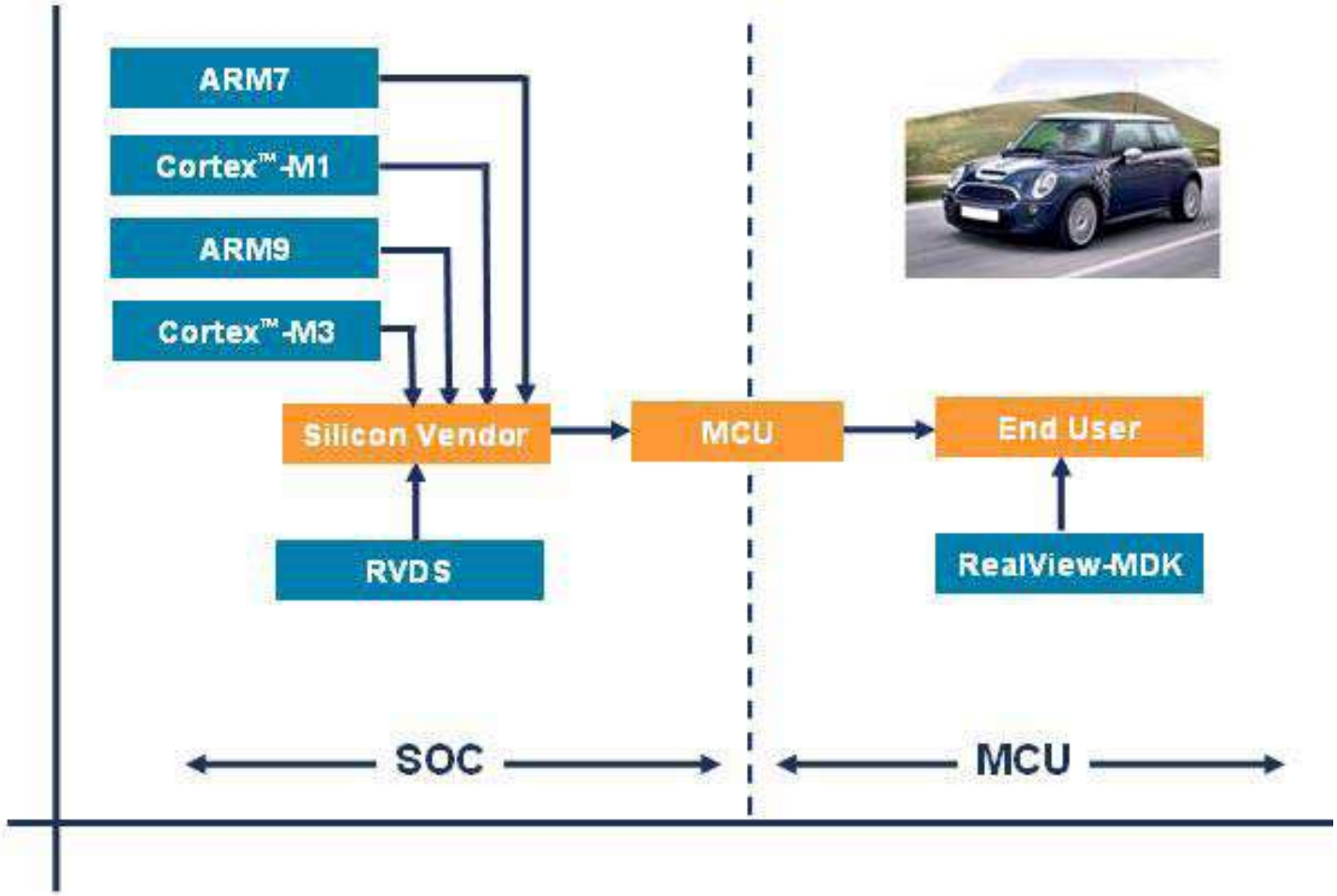
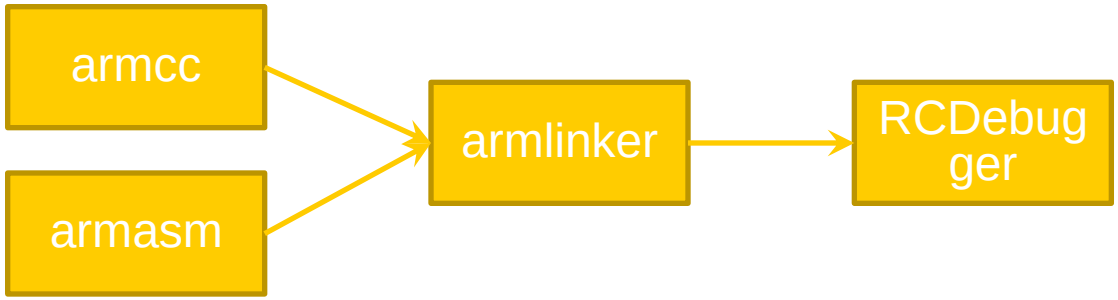
Embedded Firmware Development with ARM Processors

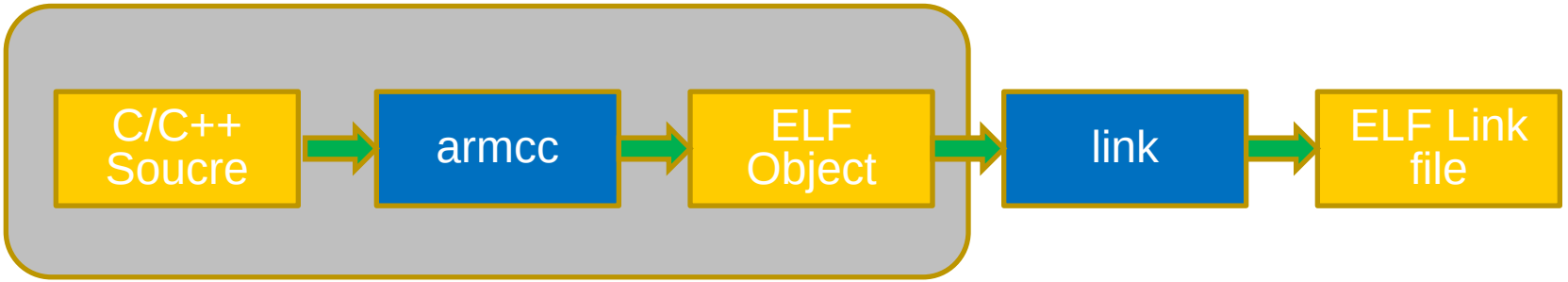
HW/SW Eco-System



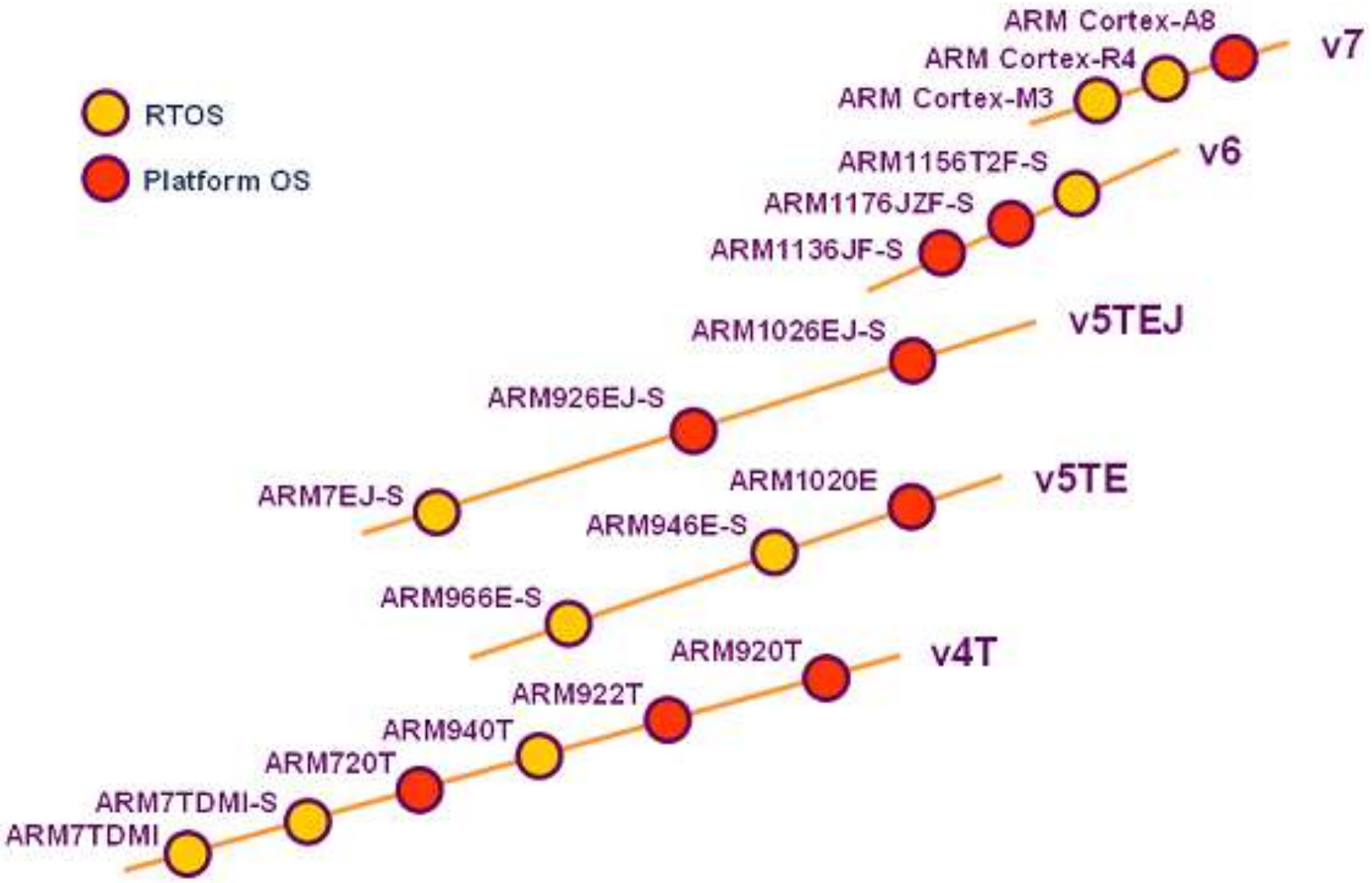
Firmware Compile and Link

- **RealView Tool**
  - SDT->ADS1.2->RVDS2.2->RVDS3.1->RVDS4.0
  - RealView compiler(RVCT)/RealView assembler (armasm), RealView linker (armlinker), and RealView debugger (RVDebugger)

