_LIB_STACK$$ZI$$Limit
extern unsigned int Image$$ARM_LIB_STACKHEAP$$ZI$$Limit

…
```

Define Stack and Heap

- Define stack and heap in assembly language:

```
Stack_Size              EQU     0x00000400      ; 256KB of STACK

                        AREA     STACK, NOINIT, READWRITE, ALIGN=4
Stack_Mem               SPACE   Stack_Size
__initial_sp

Heap_Size               EQU     0x00000400      ; 1MB of HEAP

                        AREA    HEAP, NOINIT, READWRITE, ALIGN=4
__heap_base
Heap_Mem                SPACE   Heap_Size
__heap_limit
```

Accessing Peripherals in C

- Define base addresses for peripherals, e.g.

```
#define AHB_VGA_BASE          0x50000000
#define AHB_UART_BASE         0x51000000
#define AHB_TIMER_BASE            0x52000000
#define AHB_GPIO_BASE         0x53000000
#define AHB_7SEG_BASE         0x54000000
#define NVIC_INT_ENABLE           0xE000E100
```

- Write a value to a peripheral register, e.g.

```
*(unsigned int*) AHB_TIMER_BASE = 0x3FFFF;//store a value to the peripheral
```

- Read a value from a peripheral register, e.g.

```
i=*(unsigned int*) AHB_GPIO_BASE; //read a value from the peripheral
```

Calling a C Function from Assembly

- When a C function is called from an assembly file, the following areas should be aware:
  - Register R0, R1, R2, R3, R12, and LR could be changed, hence it is better to save them to the stack;
  - The value of SP should be aligned to a double-word address boundary;
  - Input parameters have to be stored in the correct registers, for example, registers R0 to R3 can be used for passing four parameters;
  - The return value is usually stored in R0;

Calling a C Function from Assembly

- ISR can be written in either assembly or C language, for example in C:

```
void UART_ISR() {
    char c;
    c=*(char*) AHB_UART_BASE;     //read a character from UART
    …
}
```

- Call a C function from the assembly code, for example:

```
UART_Handler     PROC
                 EXPORT UART_Handler  // label name in assembly
        IMPORT UART_ISR       // function name in C
                 PUSH    {R0,R1,R2,LR}// context saving
        BL UART_ISR       // branch to ISR written in C
                 POP     {R0,R1,R2,PC}// context restoring
                 ENDP
```

Calling an Assembly Function from C

- When calling an assembly function from C code, following areas should be aware:
  - If registers R4 to R11 need to be changed, they have to be stacked and restored in the assembly function;
  - If another function is called inside the assembly function, LR register needs to be saved on the stack and used for return;
  - The function return value is normally stored in R0.

Calling an Assembly Function from C

- Write a function in assembly, for example:

```
        EXPORT add_asm
add_asm     FUNCTION
                ADDS R0, R0, R1
                ADDS R0, R0, R2
                ADDS R0, R0, R3
        BX  LR      ; result is returned in R0
                ENDFUNC
```

- Calling an assembly function in C, for example:

```
external int  add_asm( int k1, int k2, int k3, int k4);
void main {
    int x;
    x = add_asm (11,22,33,44);    // call assembly function
    …
}
```

Embedded Assembly

- The embedded assembler allows developer to write assembly functions inside C files, for example:

```
_asm int  add_asm(  int k1, int k2, int k3, int k4) {

        ADDSR0, R0, R1
                ADDSR0, R0, R2
                ADDSR0, R0, R3
        BX   LR
}

void main {
    int x;
    x = add_asm (11,22,33,44);    // call assembly function
    …
}
```

Lab Practice

Lab Practice

- Step1- Use C language to program the processor and the peripherals, use the simulator to observe the variables, processor registers and data in the memory;
- Step2- Compare the assembly previously written by yourself, with the assembly compiled by the compiler, evaluate programing effort and the coding efficiency
- Step3- Demonstrate your SoC using a mixture of C and assembly language;
  - For example, displaying a counting-up number on VGA (or 7-segment display), printing and receiving text through UART, reading a value from the switches and writing a value to the LEDs.

Useful Resources

- Reference1
  - Book: “The Definitive Guide to the ARM Cortex-M0” by Joseph Yiu, ISBN-10: 0123854776, ISBN-13: 978-0123854773, March 11, 2011

- Reference2
  - Cortex-M0 Devices Generic Users Guide:

    http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf