M-profile Booting & Initialization

Learning objectives

At the end of this module you will be able to:

- Develop startup and initialization code
- Write memory description files
- Choose how to utilize the C library
- Create an executable image suitable for booting on your device

Embedded development process

Development environment

Standalone Embedded Application

In the process of moving from a development environment to a standalone embedded application, several issues need to be considered:

- Application startup
- Target memory map
- C library use of hardware

Agenda

Default compilation tool behavior

Writing startup and initialization code

- Initial startup and system initialization code using CMSIS-Core
- Post startup initialization

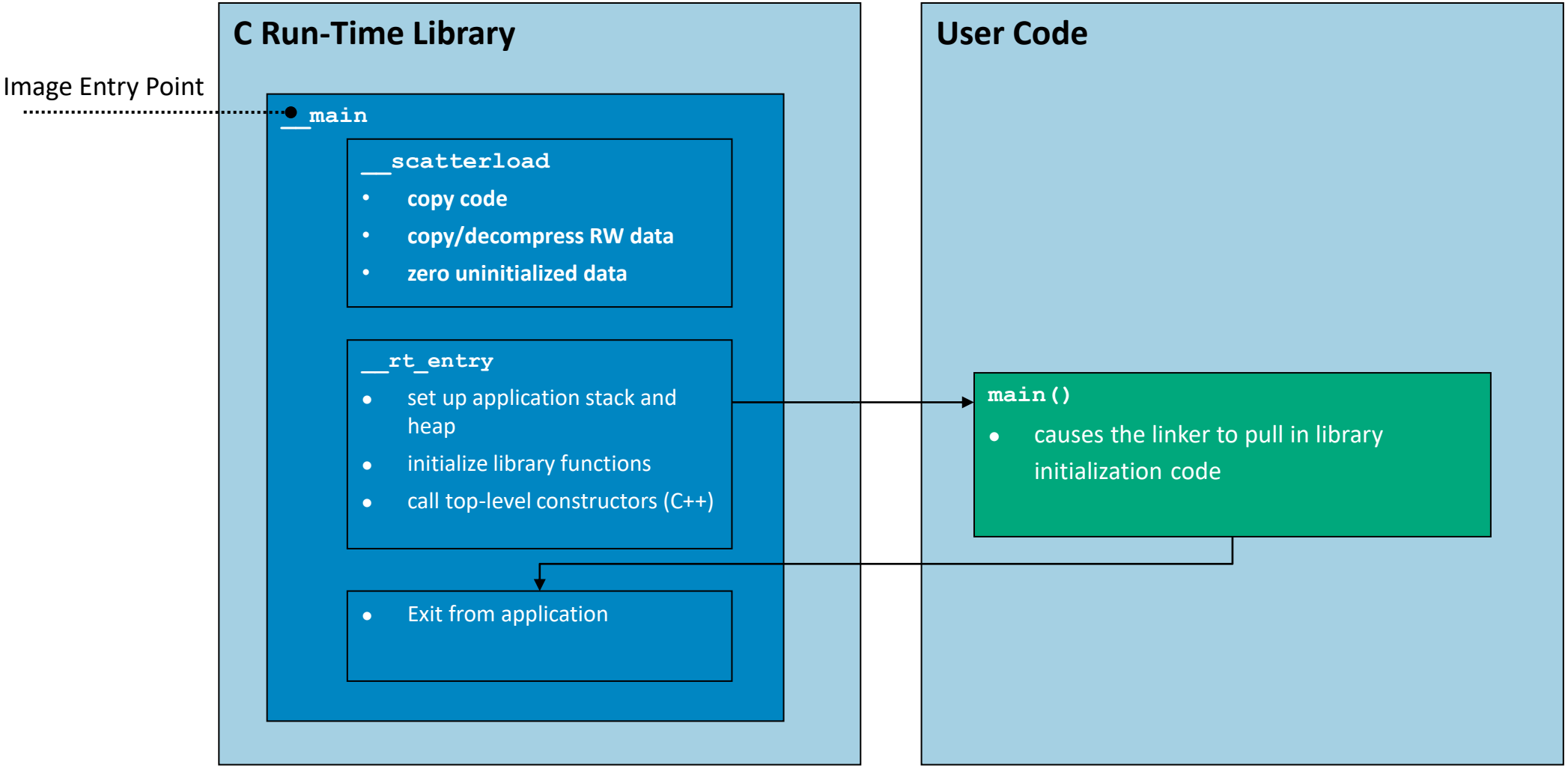
Tailoring the image memory map to a device

- Writing memory description files (scatter files / linker scripts)
- Placing the stack(s) and heap
- Further memory map considerations

Retargeting the C library behavior to work with your system

Building an image ready for booting

Default startup and initialization sequence – Arm Compiler (1)



Default startup and initialization sequence – Arm Compiler (2)

The **`main()`** function causes the Arm Linker (**`armlink`**) to pull-in functions to set up the stack and heap, and initialize the C library

The function **`__main`** is the entry point to the C library

- Unless it is changed, `__main` is the default entry point to the ELF image that the Arm linker uses when creating the image
- `__main` calls **`__scatterload`** and **`__rt_entry`**

`__scatterload` initializes execution-time regions

- It uses a region table that contains address information and determines whether a region needs to be copied, zeroed, or decompressed (more later)

`__rt_entry` calls various initialization functions and then calls the user-level `main()`

- It is also responsible for setting up the stack and heap
  - Setting up the stack depends on the method specified by the user (more later)
- If `main()` returns, its return value is passed to **`exit()`** and the application exits

__main

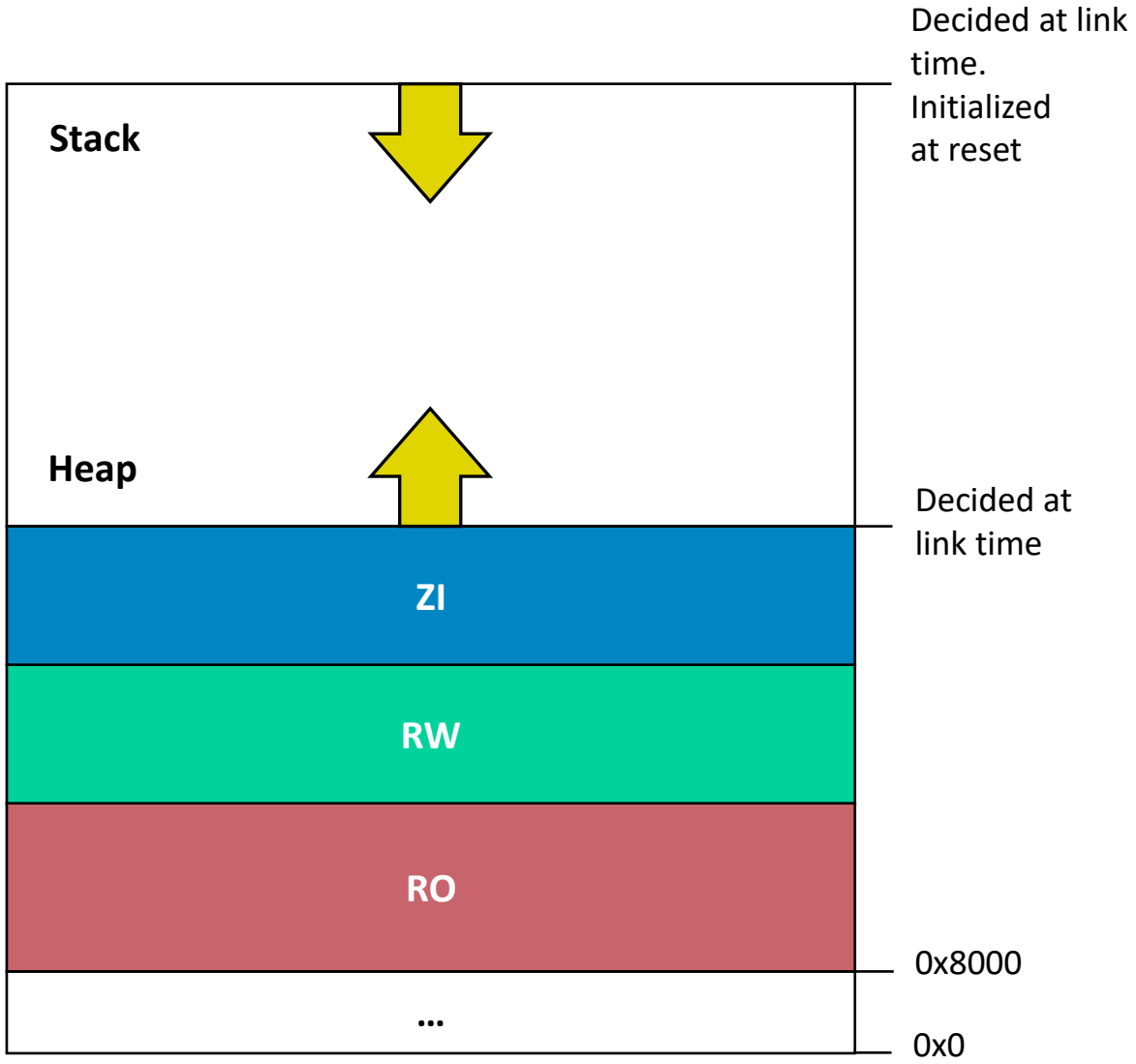
- __scatterload
- __rt_entry
  - __rt_lib_init
  - main (user's code)
  - exit

Default memory map - Arm Compiler

By default, the Arm Compiler links code to load and execute at 0x8000

The heap is placed directly above the data region

The stack base location is defined at link time and configured by the processor at reset



Default linker placement rules – Arm Compiler

Within each execution region, the linker orders code and data according to some basic rules

The basic ordering is organized by attribute

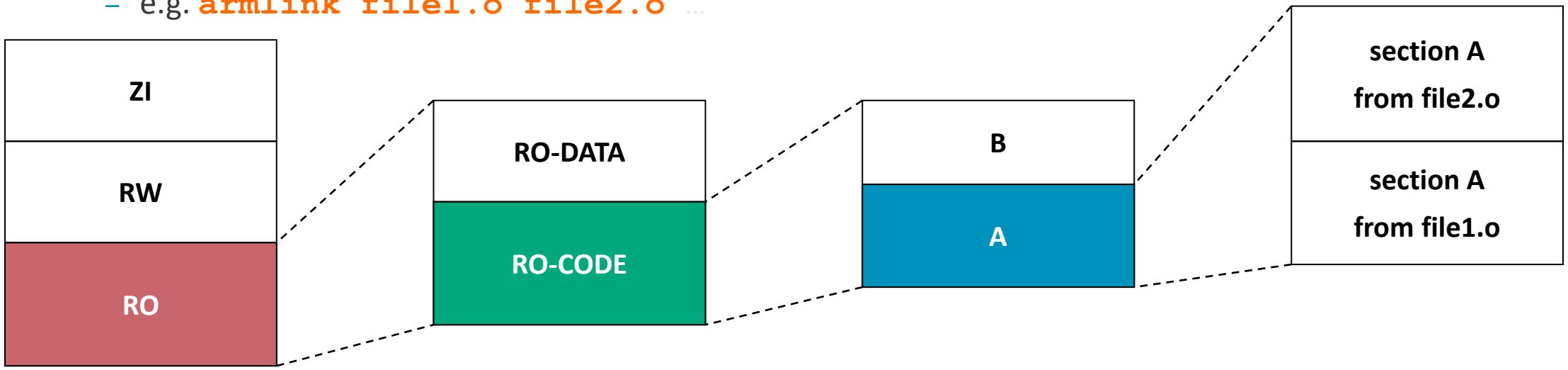
- RO precedes RW which precedes ZI
- Within the same attribute, code is placed before data

Further ordering is determined

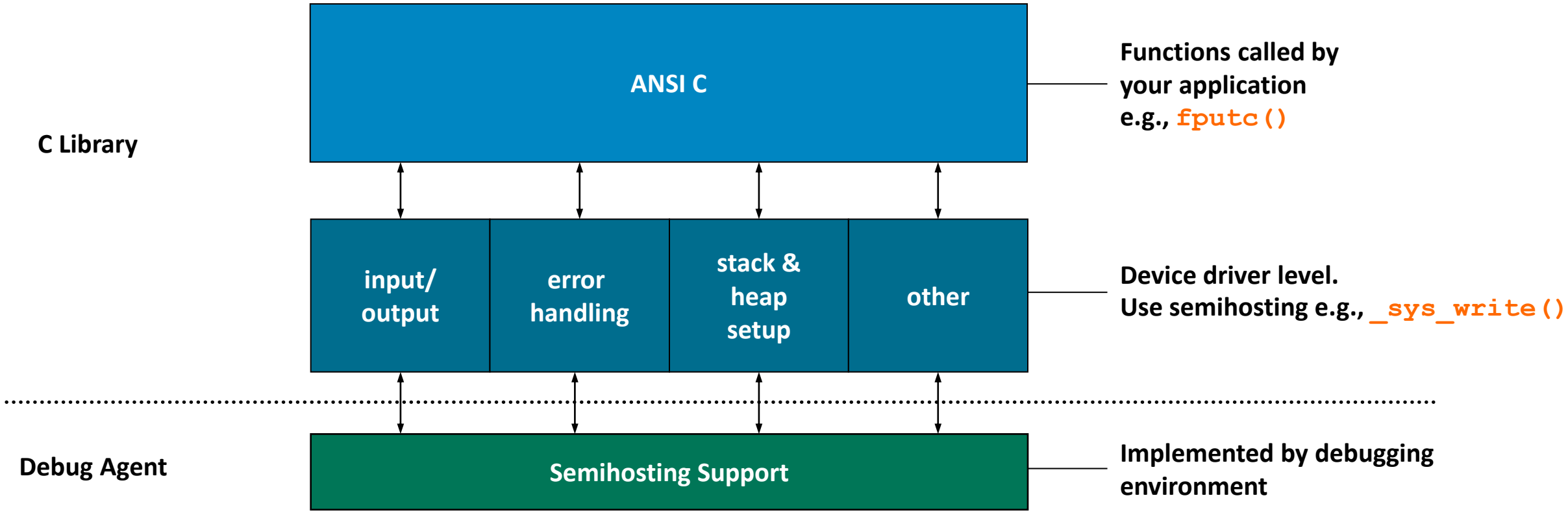
- By input section name in alphabetical order, then …

  …by the order objects are specified on the armlink command line

  - e.g. `armlink file1.o file2.o …`



Default C library – Arm Compiler



Target-dependent C library functionality is supported “out-of-the-box” by a device driver level that makes use of debug target semihosting support

T32 C libraries – Arm Compiler

The Arm Compiler provides several ready-built T32 (formerly Thumb) C libraries with ‘semihosting’ support, for example:

```
c_w.l      - Little-endian C library
h_w.b      - Big-endian helper library
f_ws.l/.b  - Software floating-point library
```

- They use **`BKPT 0xAB`** to communicate with the debug system

Normally, you do not have to list any of these libraries explicitly on the linker command line

- The Arm linker automatically selects the correct C or C++ libraries to use (and it might use several), based on the accumulation of the object attributes

Smaller, lower functionality “microlib” C libraries help to reduce overall code size

Agenda

Default compilation tool behavior

Writing startup and initialization code

- Initial startup and system initialization code using CMSIS-Core
- Post startup initialization

Tailoring the image memory map to a device

- Writing memory description files (scatter files / linker scripts)
- Placing the stack(s) and heap
- Further memory map considerations

Retargeting the C library behavior to work with your system

Building an image ready for booting

Reset and initialization

In real systems you will probably not want your application's entry point to be a compiler-specific library function like `__main` or `_start`

M-profile processors start up in privileged Thread mode because system initialization can only be executed in privileged mode

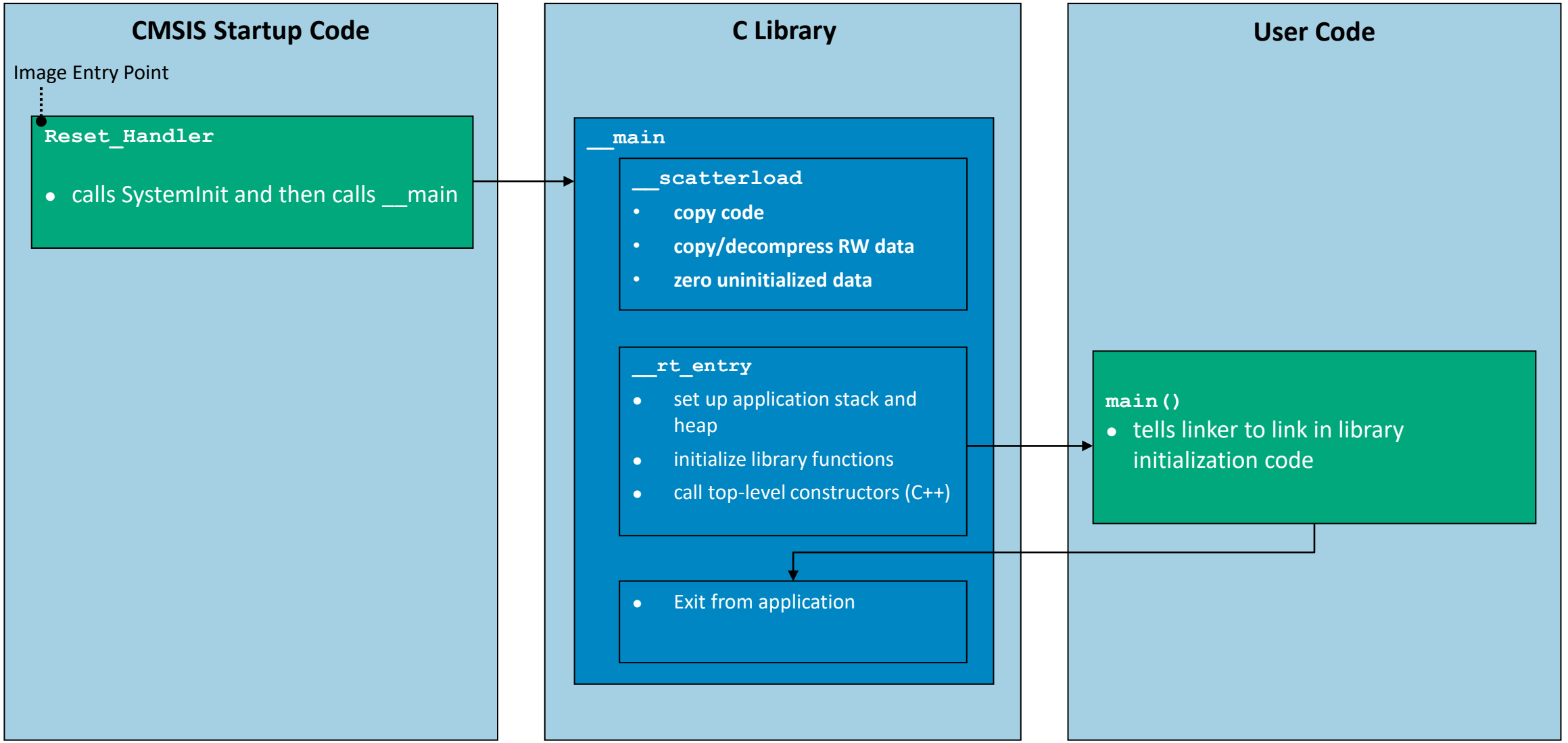
- Need to carry out privileged operations, e.g., setup MPU, SAU, enable interrupts etc.

Before entering the main application, the startup code might need to configure and initialize various system components, for example:

- Setup and enable MPU (if available)
- Setup and enable SAU (if Security Extension is enabled)
- Complete any memory (Flash/RAM) remapping
- Prioritize and enable interrupts
- Initialize RTOS (if present)
- (Last) Change Thread mode to Unprivileged – if required
  - Availability of unprivileged mode is IMPLEMENTATION DEFINED in Armv6-M and Armv8-M Baseline

CMSIS-Core: Startup & System Initialization

Possible initialization sequence (1) – Arm Compiler



CMSIS startup and initialization

CMSIS-Core provides the following files for an embedded application:

- Startup File `startup_<device>.c` (or `startup_<device>.s`) with reset handler and exception vectors
- System Configuration Files `system_<device>.c` and `system_<device.h>` with general device configuration (i.e., for clock and features/peripheral setup)
- Device Header File `<device.h>` gives access to processor core and all peripherals

For any existing device, Arm recommends including CMSIS-Core startup and system initialization code from the device vendor

- If this is not available, startup and system initialization code will have to be created manually or by using an alternative, non-CMSIS compliant solution

When designing a new device, Arm recommends that the device vendor:

- Creates a CMSIS-SVD description to automatically generate CMSIS compliant device header files
  - Formalizes the programmer's view for device-specific peripherals
- Adapts the system initialization and startup template files to their device

CMSIS-Core: startup file

The startup file `startup_<device>.c` or `startup_<device>.s` contains the following:

- Vector table
  - M-profile exception vectors
  - Interrupt vectors that are device specific
  - Weak functions that implement default routines
- Reset handler
  - calls **`SystemInit`** followed by the **`__PROGRAM_START()`** macro
  - **`__PROGRAM_START()`** expands onto default C run-time entry point functions such as **`__main`** and **`_start`**
- Stack and heap configuration

The file exists for each supported toolchain and is the only toolchain specific CMSIS file

- Arm Compiler, GCC, IAR
- For example, for Cortex-M33, there are some device startup file templates available from: https://github.com/ARM-software/CMSIS_5/tree/develop/Device/ARM/ARMCM33/Source

CMSIS-Core: C & assembly language vector tables

```
                .section RESET
                .align  2
                .globl  __Vectors
                .globl  __Vectors_End
                .globl  __Vectors_Size
__Vectors:
                .long   __INITIAL_SP                    /*     Initial Stack Pointer */
                .long   Reset_Handler                   /*     Reset Handler */
                .long   NMI_Handler                     /* -14 NMI Handler */
                .long   HardFault_Handler               /* -13 Hard Fault Handler */
                .long   MemManage_Handler               /* -12 MPU Fault Handler */
                .long   BusFault_Handler                /* -11 Bus Fault Handler */
                .long   UsageFault_Handler              /* -10 Usage Fault Handler */
                .long   SecureFault_Handler             /*  -9 Secure Fault Handler */
                .long   0                               /*     Reserved */
                .long   0                               /*     Reserved */
                .long   0                               /*     Reserved */
                .long   SVC_Handler                     /*  -5 SVCall Handler */
                .long   DebugMon_Handler                /*  -4 Debug Monitor Handler */
                .long   0                               /*     Reserved */
                .long   PendSV_Handler                  /*  -2 PendSV Handler */
                .long   SysTick_Handler                 /*  -1 SysTick Handler */

                /* Interrupts */
                .long   Interrupt0_Handler              /*   0 Interrupt 0 */
                .long   Interrupt1_Handler              /*   1 Interrupt 1 */
                .long   Interrupt2_Handler              /*   2 Interrupt 2 */
                .long   Interrupt3_Handler              /*   3 Interrupt 3 */
                .long   Interrupt4_Handler              /*   4 Interrupt 4 */
                .long   Interrupt5_Handler              /*   5 Interrupt 5 */
                .long   Interrupt6_Handler              /*   6 Interrupt 6 */
                .long   Interrupt7_Handler              /*   7 Interrupt 7 */
                .long   Interrupt8_Handler              /*   8 Interrupt 8 */
                .long   Interrupt9_Handler              /*   9 Interrupt 9 */

                .space  (470 * 4)                       /* Interrupts 10 .. 480 are left out */
__Vectors_End:
                .equ    __Vectors_Size, __Vectors_End - __Vectors
                .size   __Vectors, . - __Vectors
```

Cortex-M33 CMSIS-Core Vector Table Templates

```
extern const VECTOR_TABLE_Type __VECTOR_TABLE[496];
       const VECTOR_TABLE_Type __VECTOR_TABLE[496] __VECTOR_TABLE_ATTRIBUTE = {
  (VECTOR_TABLE_Type)(&__INITIAL_SP),       /*     Initial Stack Pointer */
  Reset_Handler,                            /*     Reset Handler */
  NMI_Handler,                              /* -14 NMI Handler */
  HardFault_Handler,                        /* -13 Hard Fault Handler */
  MemManage_Handler,                        /* -12 MPU Fault Handler */
  BusFault_Handler,                         /* -11 Bus Fault Handler */
  UsageFault_Handler,                       /* -10 Usage Fault Handler */
  SecureFault_Handler,                      /*  -9 Secure Fault Handler */
  0,                                        /*     Reserved */
  0,                                        /*     Reserved */
  0,                                        /*     Reserved */
  SVC_Handler,                              /*  -5 SVCall Handler */
  DebugMon_Handler,                         /*  -4 Debug Monitor Handler */
  0,                                        /*     Reserved */
  PendSV_Handler,                           /*  -2 PendSV Handler */
  SysTick_Handler,                          /*  -1 SysTick Handler */

  /* Interrupts */
  Interrupt0_Handler,                       /*   0 Interrupt 0 */
  Interrupt1_Handler,                       /*   1 Interrupt 1 */
  Interrupt2_Handler,                       /*   2 Interrupt 2 */
  Interrupt3_Handler,                       /*   3 Interrupt 3 */
  Interrupt4_Handler,                       /*   4 Interrupt 4 */
  Interrupt5_Handler,                       /*   5 Interrupt 5 */
  Interrupt6_Handler,                       /*   6 Interrupt 6 */
  Interrupt7_Handler,                       /*   7 Interrupt 7 */
  Interrupt8_Handler,                       /*   8 Interrupt 8 */
  Interrupt9_Handler                        /*   9 Interrupt 9 */
                                            /* Interrupts 10 .. 480 are left out */
};
```

CMSIS-Core: C & assembly language exception handlers

```
/*----------------------------------------------------------------------------
  Reset Handler called on controller reset
 *----------------------------------------------------------------------------*/
__NO_RETURN void Reset_Handler(void)
{
  __set_PSP((uint32_t)(&__INITIAL_SP));

  __set_MSPLIM((uint32_t)(&__STACK_LIMIT));
  __set_PSPLIM((uint32_t)(&__STACK_LIMIT));

#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
  __TZ_set_STACKSEAL_S((uint32_t *)(&__STACK_SEAL));
#endif

  SystemInit();                             /* CMSIS System Initialization */
  __PROGRAM_START();                        /* Enter PreMain (C library entry point) */
}
```

Example:

Cortex-M33 CMSIS-Core Exception Handler Templates that refer to extern symbols defined by memory description files (more later)

```
/*----------------------------------------------------------------------------
  Hard Fault Handler
 *----------------------------------------------------------------------------*/
void HardFault_Handler(void)
{
  while(1);
}

/*----------------------------------------------------------------------------
  Default Handler for Exceptions / Interrupts
 *----------------------------------------------------------------------------*/
void Default_Handler(void)
{
  while(1);
}


/*----------------------------------------------------------------------------
  External References
 *----------------------------------------------------------------------------*/
extern uint32_t __INITIAL_SP;
extern uint32_t __STACK_LIMIT;
#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
extern uint32_t __STACK_SEAL;
#endif
```

CMSIS-Core: system configuration file

The `SystemInit` function is called before __main and is responsible for initializing various system functionality:

- Vector table Offset Register (VTOR)
- Floating-point Unit (FPU)
- Microcontroller Vector Extension (MVE)
- Security Attribution Unit (SAU)

There's also a `SystemCoreClock` variable that may be used to update the main core clock frequency

A user might want to initialize other system components within `SystemInit` too

```
/*----------------------------------------------------------------------------
  System initialization function
 *----------------------------------------------------------------------------*/
void SystemInit (void)
{

#if defined (__VTOR_PRESENT) && (__VTOR_PRESENT == 1U)
  SCB->VTOR = (uint32_t) &(__VECTOR_TABLE[0]);
#endif

#if defined (__FPU_USED) && (__FPU_USED == 1U)
  SCB->CPACR |= ((3U << 10U*2U) |           /* enable CP10 Full Access */
                 (3U << 11U*2U)  );         /* enable CP11 Full Access */
#endif

#ifdef UNALIGNED_SUPPORT_DISABLE
  SCB->CCR |= SCB_CCR_UNALIGN_TRP_Msk;
#endif

#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
  TZ_SAU_Setup();
#endif

  SystemCoreClock = SYSTEM_CLOCK;
}
```

https://arm-software.github.io/CMSIS_5/Core/html/system_c_pg.html

Post Startup Initialization

Extending functions in Arm Compiler

Ideally, system initialization code is executed before entering the main application

- However, the reset handler is not always an appropriate place to, for example, enable interrupts, initialize the MPU, setup PSP, etc.
- The user might also want the `main()` function to execute from unprivileged mode.

The `$Sub$$` and `$Super$$` function wrapper mechanism may be used to fully initialize a system before entering the main application

```
extern void $Super$$main(void);

void $Sub$$main(void)
{
  int_enable();           // enables interrupts
  thread_mode_unpriv();   // change privilege of thread mode
  $Super$$main();         // calls original main()
}
```

Extending functions in GCC

Ideally, system initialization code is executed before entering the main application

- However, the reset handler is not always an appropriate place to, for example, enable interrupts, initialize the MPU, setup PSP, etc.
- The user might also want the `main()` function to execute from unprivileged mode.

A symbol wrapping mechanism in GCC may be used to fully initialize a system before entering the main application:

- Link with `--wrap=<symbol>`
- Add `__wrap_<symbol>` function that calls `__real_<symbol>`

```
__wrap_main ()
  {
    int_enable();              // enables interrupts
    thread_mode_unpriv();      // change privilege of thread mode
  __real_main;
}
```

Change Thread mode to unprivileged

Simple applications without an embedded OS will typically not require Thread mode to be unprivileged

- The whole application is privileged and uses MSP

Applications that use an embedded OS will typically set Thread mode to unprivileged access level

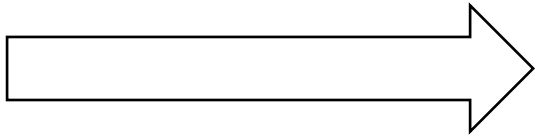
- **`CONTROL.SPSEL = 1 and CONTROL.nPRIV = 1`**

Software running at unprivileged level cannot switch itself back to privileged access level

- Untrusted applications run at unprivileged access level
- This is essential in order to provide a basic security usage model
- A Supervisor Call (**`SVC`**) instruction may be used to request a privileged operation

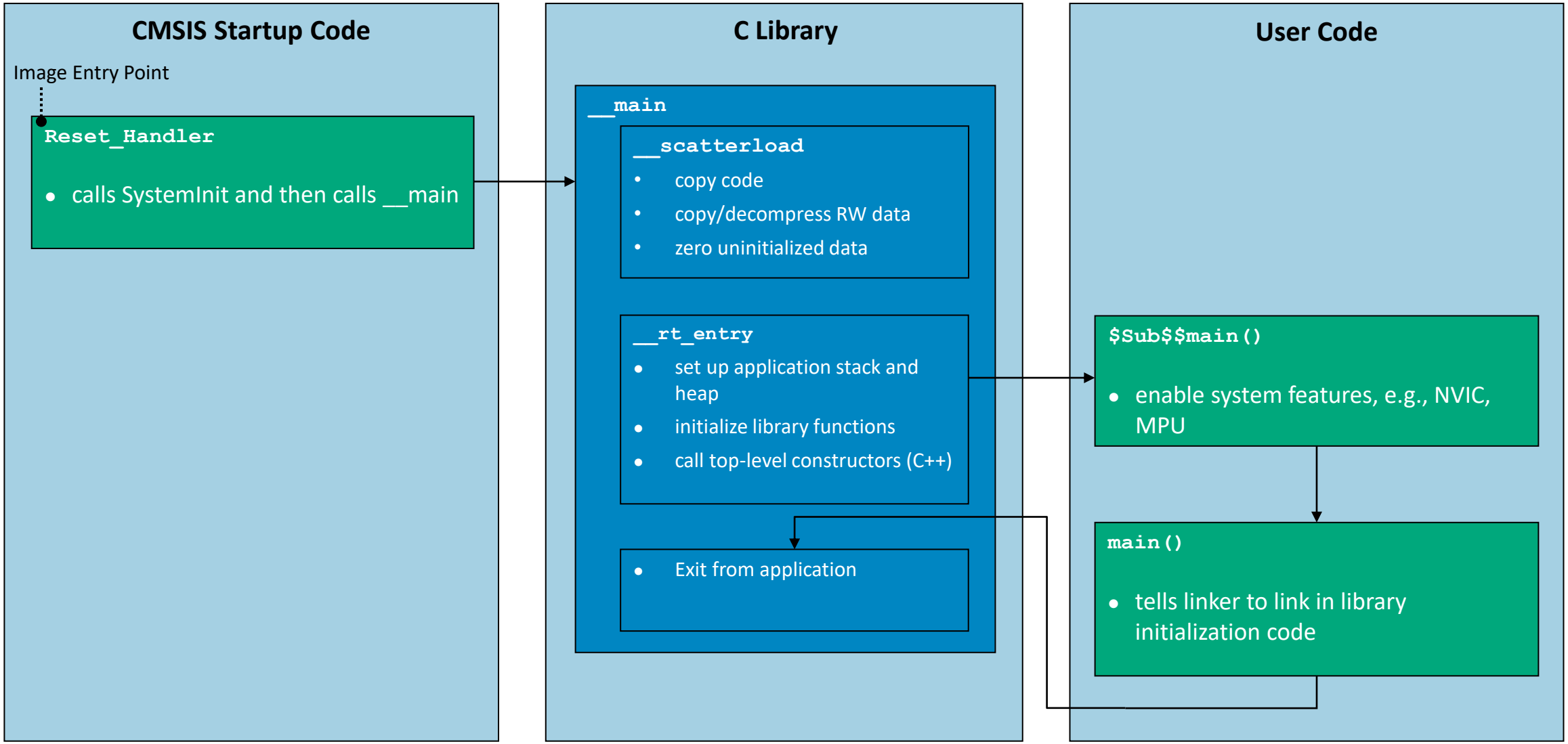
The CONTROL register can be accessed using CMSIS-Core functions

```
/* Change Thread mode to unprivileged and use PSP  */
__set_CONTROL(CONTROL_nPRIV_Msk|CONTROL_SPSEL_Msk);
/* Flush and refill pipeline with unprivileged permissions */
__ISB();
```



```
MOVS r0,#3
MSR  CONTROL,r0
ISB
```

Possible initialization sequence (2) – Arm Compiler



Agenda

Default compilation tool behavior

Writing startup and initialization code

- Initial startup and system initialization code using CMSIS-Core
- Post startup initialization

Tailoring the image memory map to a device

- Writing memory description files (scatter files / linker scripts)
- Placing the stack(s) and heap
- Further memory map considerations

Retargeting the C library behavior to work with your system

Building an image ready for booting

Custom memory map

In real systems, you will probably not want to load to your application to execute from address 0x8000

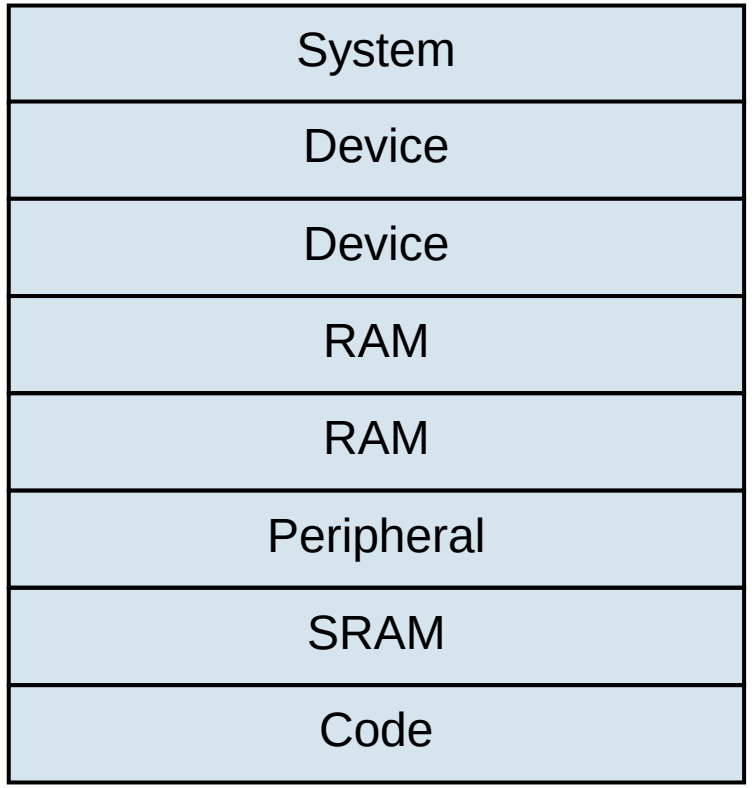
- Most embedded systems have memory devices spread across the memory map

A typical Arm microcontroller may contain the following software elements:

- Code
- Data
- Stack and Heap
- Real-time Operating System (RTOS)
- Peripherals (internal and external)
- System Control Space (SCS)

Although M-profile processors have a fixed memory map, the memory usage is flexible:

- When using a Memory Protection Unit (MPU)
- Multiple SRAM blocks may be placed in SRAM
- Program code can execute from SRAM or RAM



M-profile 4GB System Address Space

CMSIS-Core memory map description templates (1)

CMSIS-Core provides memory map description file templates

- Arm Compiler scatter files
- GNU linker scripts
- IAR linker configuration files

Each file describes basic/essential memory regions/segments for:

- RO/XO code and data
  - CMSIS vector table and startup/system initialization
  - C run-time library initialization
  - Memory initialization code
- RW and ZI data
- Main stack

Scatter files uses C pre-processor directives to

- Configure region base address and size information
- Determine whether regions are required for optional run-time memory like heap and security features

```
/*----------------------------------------------------------------------------
  Scatter Region definition
 *----------------------------------------------------------------------------*/
LR_ROM __RO_BASE __RO_SIZE  {                       ; load region size_region
  ER_ROM __RO_BASE __RO_SIZE  {                     ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }

  RW_RAM __RW_BASE __RW_SIZE  {                     ; RW data
   .ANY (+RW +ZI)
  }

#if __HEAP_SIZE > 0
  ARM_LIB_HEAP  __HEAP_BASE EMPTY  __HEAP_SIZE  {   ; Reserve empty region for heap
  }
#endif

  ARM_LIB_STACK __STACK_TOP EMPTY -__STACK_SIZE {   ; Reserve empty region for stack
  }

#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
  STACKSEAL +0 EMPTY __STACKSEAL_SIZE {             ; Reserve empty region for stack seal immediately after stack
  }
#endif
}

#if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
LR_CMSE_VENEER __CV_BASE ALIGN 32 __CV_SIZE  {      ; own load/execution region for CMSE Venners
  ER_CMSE_VENEER __CV_BASE __CV_SIZE  {
   *(Veneer$$CMSE)
  }
}
```

Creating or modifying CMSIS-Core memory map description files

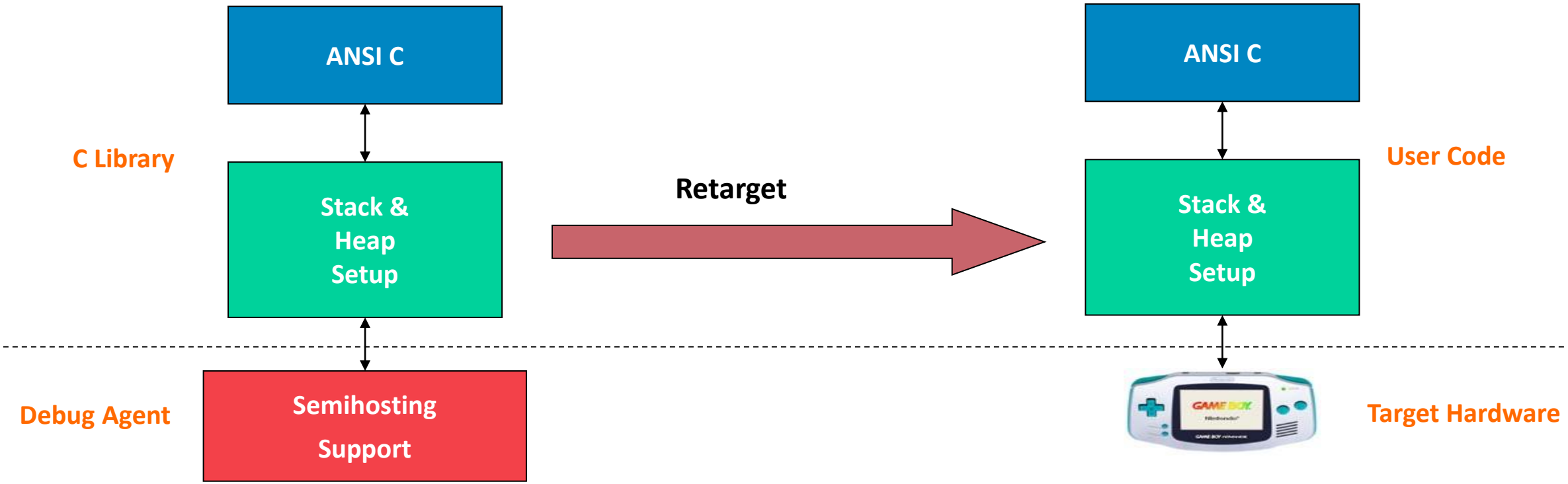
Whether you are a device vendor or software developer, it's important to understand the memory map of the system you are designing or using before creating or modifying memory map description files

- What memories are integrated on the device?
- Are some areas of memory faster/slower than others?
  - Consider placing critical code/routines to run from the faster memory
- Do you need to reserve regions/segments for specific software that will run on the device?
  - Privileged/unprivileged code
  - Different stacks
  - Dynamic heap memory
  - Vector table
  - Startup/system initialization code
- What toolchains do you need to support?
  - Who's going to be developing software for the device and what tools do they have access to?
- Consider whether you need to create memory map description files for different types of images, e.g., secure and non-secure images?

Placing the stack and heap

Run-time memory management

How do we set up the stack and the heap to suit our target memory?

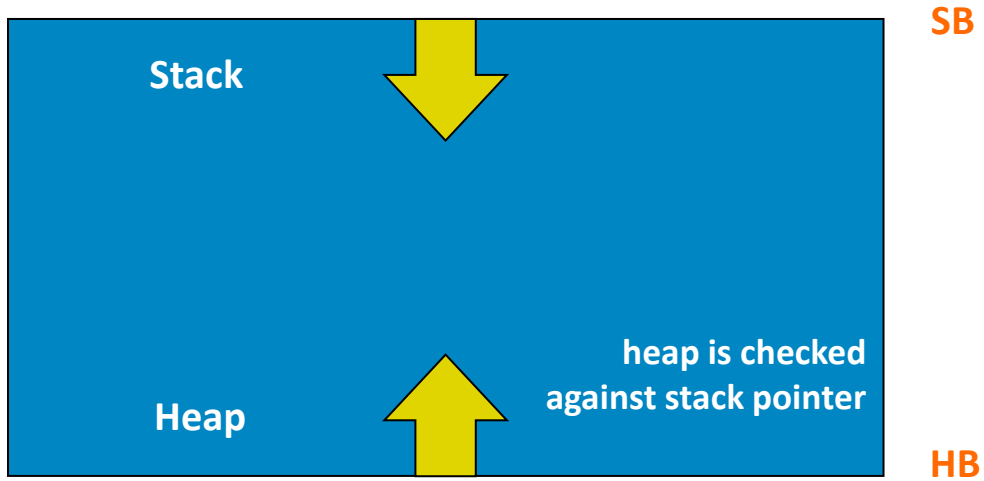


The C library runtime memory model must be retargeted

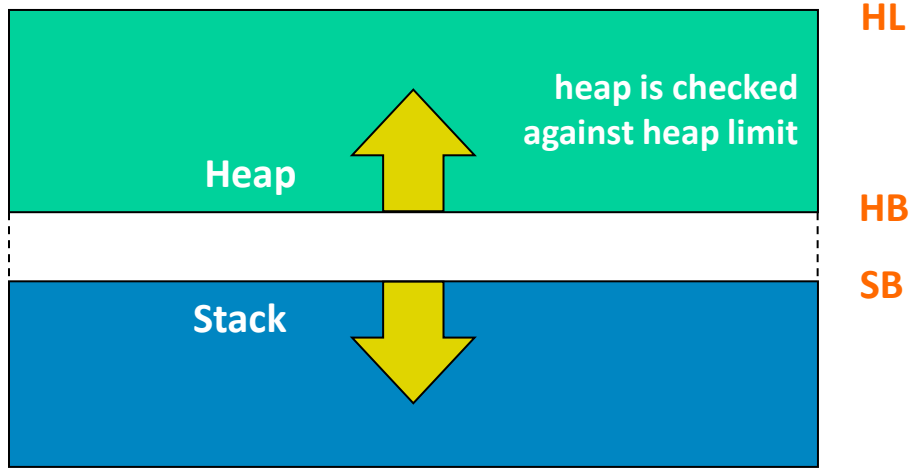
- There are different ways to achieve this …