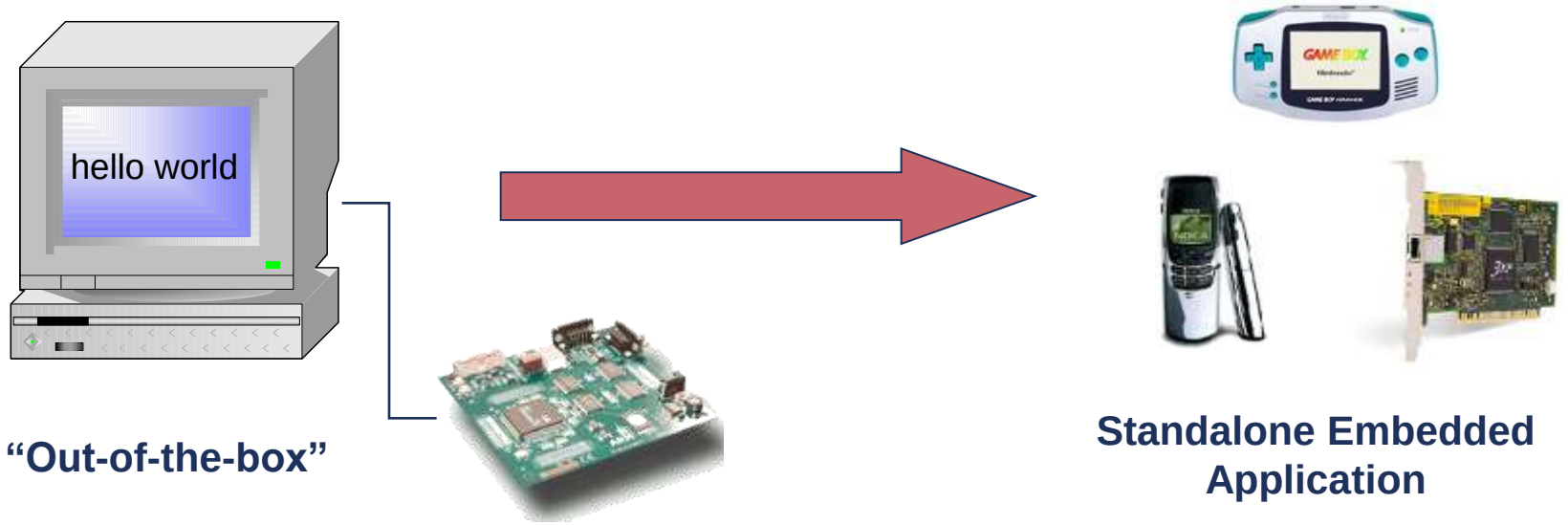




ARM Processor Road Map

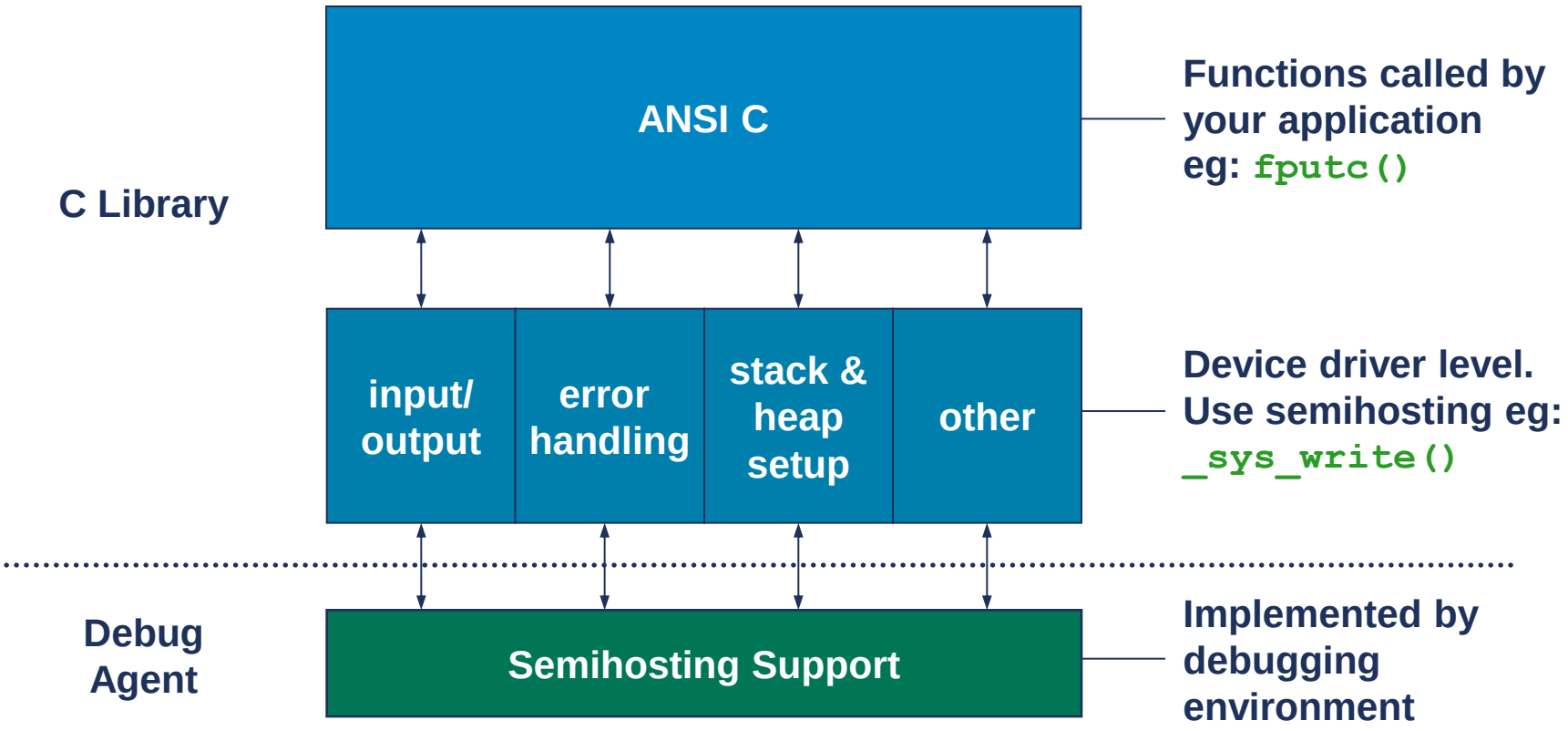


Embedded Development Process



- **In the process of moving from an “out-of-the-box” build to a standalone embedded application, several issues need to be considered:**
  - C library use of hardware
  - Target memory map
  - application startup

Default C Library



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

Default Memory Map

- **By default code is linked to load and execute at 0x8000**
- **The heap is placed directly above the data region**
- **The stack base location is read from the debugging environment by C library startup code**
  - **RVISS** => from configuration file (peripherals.ami)

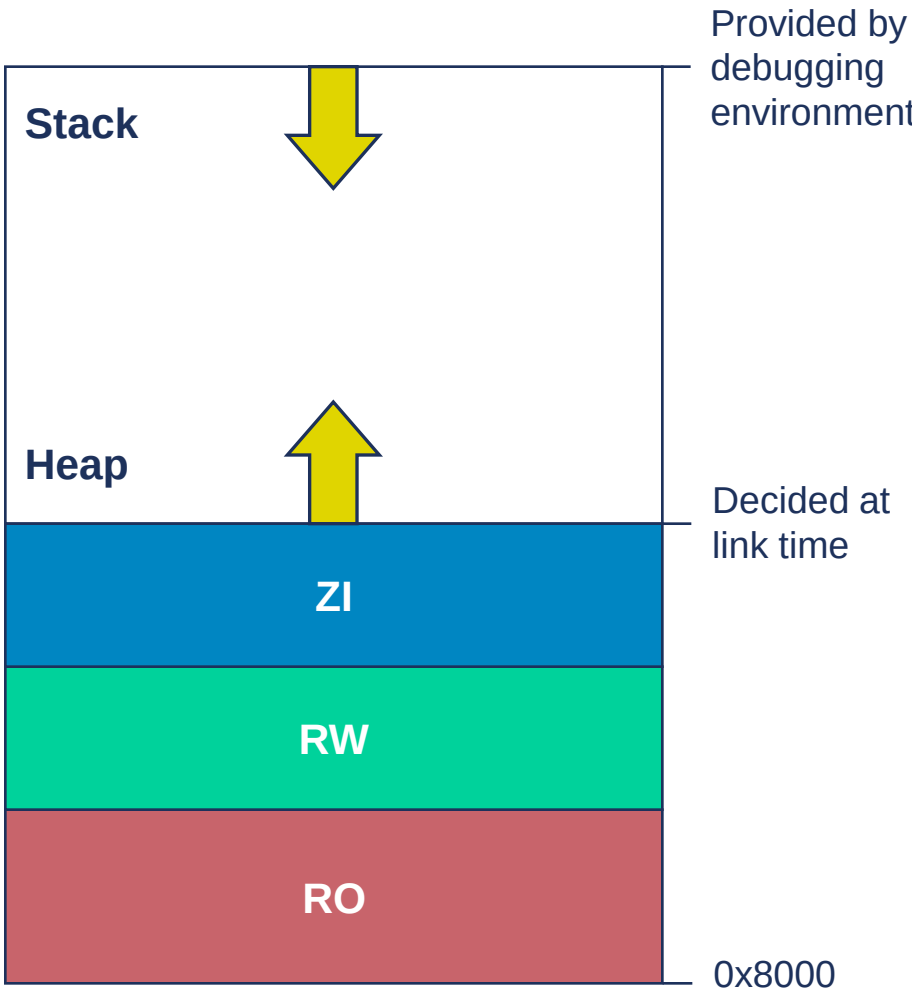
    default = 0x08000000
  - **ISSM** => from setting on configuration dialog in RVD “Connection properties”

    default = 0x10000000 or 0x22000000
  - **RVI and Multi-ICE** => from debugger internal variable $top_of_memory

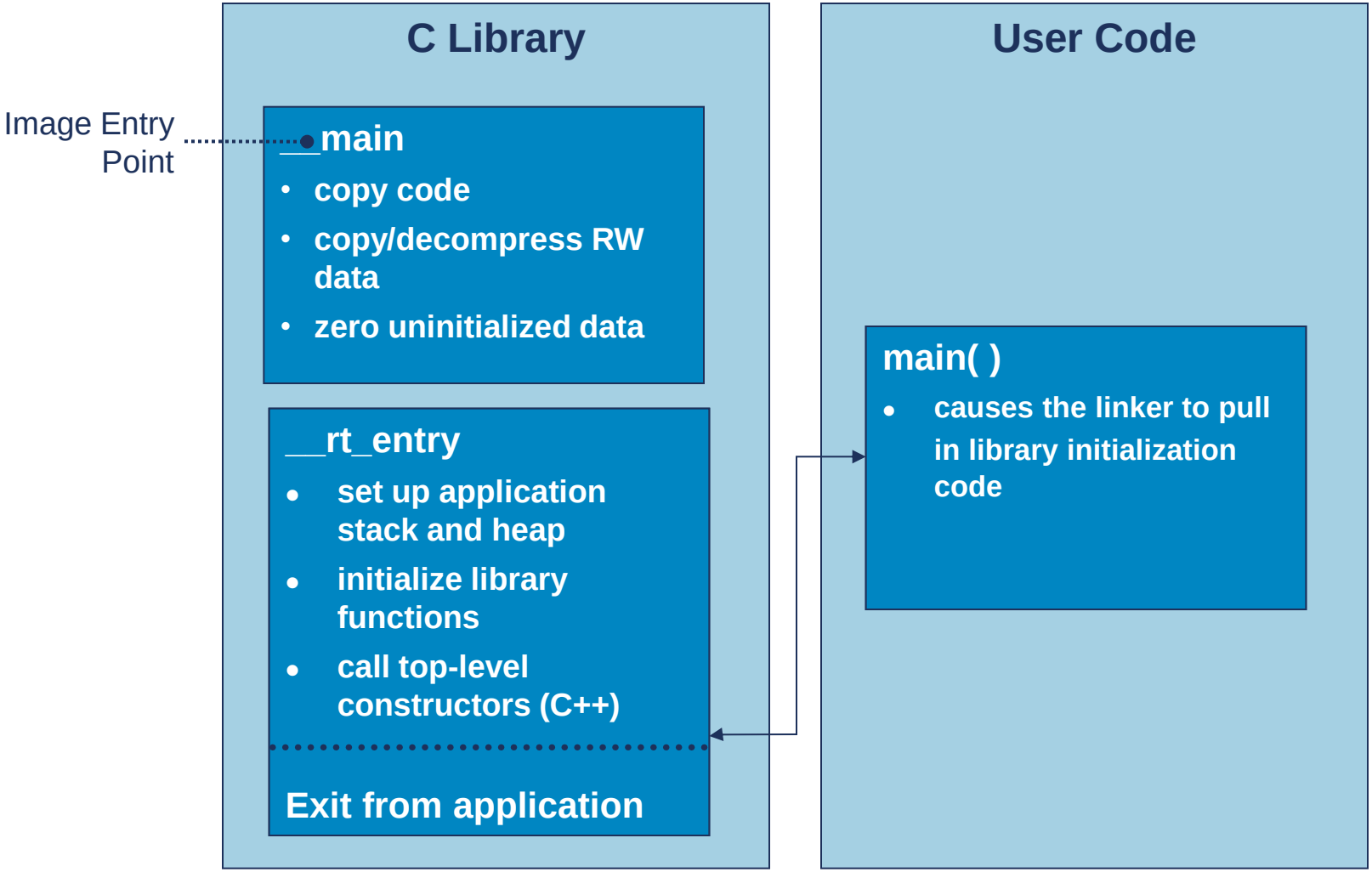
    default = 0x80000

    Changed in “Connection properties” in RVD:

    Advanced_information\Default\ARM_Config



Application Startup



Application Startup

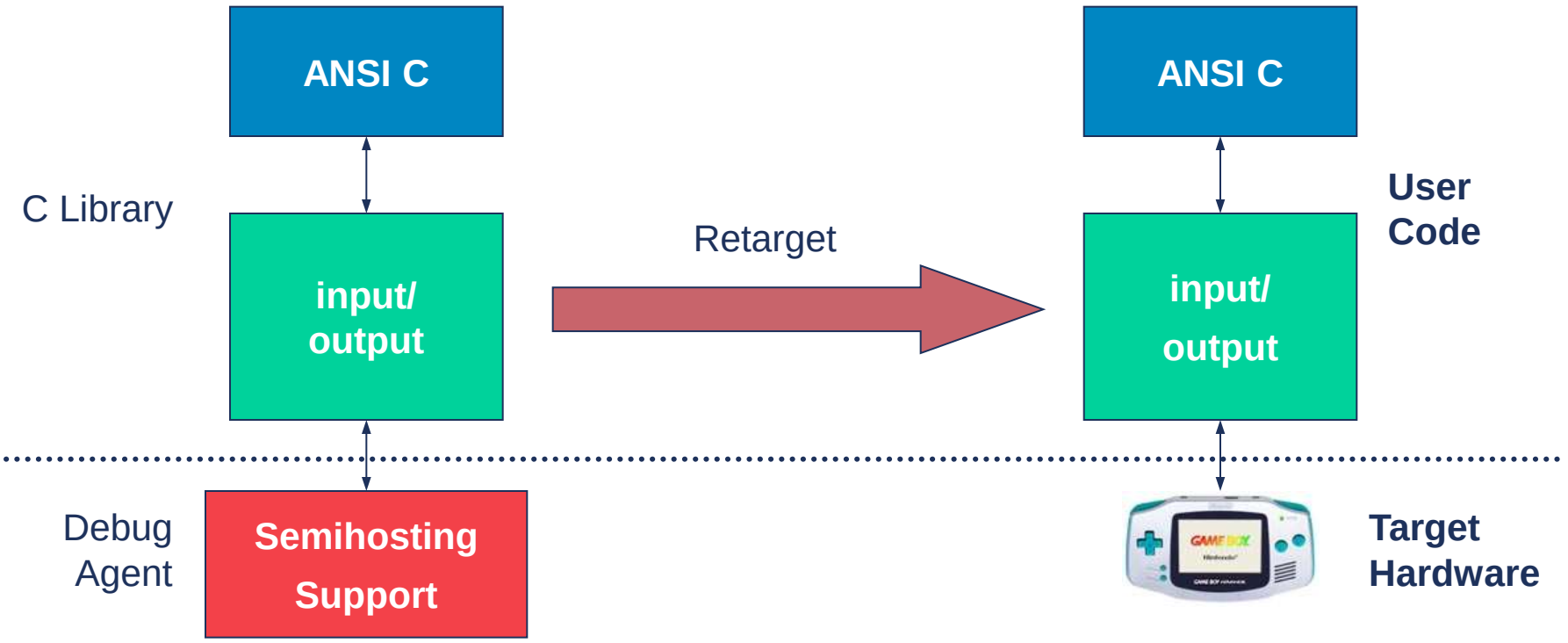
At a high level, __main is responsible for setting up memory, whereas __rt_entry is responsible for setting up the run-time environment. Together, they ensure that the system is in a known good state when the user application begins executing.

In an out-of-the-box build, an application will start executing at the function __main which is provided as part of the C library. This will carry out code and data copying, and zeroing of ZI data. (Don't dwell on this bit too much, as we will cover this in the scatterloading section). __main then branches to __rt_entry (run-time entry). This sets up the application stack and heap (don't mention __user_initial_stackheap until later), initializes library functions and static data, and calls any top-level C++ constructors.

__rt_entry then BL's to main(), the entry to the user's application. Once this application has finished executing, __rt_entry shuts down cleanly and hands control back to the debugger.

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 (1)

By default the C library makes use of semihosting to provide device driver level functionality. In a real system, you will have real target hardware that you make use of.

For example, you might have a peripheral I/O device such as an LCD screen, in which case you would want to override the default library I/O implementation - which writes to the debugger console - with one that actually outputs to the LCD. Fortunately, RVDS(RVCT) is designed so that this can be done easily

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()`
  - The default implementation of `fputc()` uses semihosting
  - Replace this with:

```
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;
}
```

- **See `retarget.c` in the "emb_sw_dev" example directory of your tools installation for further examples of retargeting**
- **How can you be sure that *no* semihosting-using functions will be linked in from the C library?……**

Retargeting the C Library (2)

This example assumes that sendchar() provides device-driver level I/O to the LCD. In this way the example fits nicely with the diagram on the previous slide (Note: The use of '`tempch`' ensures this function is endian-independent).

We retarget fputc instead of __sys_write() in this case. By default printf() calls fputc() for each character in the string, and fputc() essentially fills a buffer. printf() then calls __sys_write() once the buffer is full. By retargeting fputc(), we override this buffered output, ie: printf() calls fputc() which prints out the UART directly - __sys_write() is not called.

Take care if you want to collect retargeted functions together into a library. Be sure you understand the library searching order!

Avoiding C library Semihosting

- To ensure that no functions which use semihosting are linked in from the C library, import the 'guard' symbol `__use_no_semihosting`

```
#pragma import(__use_no_semihosting)
```

- 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.
```

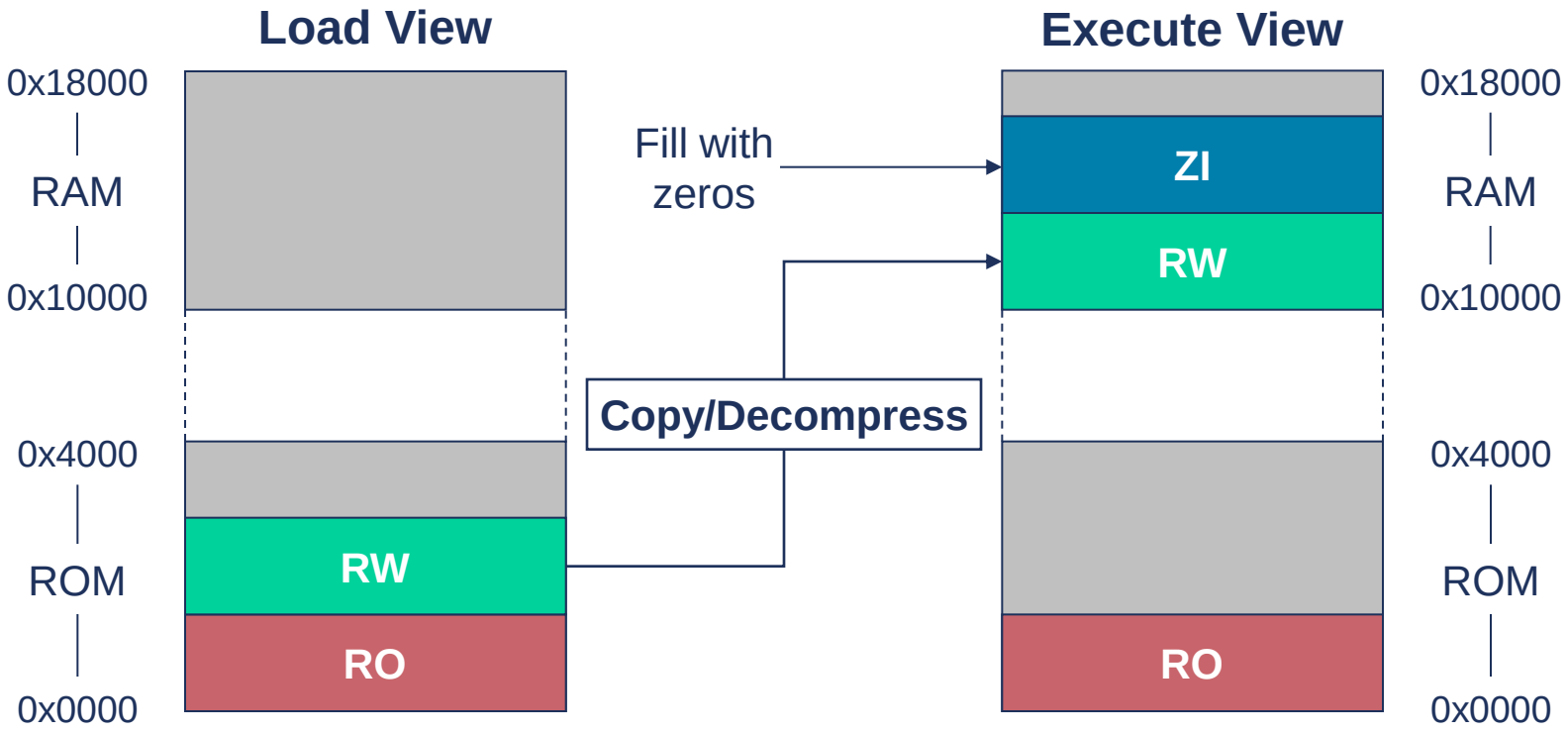
- To fix this provide your own implementations of these functions
- RVCT Compiler and Libraries Guide, Table 5-3 give a full list of semihosting C library functions
- Note: The linker will NOT report any functions in the user's own application code that use semihosting or SVC calls

Introduction to Scatterloading

- **In a real application, you will not want to load and execute your code at address 0x8000**
  - Most embedded systems have memory devices spread across the memory map
  - Scatterloading provides a way of placing your code and data into the appropriate locations in memory
- **Scatterloading defines two types of memory region**
  - Load Regions - contain application code & data at reset/load time (typically ROM)
  - Execution Regions - contain code and data whilst the application is executing
    - One or more execution regions will be created from each load region during application startup
- **The details of the memory map used by a scatterloaded application are contained in a description file which is passed as a parameter to armlink**

  e.g. `armlink program.o --scatter scatter.scat -o program.axf`

Scatterloading (Simple Example)



- **RO code and data stays in ROM**
- **C library initialization code (in `__main`) will :**
  - Copy/decompress RW data from ROM to RAM
  - Initialize the ZI data in RAM to zero

Scatterloading (Simple Example)

This slide shows a simple example of how code and data can be placed using scatterloading.

In this example, we have a simple embedded system with one ROM device and one RAM device. In our system (for the purposes of this example), we want to load all our code and data directly into the ROM device (see load view).

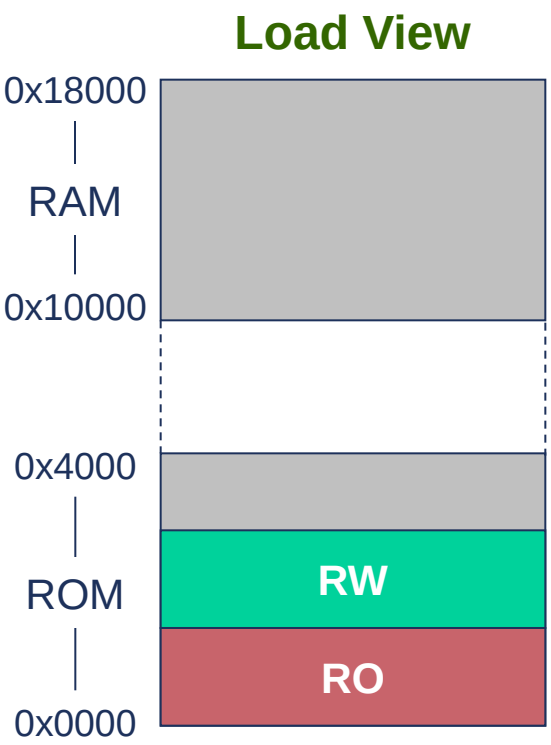
At run-time we will want a different memory map. We want our RO code and data to remain in ROM, but we want to move our RW data into RAM. Additionally, we want to initialize a region of ZI data directly following our RW region in RAM. (execute view).

During startup, code in __main carries out the copying and zeroing in order to move from our load view to our execute view.

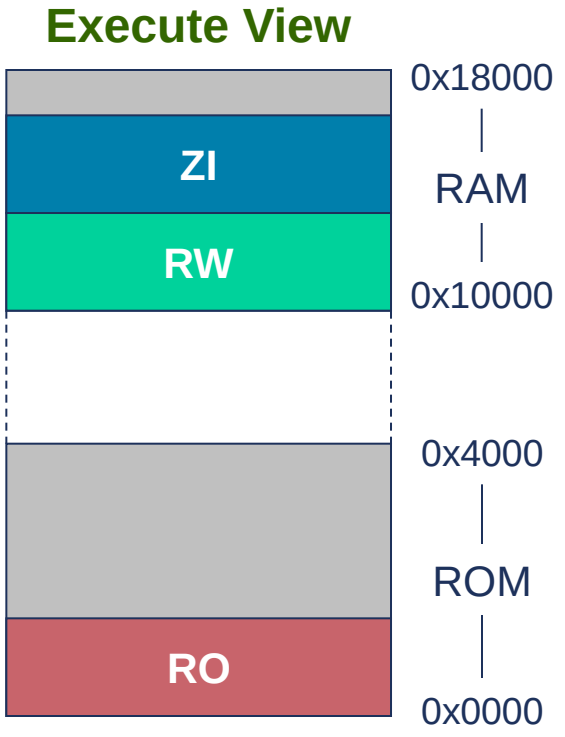
Using scatterloading terminology, the example here has one load region containing all code and data, starting at address zero. From this load region we create two execution regions. One contains all RO code and data, which executes at the same address at which it is loaded. We also have an execution region at address 0x10000, which contains all of our RW and ZI data.

Precise control of this process is achieved by describing our desired memory map in a textual description file, from here on known as a scatter file. Understanding of scatter file syntax is crucial if we want to make the best use of this feature. Let's take a look at the scatter-file for this example…

Scatter Description Files




```
LOAD_ROM 0x0000 0x4000
{
  EXEC_ROM 0x0000 0x4000
  {
       * (+RO)
  }
  RAM 0x10000 0x8000
  {
       * (+RW,+ZI)
  }
}
```



- The wildcard (*) syntax allows for easy grouping of code and data
- Scatter Description files can be pre-processed

Scatter Description Files

In our scatter file we need to give both our load and execution regions a name. We also need to specify a start address. There is an optional length parameter which can follow the address parameter. For our example, the following syntax describes our single load region.

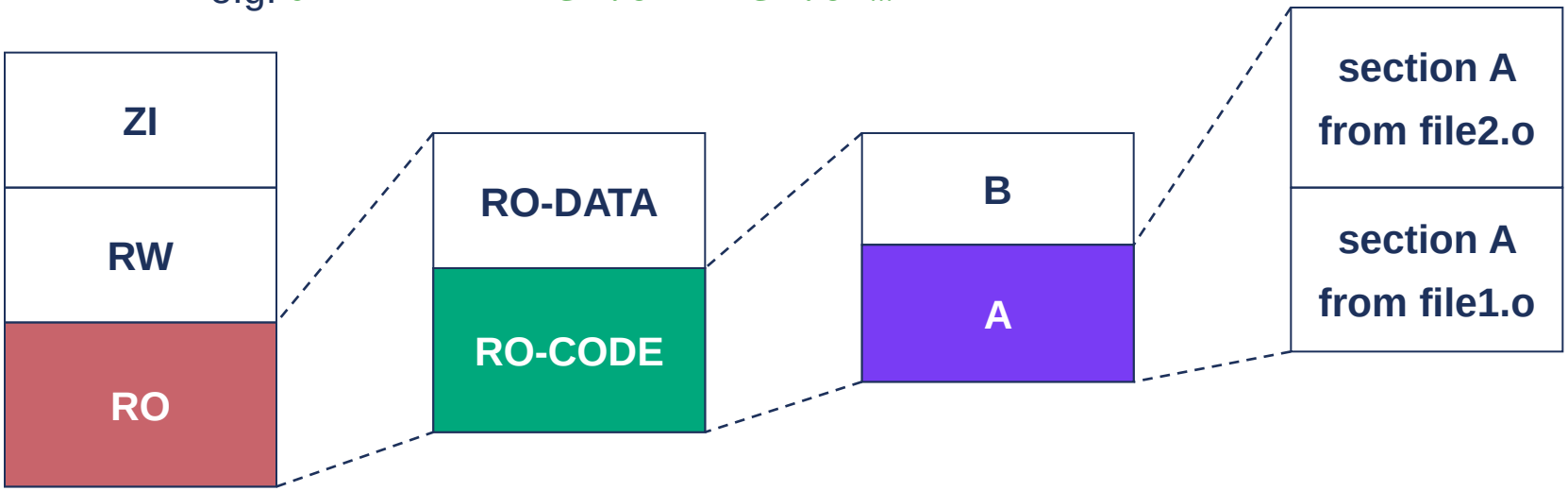
The header syntax for execution regions is very similar to the header syntax for load regions. As you can see from the execution region names, the first is located in ROM, and the second is located in RAM.

You must specify the contents of each execution region with the curly braces. In this example, we make use of the wildcard (*) syntax which collects all sections with a given attribute. In the EXEC_ROM region, we collect all sections with the RO attribute. In the RAM region, we collect all sections with either RW or ZI attributes.

Note that the linker placement rules discussed earlier apply to each execution region individually, so all RW code and data will be located before all ZI data. (As shown in the diagram).

Linker Placement Rules

- **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` …



Linker Placement Rules

The linker observes some basic rules to decide where code and data.

The first level of the hierarchy is attribute. RO precedes RW precedes ZI. Within each attribute code precedes data.

From there, the linker places input sections alphabetically by name. So, in this example, section A is placed before section B.

Finally, for same named, code and data from individual objects is placed in the order the object files are specified on the linker command line. So, in our example, any code in section A from file1.o is placed before any code from file2.o.

In practice, the user is not advised to rely on these rules if precise placement of code and data is needed. Full control of placement of code and data is available through scatterloading (to be discussed later).

Linker Placement Rules

For the most part, the default section name for compiler output (and much of the library code) is ".text". __main.o, gets a section labelled "!!!" (to ensure that it is placed first in a given region).

The linker will uses a different, more complex, sorting algorithm for large thumb execution regions, this is because of the restricted branch range of the Thumb instruction set.

At this point we haven't mentioned execution regions, since we are discussing the "out-of-the-box" experience. All these rules are observed within each execution region. This point needs to be emphasized in the scatterloading section of this module.

In reality, there is a distinction made between "output sections" and "input sections". Output sections are really what is depicted on the left of the diagram. These are ordered by attribute (RO-code, RO-data etc…).

Ordering Objects in a Scatter File

- 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
  {
    vectors.o (Vectors, +FIRST)
    file1.o (+RO)
    file2.o (+RO)
  }
  :
}
```

- The ordering of objects in the scatter file execution region does NOT affect their ordering in the output image
- The linker uses a far more complex sorting algorithm for Thumb regions over 4MB - this minimizes the number of veneers required

Ordering Objects in a Scatter File

You shouldn't rely on linker placement rules if you need precise placement of code and data in your image. Especially if using large thumb regions.

You can use the +FIRST and +LAST directives in the scatter-loading description file to override the standard linker placement rules.

For example, in a real image, we will need to place the vector table at 0x0000 (or 0xFFFF0000 for WinCE). To do this, place the object file containing the vector table - in this case vectors.o - in an execution region at 0x0000 and mark it as +FIRST.

It is important to note that the ordering of objects in an execution region does not affect their ordering in the output image. The rules covered in the previous slide apply normally to all objects except those marked as first or last.

For instance, in the above example, we don't know the ordering of file1.o and file2.o.

Ordering Objects in a Scatter File

linker placement rules still apply, i.e. RO then RW then ZI. Can work around this by splitting execution region into two regions, with the second's execution address specified using the +0 notation.

Ie: This won't work:

```
RAM 0x0
{ a.o (+RO)
  b.o (+RW, +FIRST)
}
```

but this will:

```
RAM 0x0
{ b.o (+RW, +FIRST)
}
RAM2 +0
{ a.o (+RO)
}
```

Root Region Notes

- A root region is an execution region whose load address is equal to its execution address
- Each scatter description 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 e.g:

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

- Forward-compatible way to specify these is using `*(InRoot$$Sections)`
- Note: If `*(+RO)` is located in a root region, the above will be located there automatically
- The main application entry point must also lie in a root region

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

Root Regions

```
LOAD_ROM 0x0000 0x4000              ; start address and length
{
    EXEC_ROM 0x0000 0x4000          ; root (load = exec address)
    {
        * (InRoot$$Sections)
        ; Alternatively use
        ; __main.o (+RO)            ; entry point
        ; __scatter*.o (+RO)        ; copy/zero code
        ; __dc*.o (+RO)             ; decompression code
        ; * (Region$$Table)         ; addresses of regions to copy/zero
    }

    RAM 0x10000 0x8000
    {
        * (+RO)                     ; All other RO areas
        * (+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

Root Regions

To demonstrate what needs to be kept in a scatter file root region. This is a very common pitfall that users come across. Often, users have a * (+RO) in a non-root region

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

Each scatter description must have at least one root region which must contain at least:

- `__main.o` - contains the code that copies code/data
- `Region$$Table, __scatter*.o` and `__dc*.o` - sections which contain the addresses of the code/data to be copied. These sections are generated by the linker so do not have a corresponding object file (so * must be used)
- init_aeabi.o and init_array should also be placed in the same section, these are used for C++ Constructors

If `* (+RO)` is located within a root region, the above are located there automatically.

The main application entry point must also lie in a root region.

==========

Note that we only need to make the effort to include the essentials in the root region because * (+RO) is located outside a root region.

Loading Region Descriptions

- **Load region format:**
  - load-region-description ::= load-region-name
  - base-designator [attribute] [max-size]
  - LBRACE execution-region-description + RBRACE
  - base-designator ::= base-address | (PLUS offset)
    - load-region-name
    - base-designator, base address of objects in this LR
    - +offset, object linking start address is base address plus offset
    - Attribute-list, default ABSOLUTE
      - PI, location independent
      - RELOC, relocation
      - OVERLAY ADS
      - ABSOLUTE, default
    - Max-size, maximum size of this load region

Execution Region Descriptions

- **Exec region format:**
  - execution-region-description ::= execution-region-name base-designator [attribute] [max-size]

    LBRACE input-section-description * RBRACE
  - base-designator ::= base-address | (PLUS offset)
    - exec-region-name
    - base-designator, base address of objects in this ER
    - +offset
    - Attribute-list, default ABSOLUTE
      - PI, location independent
      - RELOC, relocation
      - OVERLAY ADS
      - ABSOLUTE
    - Max-size

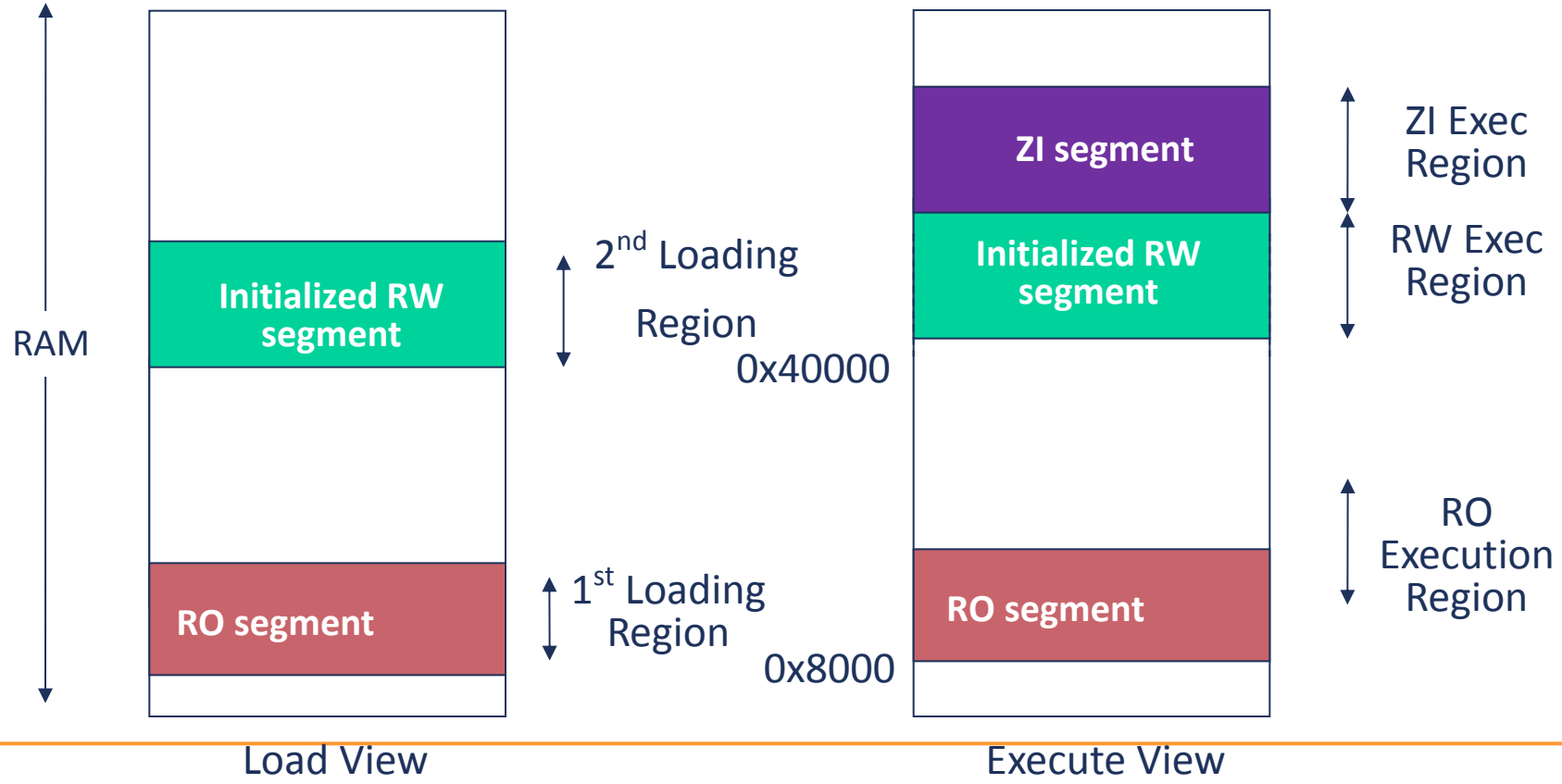
Scatter Example——
Two load regions and three disjoint execution regions

- **Linker command options**

  -split

  -ro-base 0x8000

  -rw-base 0x40000



Scatter Example—Two load regions and three disjoint execution regions

- **Scatter file**

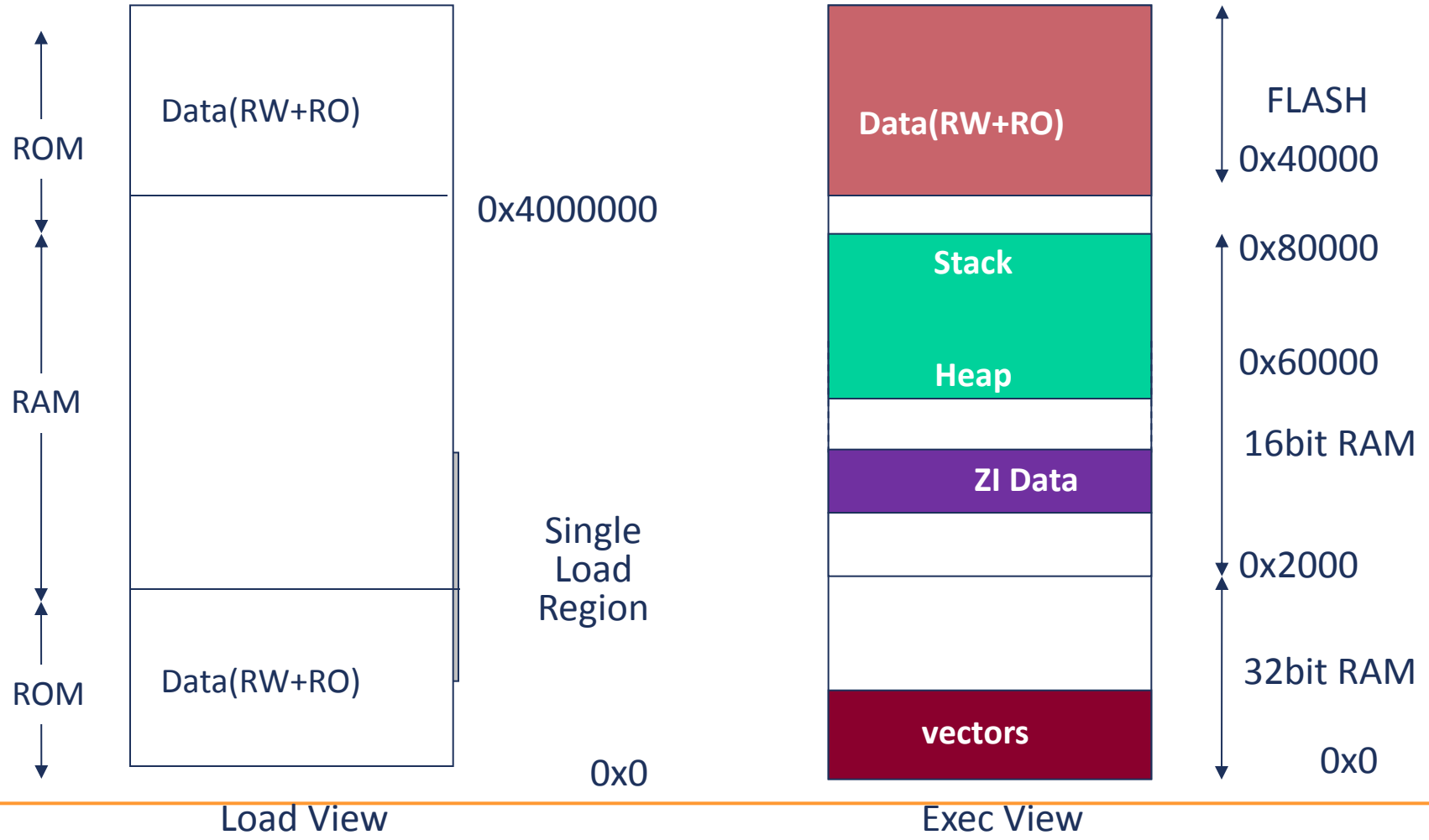
```
LR_1    0x08000              ;Define load region LR_1，start addr 0x8000，in ROM
{
    ER_RO +0                  ;1st execution region ER_RO，start right after the previous exec
    region. Since it is the 1st exec region, its start addr is 0x8000
   {
    *(+RO)                    ;Include all RO segments which are placed consecutively.
    }
}
LR_2    0x40000
{
    ER_RW +0                  ;2nd exec region ER_RW，start addr 0x40000
    {
    *(+RW)                    ;Include all RW segments which are placed consecutively
    }
    ER_ZI +0                  ;3rd exec region ER_ZI，start add 0x40000+sizeof(ER_RW)
    {
    *(+ZI)                    ; Include all ZI segments which are placed consecutively
    }
}
```

Scatter Example—— FIXED Attribute

- **Use FIXED to place data in fixed ROM location**

```
LOAD_ROM 0x0
{
ER_INIT       ;1st exec region ER_INIT，start addr 0x0
{
Init.o(+RO) ;this ER includes init.o which is read only
}
ER_ROM +0   ;2nd exec region ER_ROM, located right after ER_INIT
{
.ANY(+RO)   ;Use .ANY attribute so this ER can contain segments without location specified
}
DATABLOCK 0x7000 FIXED ;3rd exec region, start addr 0x7000
{
Data.o(+RO) ;data.o placed between 0x7000 and 0x8000
}
ER_RAM 0x8000 ;4th exec region ER_RAM, start addr0x8000
{
*(+RW,+ZI)  ;Include RW and ZI data in this ER.
}
}
```

Scatter Example —— A Practical Example



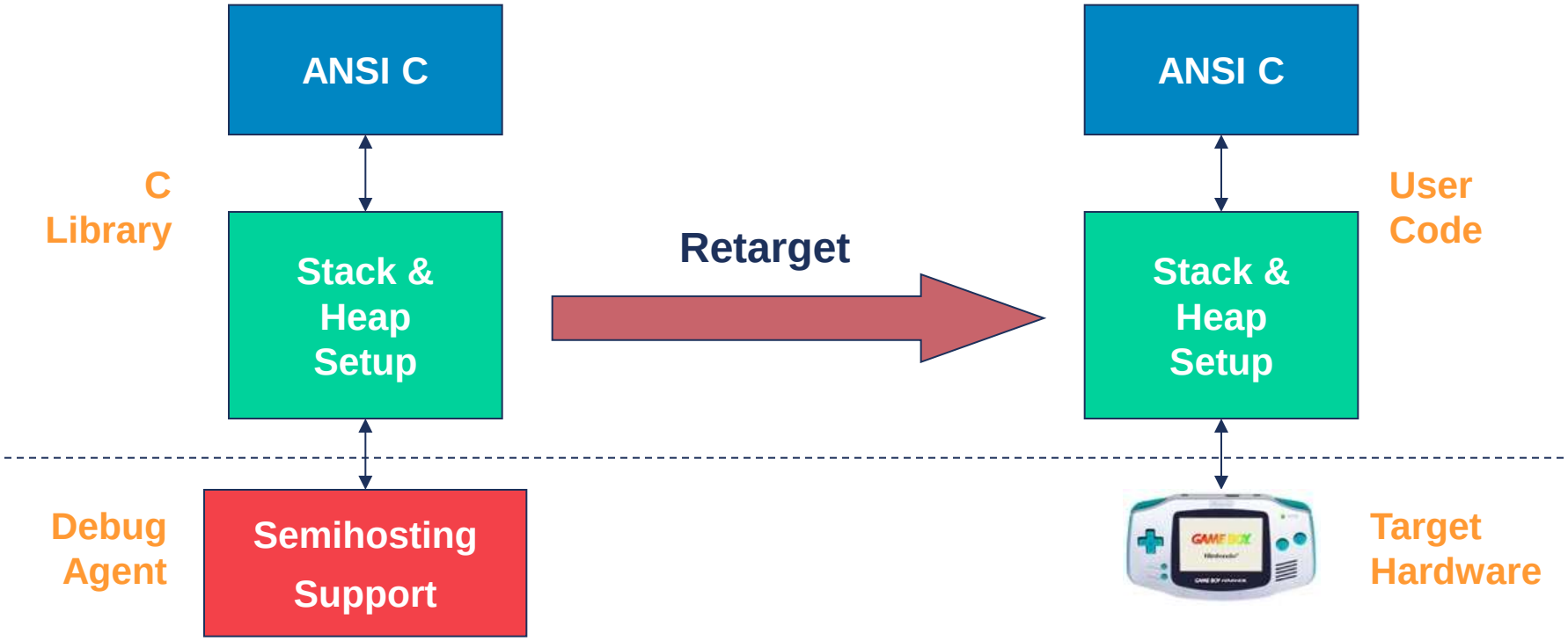
Scatter Example —— A Practical Example

- Scatter file

```
FLASH_MEMORY 0x04000000 0x80000      ;1st load region FLASH
{                                        ; start addre 0x4000000，size 0x80000
   FLASH 0x04000000 0x80000              ;1st exec region FLASH
{                                        ;start addr 0x4000000, size 0x80000
   init.o(Init, +First)                 ;this ER contains all RO code
   *(+RO)                               ;init.o located at the beginning of this ER
}
32bitRAM 0x0000 0x2000       ;2nd exec region is 32bit RAM,
                                ;start addr 0x0，size 0x2000 (within 32bit RAM)
{
   vectors.o(Vect, +First)       ;include vectors.o
}
16bitRAM 0x2000 0x80000      ;3rd exe region, 16bitRAM，
                                 ;start addr 0x2000, size 0x80000
{
   *(+RW, +ZI)                   ;include RW and ZI data
}
}
```

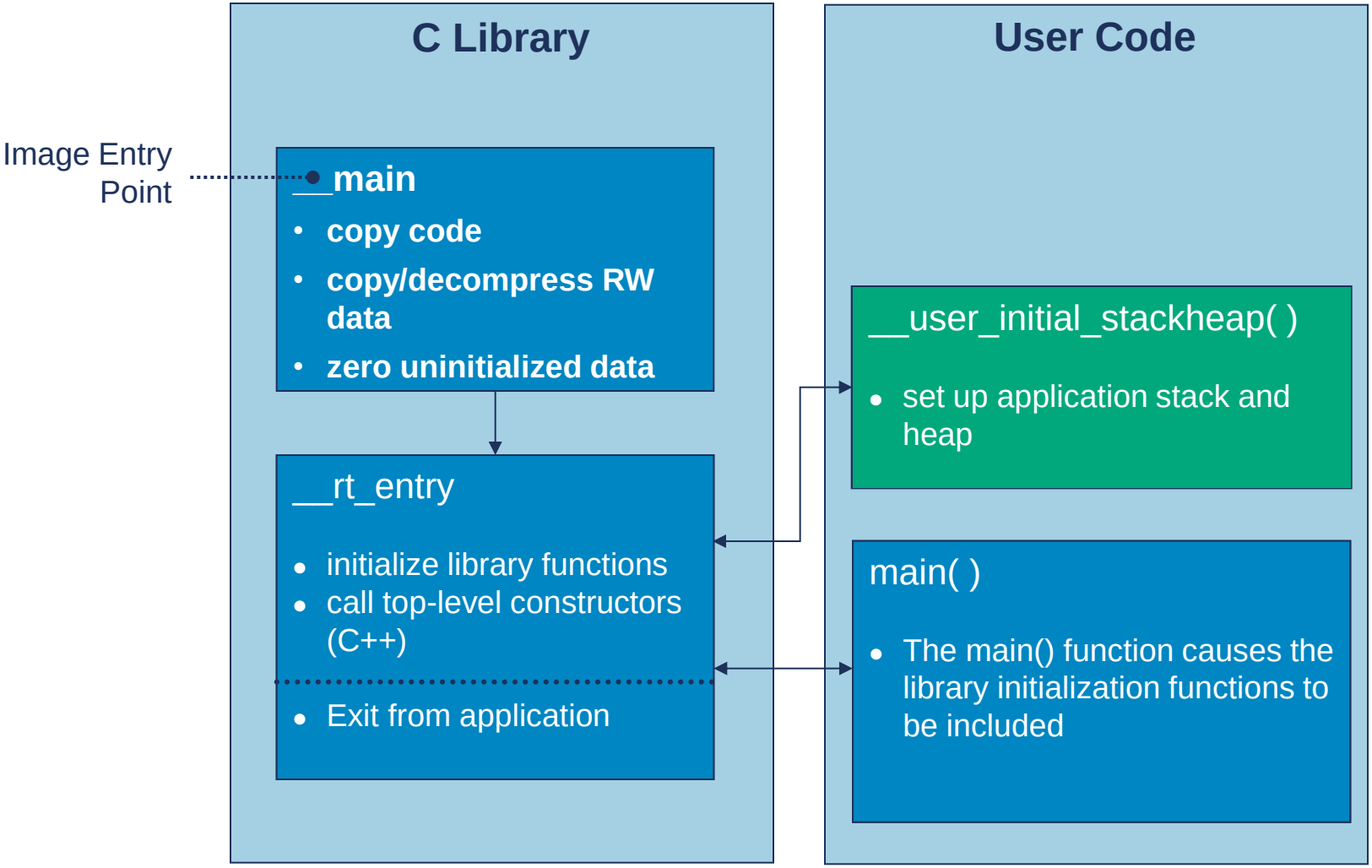
Run-time Memory Management

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



- We have to retarget the C library runtime memory model by re-implementing `__user_initial_stackheap()` or using the scatter file

Stack and Heap Initialization



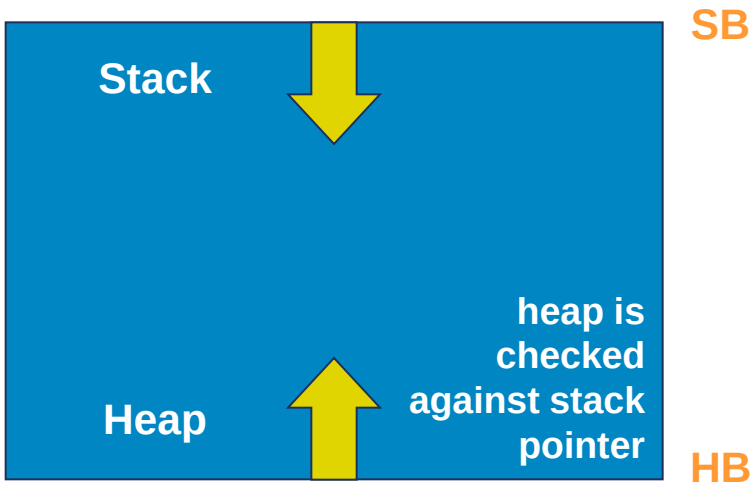
Stack and Heap Initialization

This diagram shows the C library initialization process with a retargeted __user_initial_stackheap(). The rectangle in yellow shows that __user_initial_stackheap() has been added as a new block in this process. It is called from __rt_entry(). The rest of the initialization process is unchanged.

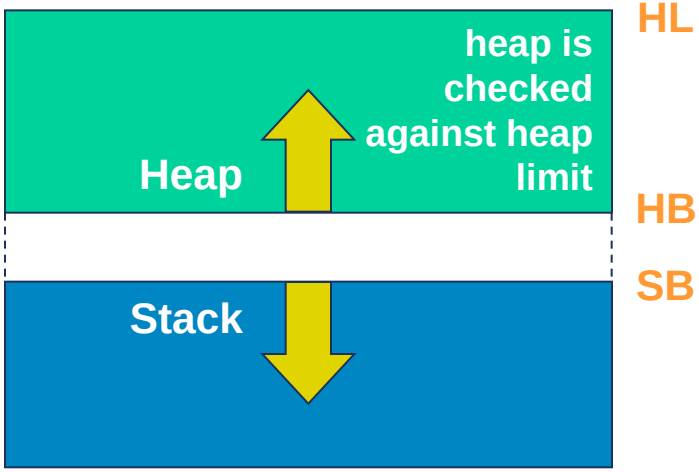
If you specify ARM_LIB_STACKHEAP or ARM_LIB_STACK and ARM_LIB_HEAP in your scatterfile you do not need to provide __user_initial_stackheap().

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`

Run-time Memory Models

RVCT provides two possible run-time memory models. You can have the application stack and heap growing towards each other in the same region of memory, ie: a one region model. Alternatively you can have the stack and heap each in their own region of memory, ie: a two-region model. In the case of a one-region model, the heap is checked against the value of the stack pointer when malloc() is called. In a two-region model, the heap is checked against a separate heap limit.

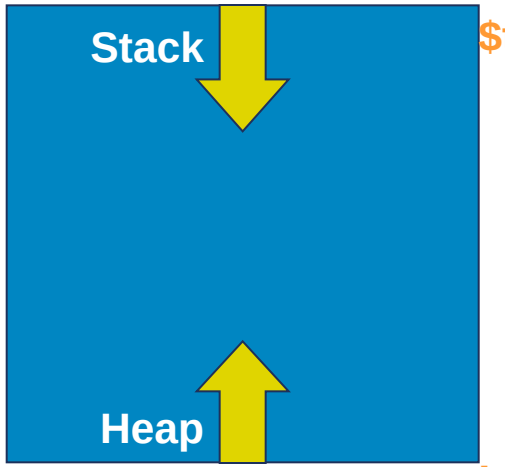
One region is the default. However, your system design might require the stack and heap to be placed in separate regions of memory. For instance you might have a small block of fast RAM in which you want to reserve for the application stack. In order to inform RVCT that you wish to use a two-region model, you must import the symbol __use_two_region_memory. This can be done anywhere in your source.

In both cases, the stack grows unchecked.

__user_initial_stackheap( )

- `__user_initial_stackheap()` is used to configure the location for the Stack and Heap during the C Library Initialization
- There is an implementation already provided in the library for none scatter loaded images and uses semihosting to get the stack base

```
        EXPORT __user_initial_stackheap
        ; pseudo non-scatterloading
        ; __user_initial_stackheap
__user_initial_stackheap
        LDR r0, =Image$$ZI$$Limit ;HB
        LDR r1, =$top_of_memory ;SB
        ; r2 not used (HL)
        ; r3 not used
        MOV pc, lr
```



SB = $top_of_memory

HB = Image$$ZI$$Limit

- If using scatterloading you will need to place your own stack and heap - you will get a linker error if you do not

__user_initial_stackheap( )

This slide is just a note to point out that, if you are scatterloading, you must retarget UISH.

The reason for this is essentially that the default implementation references linker generated symbols that are not valid in a scatterloaded image. (refer to details on slide).

The actual result of not doing this depends on what toolkit you are using (refer to details on slide).

RVCT 2.2 and earlier generated:

```
Error: L6218E: Undefined symbol Image$$ZI$$Limit
(referred from sys_stackheap.o).
```

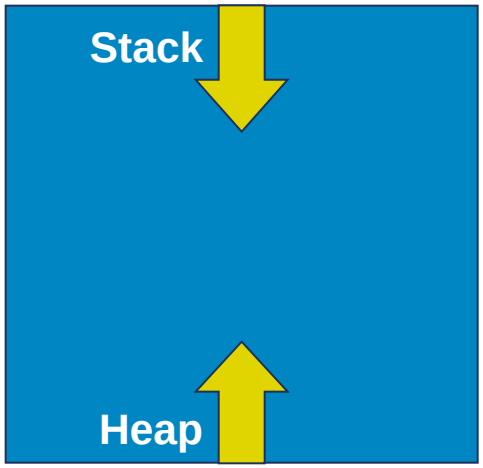
To fix this:

Define __user_initial_stackheap()

__user_initial_stackheap( ) (2)

- **To place the stack and heap you can re-implement `__user_initial_stackheap()`**
- **It may be written in C or assembler - it should return...**
  - Heap Base address in r0, Stack Base address in r1
  - Heap Limit address (if used) in r2

```
        EXPORT __user_initial_stackheap

__user_initial_stackheap
        LDR r0, =0x80000 ;HB
        LDR r1, =0x88000 ;SB
        ; r2 not used (HL)
        ; r3 not used
        MOV pc, lr
```



SB = 0x88000

HB = 0x80000

- **Heap Limit is not used in a one region model**

__user_initial_stackheap( ) (2)

Your implementation of UISH may be written in either C or assembler. As a minimum, it should return the heap base address in r0, and the stack base address in r1. If required, you should return the heap limit address in r2.

This example implements a simple one region model, where the stack grows down from address 0x88000, and the heap grows up from 0x80000.

Our implementation of UISH simply loads the appropriate values into the registers r0 and r1, and then returns.

The heap limit is not used in a one region model.

==========

NB: We have not covered the reset handler at this point, so there is no discussion of inheriting the SP from the execution environment.

This function must not corrupt registers other than r0 to r3 and ip.

Equivalent C implementation...

```
__value_in_regs struct __initial_stackheap __user_initial_stackheap(
        unsigned R0, unsigned SP, unsigned R2, unsigned unused)
{
    struct __initial_stackheap config;
    config.heap_base = 0x80000;
    config.stack_base = 0x88000;
    return config;
}
```

Stack and Heap Regions (1)

- In RVCT 3.0 and later you can place your stack and heap using special region names, without re-implementing `__user_initial_stackheap()`
- This will include a version of `__user_initial_stackheap()` from the RVCT library that uses linker defined symbols for these region names

```
LOAD_FLASH 0x24000000 0x04000000
{
  :
  ARM_LIB_STACK 0x13000 EMPTY -0x3000
  { }
  ARM_LIB_HEAP 0x15000 EMPTY 0x3000
  { }
}
```

- You still need to import `__use_two_region_memory` if using two region model
- `ARM_LIB_STACKHEAP` can be used for single region model

The Vector Table

```
        AREA Vectors, CODE, READONLY
        IMPORT Reset_Handler
        ; import other exception handlers
        ; …
        ENTRY
start
        B       Reset_Handler
        B       Undefined_Handler
        B       SWI_Handler
        B       Prefetch_Handler
        B       Data_Handler
        NOP     ; Reserved vector
        B       IRQ_Handler
        ; FIQ_Handler will follow directly

        END
```

- **Locate this table at `0x0000` (or `0xFFFF0000`) using the `+FIRST` directive**
- **`ENTRY` directive tells linker that this is an entry point**
  - This prevents the section from being removed

The Vector Table

ARM systems will all have a vector table of some form. The vector table will not form part of your reset handler per se, but it needs to be present in order for any exception to be serviced, including the reset exception. (Not quite true if using ROM/RAM remapping or similar.)

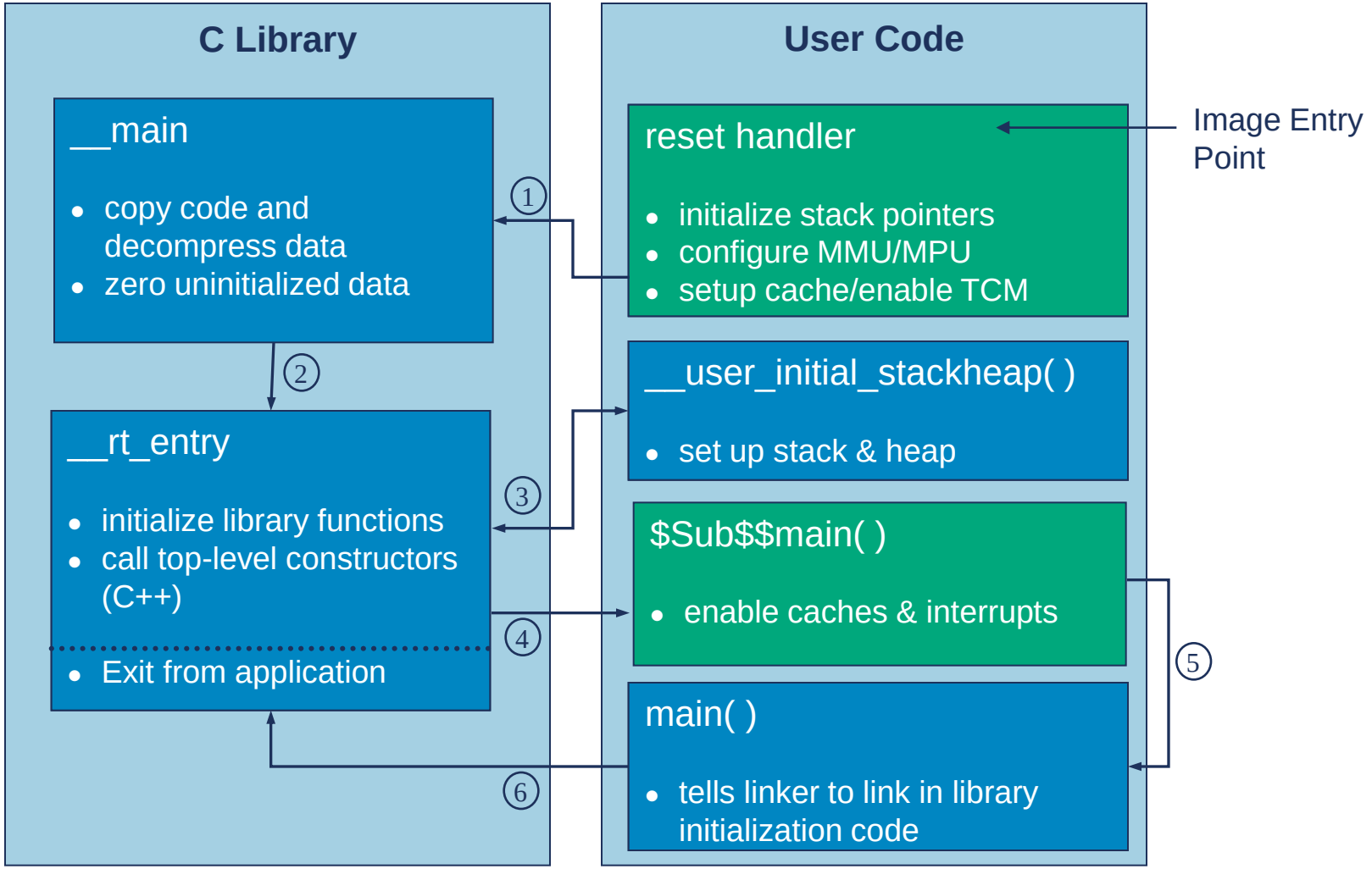
The above code imports the various exception handlers, presumably coded in other modules. The table itself is simply a list of branch instructions (remember 32MB range) to the various exception handlers.

The exception to note is the FIQ handler. The FIQ handler can be placed at address 0x1C directly. In this way, we save the processor from executing a branch to the FIQ handler.

Note also that the vector table is marked with the label ENTRY. This effectively tells the linker that this code is a possible entry point, and so it cannot be removed from the image at link time. (see RVCT linker and utilities guide for more information).

The vector table must be placed at a specific address (normally 0x0). The scatterloading +FIRST directive is well suited to do this.

Initialization Steps



Initialization Steps

This diagram shows a complete initialization process. As you can see, two blocks in green have been added to the sequence.

The first block shows a reset handler. The reset handler is a short module coded in assembler that is executed upon system reset. As a minimum , your reset handler will initialize stack pointers for the modes that your application will be running in. For cores with local memory systems, (ie: caches and/or TCM's), you will likely want to set up these systems, at this stage in the initialization process.

After executing, the reset handler will typically branch to __main to begin the C library initialization as before.

The other block that has been added enables caches (if present) and interrupts before branching to the main application.

We will discuss both of these blocks in much more detail in the following slides.

Initialize Stack Pointers

```
; Amount of memory (in bytes) allocated for stacks
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
…

…
Reset_Handler
; stack_base could be defined above, or located in a scatter file
        LDR     r0, stack_base ;
; Enter each mode in turn and set up the stack pointer
        MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit ; No interrupts
        MOV     sp, r0
        SUB     r0, r0, #Len_FIQ_Stack

        MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit ; No interrupts
        MOV     sp, r0
        …
        B       __main

       IMPORT ||Image$$ARM_LIB_STACK$$ZI$$Limit||

stack_base DCD ||Image$$ARM_LIB_STACK$$ZI$$Limit||
```

Initialize Stack Pointers

As a minimum, in your reset handler you will want to assign initial values to the stack pointers of any execution modes that your application will make use of.

In this example, the stacks are located at stack_base, a symbol that is defined in a separate assembler source file and located by a scatter file. Details of how this is done will be covered later.

We allocate 256 bytes of stack for FIQ and IRQ mode, and could do the same for any other execution mode.

To set up the stack pointers, we simply enter each mode (interrupts disabled) and assign the appropriate value to the stack pointer. We set up the system mode stack last so we exit the reset handler in system mode. We will discuss why this is done in a few moments (see slide 31).

The code uses separate SUB and MOV instructions as using SP in a data processing instruction (e.g. SUB) is deprecated in later architectures.

User should branched directly to __main at the end. If the users do not have a main function they should uses a BL (the Link is important) to __rt_lib_init to initialize the C library.

Local Memory Setup

- **The run-time memory view should be defined before entering C library initialization**
  - If your core has an MMU/MPU, it should be setup
  - ROM/RAM remapping should be completed
- **TCMs - if present should typically be enabled**
  - Be careful of TCMs masking ROM when enabled
  - TCMs should be turned on before caches
- **Coherency issues may be reduced if caches are enabled after the C library initialization code has executed**

Local Memory Setup

If you are working with a processor with a cache or TCM, you will need to consider carefully the order in which you will initialize various parts of your system.

As we have seen, parts of the C library initialization code are responsible for setting up the static memory map as well as the stack and heap. Therefore, the ARM core's run-time memory view must be setup before entering the C library initialization code.

Essentially, this means that any MMU or MPU should be setup as part of the reset handler (ROM/RAM remapping should happen here also, see the appendix for an explanation of ROM/RAM remapping).

TCMs should also be enabled if they are present (probably before MMU/MPU), because you will generally want to scatterload code and data into TCMs. As a side issue, you should be careful that you don't need to access memory that is masked by the TCMs when they are enabled. TCMs should, in general, be turned on before caches.

One final issue to note is that you run the risk of cache coherency issues if caches are enabled during scatterloading. You can avoid all of this easily if you actually turn on caches after __main finishes executing. However, you may want to keep all system initialization code separate from your main application. The next slide gives a suitable method for achieving this.

Extending Functions

- **Ideally, system initialization code is executed before entering our main application**
  - However, reset handler is not an appropriate place to enable caches or interrupts
- **We can use the $Sub and $Super function wrapper symbols**

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

void $Sub$$main(void)
{
  cache_enable();                 // enables caches
  int_enable();                   // enables interrupts
  sys_to_usr_mode();              // change mode - see next slide
  $Super$$main();                 // calls original main()
}
```

- **Also refer to**
  - RVCT Linker and Utilities Guide – Section 4.5

Extending Functions

To demonstrate how we can add to library initialization code in order to fully initialize our system before entering main application.

Ideally, we would want to keep all our system initialization code separate from our main application. (Portability).

However, some system init, for example enabling of caches and interrupts, should really happen after executing C library init code. Therefore reset handler is not a good place to do this.

To solve this problem, we can make use of the $Sub and $Super fuction wrapper symbols. Essentially, this mechanism allows us to extend functions without altering them. (RVCT Linker and Utilities Guide).

In this example we can replace the function call to main( ) with $Sub$$main( ). From there we can call a routine that enables caches, and another to enable interrupts.

We can then branch to the real main( ) by calling $Super$$main( ).

==========

The final instruction in your reset handler should be a branch to the C library initialization code

```
IMPORT   __main
B        __main
```

Execution Mode Considerations

- **It is important to consider which mode your main application will run in**
  - User Mode is an unprivileged mode - protects your system
- **System initialization can only be executed in privileged modes**
  - Need to carry out privileged operations, e.g. enable interrupts
- **If you want to run your application in a privileged mode, simply exit your reset handler in System Mode**
  - Provides a stack not used by exception modes (User Mode Stack Pointer)
- **If you want to run your application in user mode, you will need to change to user mode in $Sub$$main( )**
  - However, `__user_initial_stackheap()` must have access to your application mode registers
  - Solution is to exit reset handler in system mode, so that

    All C lib initialization code has access to user registers, but can still perform privileged operations

Execution Mode Considerations

It is important to consider what mode you will be running your main application in.

A good deal of the functionality that you are likely to implement at startup (both in the reset handler and $Sub$$main) can only be done while executing in privileged modes. For example, cache/MMU/MPU/TCM manipulation, enabling interrupts.

In the case that you wish to run your application in a privileged mode (ex. Supervisor), this is not an issue. Simply be sure to change to the appropriate mode before exiting your reset handler.

In the case where you wish to run your application in User mode, you will only be able to change to user mode after completing the necessary tasks in a privileged mode. The most likely place to do this would be in Sub$$main( ).

The major issue here is that __user_initial_stackheap needs to set up the application mode stack. Because of this, you should exit your reset handler in system mode (which can access user mode registers). __user_initial_stackheap will then execute in system mode, and so the application stack and heap will still be setup when the main application is entered.

Long Branch Veneers

- Code sections may be placed far apart (farther than BL branch range)
- The linker can automatically add Long Branch Veneers, so that ‘far’ functions can be called successfully, for example:

```
/* main.c */

int main(void)
{
    farfunc();
}



/* farfunc.c */

void farfunc(void);
{
    :
}
```

```
ROM_LOAD 0x0000
{
    ROM_EXEC 0x0000
    {
        * (+RO)
    }
    RAM 0x80000000
    {
        farfunc.o (+RO)
        * (+RW,+ZI)
    }
}
```

```
0x00000000
 bl Ven$AA$L$$farfunc

          :
          :
Ven$AA$L$$farfunc
 ldr  pc,[pc,#-4]
 dcd  0x80000000
          :
          :
0x80000000
          :
 bx lr
```

Long Branch Veneers

To demonstrate the operation of long branch veneers. This is not really a part of the embedded software development story. Because it is a piece of linker functionality that is closely related to scatterloading, it is included here.

Code sections that call each other can be placed far apart in a scatter description file. If the called function is out of range of a normal branch instruction, the linker will insert a long branch veneer.

In this example, we have a fuction main( ) that calls another farfunc ( )

We have placed all code at address zero with the exception of farfunc.o, which is over 2 gigabytes away! In order for main to call farfunc successfully the linker inserts a long branch veneer.

This example shows an ARM-ARM long branch veneer (Ven$AA). ARM-Thumb and Thumb-ARM veneers also support long branches.

The linker is intelligent about creating veneers. It will ‘share’ veneers where possible.

Memory-Mapped Registers

- **You can use scatterloading to place memory mapped peripheral registers**
- **Define them in a file e.g.** `timer_reg.c`

```
__attribute__ ((zero_init)) struct {
    volatile unsigned reg1; /* timer control */
    volatile unsigned reg2; /* timer value   */
} timer_reg;
```

Then add another execution region to the scatter file to place them at the required address in the memory map:

```
LOAD_FLASH 0x24000000 0x04000000
{
  :
    TIMER 0x40000000 UNINIT
    {
        timer_reg.o (+ZI)
    }
  :
}
```

- `UNINIT` **indicates that the ZI section should not be initialized to zero**

Memory-Mapped Registers

Suppose you had a timer peripheral with two memory mapped 32-bit control registers. You could declare a C structure such as the one above. To place this structure at a specific address in the memory map, create a new execution region to hold the structure as shown.

It is important that the contents of these registers are not initialized to zero during C library initialization (may lead to undefined behavior, or break your system). In order to prevent this from happening, mark the execution region with the UNINIT attribute.

==========

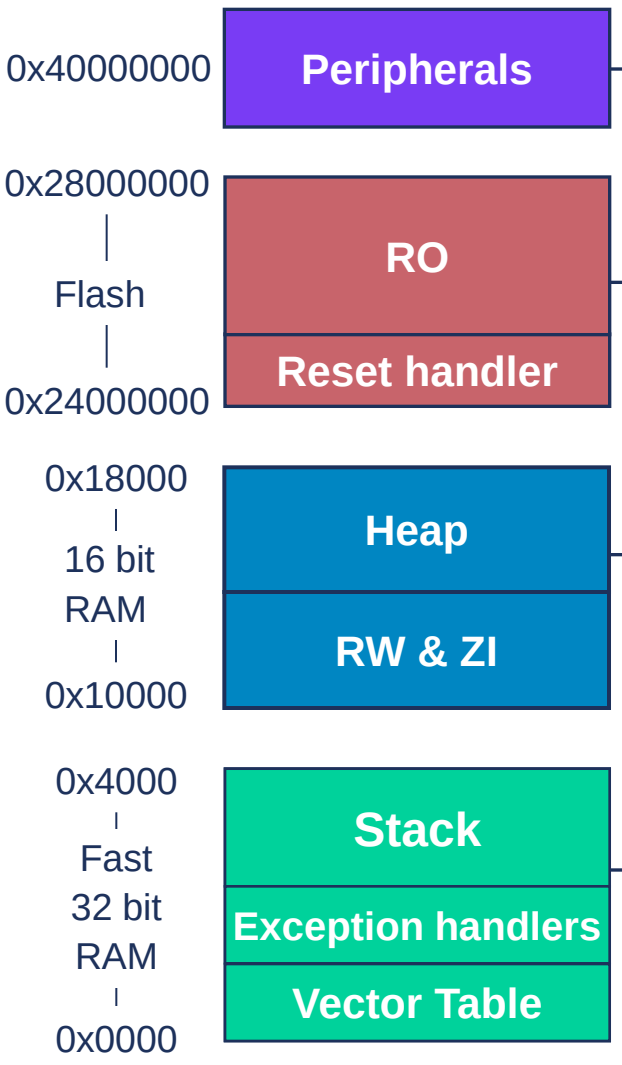
In fact, ONLY the ZI section will be uninitialized. If there is an RW section in TIMER, it will be initialized as normal. It is advisable that peripheral registers should be aligned onto WORD boundaries.

Note: In RVCT 2.1 and later, the compiler automatically places small uninitialized data/variables (<=8 bytes) in the RW section (rather than ZI). The example on the slide gets around this using the __attribute__ keyword to force the structure into the ZI region.

This compiler behaviour can also be disabled using the “--bss_threshold=0” switch on the command line.

Since the compiler only does this for data items <= 8 bytes in size, the attribute is unnecessary for structures larger than this.

Example Memory Map



Peripheral control registers are memory mapped

Bulk of code is executed directly from flash

16 bit RAM used to store data and heap region

Time critical code and data are placed in fast wide RAM

This slide shows what might be a more realistic example of a memory map. In it we have some fast wide RAM where we place our performance critical code, such as vector table, exception handlers, stack.

Following that we have a larger area of 16-bit RAM where we put our RW and ZI sections (ie: data) and our heap.

The bulk of our code is executed directly from flash.

Peripheral control registers are memory mapped.

Example Scatter File

```
FLASH 0x24000000 0x04000000
{
   FLASH 0x24000000 0x04000000
   {
     init.o (Init, +First)
     * (+RO)
   }
   32bitRAM 0x0000
   {
     vectors.o (Vectors, +First)
     handlers.o (+RO)
   }
   ARM_LIB_STACK 0x4000 EMPTY -0x3000
   {
   }
  :
```

```
    :
    :
    16bitRAM 0x10000
    {
      * (+RW,+ZI)
    }
    ARM_LIB_HEAP 0x15000 EMPTY 0x3000
    {
    }
    TIMER 0x40000000 UNINIT
    {
      timer_reg.o (+ZI)
    }
}
```

- This scatter file implements the memory map shown on the previous page

Unused Section Elimination/Entry Points

- **By default, the linker will remove from the final image any code sections that are never executed, or data that is never referred to**
  - To see if any sections have been removed, link with `'--info unused'`
  - If a section is not marked as +FIRST or +LAST, `'--keep'` can be used to prevent required sections being removed

- **To ensure that e.g. the vector table is not inadvertently removed...**
  - Mark all entry points with the assembler directive `ENTRY` (the C library has an entry point, at `__main()`), ***and***
  - Select *one* of the entry points as *the* image entry point with `'--entry'`, otherwise the linker will warn:

```
Image does not have an entry point. (Not specified or not set
due to multiple choices)
```

- **Suggested link line for ROMmable images is:**

```
armlink obj1.o obj2.o --scatter scatter.scat
                      --info unused --entry startup -o prog.axf
```

Endianness

- **Endianness determines how contents of registers relate to the contents of memory**
  - ARM registers are word (4 bytes) width
  - ARM addresses memory as a sequence of bytes
- **A basic definition of endianness**
  - Little-endian memory system: Least significant byte is at lowest address
  - Big-endian memory system: Most significant byte is at lowest address
- **ARM processors are inherently little-endian internally**
  - But, can be configured to access big-endian memory systems
- **The ARM Architecture supports three different models of endianness**
  - LE: Little-Endian
  - BE-32: Word Invariant Big-Endian
  - BE-8: Byte Invariant Big-Endian (introduced in Architecture v6)

Endianness and RVCT

- **RVCT has options for producing LE, BE-32 and BE-8 objects**
  - When compiling for Architecture v6 big endian, the tools will assume BE-8
  - Legacy BE-32 images can still be generated with an additional linker switch

Build Options	Format
`armcc -c -g test.c` `armlink test.o -o test-le.axf`	LE
`armcc -c -g --bigend test.c` `armlink test.o -o test-be8.axf --be8`	BE-8
`armcc -c -g --bigend test.c` `armlink test.o -o test-be32.axf --be32`	BE-32

- **Take care that the hardware configuration of the core is appropriate for the object format generated**

- **RVCT does not support generation of mixed endianness images**

Output Options

- **The linker produces ELF/DWARF3 images, suitable for loading into a debugger**
- **To convert the ELF image into a 'ROMmable' format, use `fromelf`, e.g.**

```
fromelf image.axf --bin -o image.bin
```

- **This generates binary files (one per load region), suitable