Run-time memory models

You must decide whether to place your stack and heap in a single region of memory (one-region model) or in separate regions (two-region model)



One region model



Two region model

- **One region model is the default**
- **To implement a two-region model, import `__use_two_region_memory`**
- **The run-time model dictates the location of the Main Stack Pointer (MSP)**
  - A Cortex-M system may use two stacks, more later …

Stack and heap setup

There are different mechanisms for setting up the stack (MSP) and heap

- CMSIS-CORE supports Arm Compiler, GCC and IAR

In Arm Compiler 6 the recommended approach for reserving memory for the stack and dynamic memory is to use a scatter file

- Define:
  - **`ARM_LIB_STACK`** and **`ARM_LIB_HEAP`** execution regions for two region model, or
  - **`ARM_LIB_STACKHEAP`** execution region for one region model
- Import the appropriate symbols in the startup file

The stack pointer can be left to the same value it had on application entry

- Must initially be aligned to a multiple of eight bytes according to the Procedure Call Standard for the Arm Architecture (AAPCS)

Process Stack Pointer (PSP) setup

Multiple stack regions are required for systems using an embedded OS

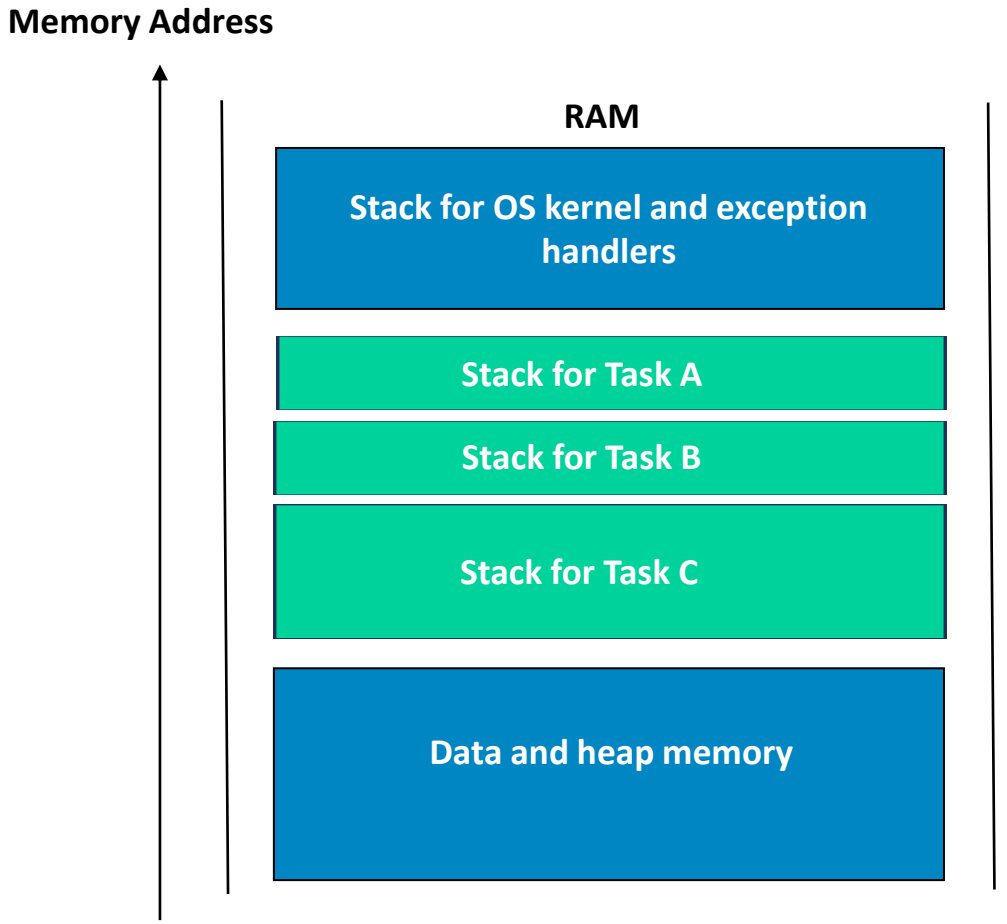
- MSP for OS kernel and exceptions
- PSP for application tasks

The OS kernel must keep track of the stack pointer values for each task during context switching

CMSIS functions are available for accessing the CONTROL register and PSP

- Example:

```
__set_PSP(0x20002000UL);
```



Further Memory Map Considerations

Accessing memory-mapped registers

In M-profile systems, peripherals are memory-mapped

- CMSIS-Core headers are the recommended way to access a device's peripheral registers
  - Peripheral registers are accessed via pointers so there's no need for utilising any special compiler features

There are two main CMSIS-Core headers for accessing peripherals:

- A Processor Core header file, e.g., core_cm3.h provides
  - Pre-processor macros/pointers/access structures for reading/writing core peripherals like the MPU, SysTick, etc.
  - Helper functions for easily programming registers, e.g., `SysTick_Config()`
  - Include other headers for specific/more complex processor core peripherals, e.g., `pmu_v8.h`
- The Device Header file (device.h), e.g., `LPC1768.h`, provides access to other device-specific registers

C compilers provide alternative methods for quickly accessing an address on the target:

- A special Arm Compiler-specific variable attribute. To create a variable that is placed automatically at this address:
  ```
  __attribute__((section(".ARM.__at_0x40000000"))) volatile unsigned int TIMER_0;
  ```
- Place data in a dedicated section/region in a scatter file or linker script

Ordering objects in a scatter file – Arm Compiler

You might need to override the standard linker placement rules in order to place certain code and data at a specific address

Use the +FIRST and +LAST directives to place individual objects first and last in an execution region

- For example: to place the vector table at the beginning of a region

```
LOAD_ROM 0x0000 0x4000
{
  EXEC_ROM 0x0000 0x4000
  {
    startup_<device>.o (RESET, +FIRST)
    file1.o (+RO)
    file2.o (+RO)
  }
  :
}
```

Root regions – Arm Compiler (1)

A root region is an execution region whose load address is equal to its execution address

- Each scatter-loading description file must have at least one root region

Some C library code (e.g., `__main.o`, etc.) and linker-generated tables (e.g, `Region$$Table`) must be placed in a root region, otherwise the linker will report an error message, for example:

```
Error: L6202E: Section Region$$Table cannot be assigned to a non-root region.
```

Forward-compatible way to specify these is using `*(InRoot$$Sections)`

- If *(+RO) is located in a root region, the above will be located there automatically

The application entry point must also lie in a root region

```
Error: L6203E: Entry point (0x00000400) lies within non-root region
```

Root regions – Arm Compiler (2)

```
LOAD_ROM 0x0000 0x4000          ; start address and max length
{
    EXEC_ROM 0x0000 0x2000      ; root (load = exec address)
    {
        * (InRoot$$Sections)
    }

    RAM 0x20000000 0x1000
    {
        .ANY (+RO)              ; All other RO areas
        .ANY (+RW,+ZI)          ; program variables
    }
}
```

Must be in a root region →

outside root region →

A root region is an execution region whose load address is equal to its execution address

MPU initialization

Before using the MPU, it is important to have knowledge of:

- Different memory regions that exist in the device
- Which regions the program or application tasks need to (and are allowed to) access

Systems without an embedded OS will typically use a static configuration

Systems with an embedded OS will typically offer a dynamic configuration

- The memory map is changed on each context switch
- A static configuration may still be preferred

There is no need to setup memory region for Private Peripheral Bus (PPB) address ranges (including System Control Space, SCS) and the Vector table

- Accesses to PPB (including MPU, NVIC, SysTick, ITM) are always allowed in privileged state, and vector fetches are always permitted by the MPU

HardFault and MemManage (Armv8-M.Mainline only) fault handlers must be defined

- When the MPU has been initialized the MemManage exception can be enabled by setting the MEMFAULTENA bit in the System Handler and Control State Register
  - `SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk; // Set bit 16`

Agenda

Default compilation tool behavior

Writing startup and initialization code

- Initial startup and system initialization code using CMSIS-Core
- Post startup initialization

Tailoring the image memory map to a device

- Writing memory description files (scatter files / linker scripts)
- Placing the stack(s) and heap
- Further memory map considerations

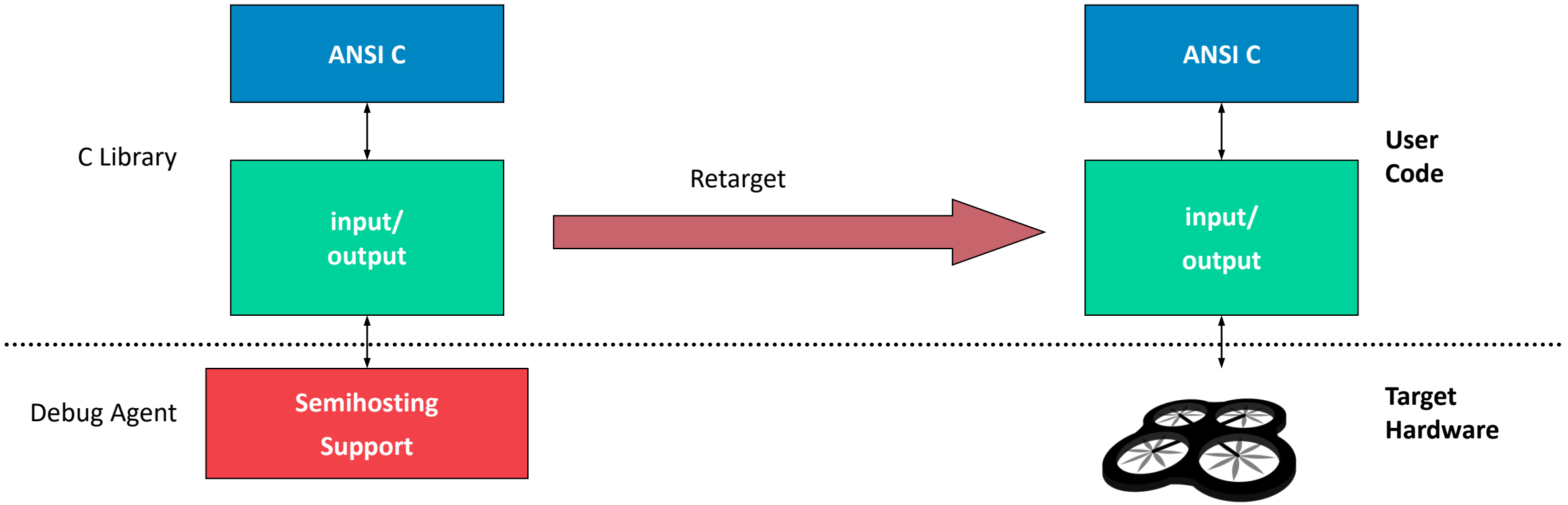
Retargeting the C library behavior to work with your system

Building an image ready for booting

Retargeting the C library (1)

You can replace the C library's device driver level functionality with an implementation that is tailored to your target hardware

- For example: `printf()` should go to LCD screen, not debugger console



Retargeting the C library (2)

To ‘retarget’ the C library, simply replace those C library functions which use semihosting with your own implementations, to suit your system

For example, the `printf()` family of functions (except `sprintf()`) all ultimately call `fputc()`, which uses semihosting by default

Instead of reimplementing `printf()` a user can just replace `fputc()`, for example:

```
extern void sendchar(char *ch);

int fputc(int ch, FILE *f)
{   /* e.g. write a character to an LCD */
    char tempch = ch;
    sendchar(&tempch);
    return ch;
}
```

How can you be sure that *no* semihosting-using functions will be linked in from the C library?…

Avoiding C library semihosting

To ensure that no functions which use semihosting are linked in from the C library, define the ‘guard’ symbol `__use_no_semihosting`

```
asm(" .global  __use_no_semihosting\n");
```

If there are still semihosting functions being linked in, the linker will report:

```
Error: L6915E: Library reports error: __use_no_semihosting was requested but
<function> was referenced.
```

To fix this provide your own implementations of these functions

The Arm Compiler C and C++ Libraries documentation provides a list of C library functions that use semihosting

Note: The linker will NOT report any functions in the user’s own application code that use semihosting or SVC calls

Agenda

Default compilation tool behavior

Writing startup and initialization code

- Initial startup and system initialization code using CMSIS-Core
- Post startup initialization

Tailoring the image memory map to a device

- Writing memory description files (scatter files / linker scripts)
- Placing the stack(s) and heap
- Further memory map considerations

Retargeting the C library behavior to work with your system

Building an image ready for booting

Debugging ROM images

The linker produces ELF/DWARF images, suitable for loading into a debugger

- Build with debug tables (**`-g`** or **`--debug`**) to debug at C source code level

To convert an ELF image into binary format, use the `fromelf` utility, for example:

```
fromelf --bin -o <output> image.axf
```

This generates binary files (one per load region), suitable for blowing onto ROM, downloading into Flash or EPROM-Emulator etc.

Other 'ROMable' formats can also be generated by fromelf, e.g.

- Motorola 32 bit Hex (**`--m32`**)
- Intel 32 bit Hex (**`--i32`**)
- Byte Oriented Hex (**`--vhx`**)

It is possible to debug ROM images by loading the DWARF debug information from the ELF image

References (1)

CMSIS GitHub repository and documentation:
https://github.com/ARM-software/CMSIS_5

Software standards documentation:
https://developer.arm.com/architectures/system-architectures/software-standards/abi

- Semihosting API
- AAPCS

DWARF debug specification:
http://dwarfstd.org/

References (2)

Arm Compiler Documentation:

- Arm Compiler User Guide
  - Embedded Software Development
  - What is Semihosting?
  - Scatter-loading Features
  - Scatter file Syntax
- Arm C and C++ Libraries and Floating-Point Support User Guide
  - The Arm C and C++ libraries
    - C and C++ library naming conventions
    - Redefining low-level library functions to enable direct use of high-level library functions in the C library

arm

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה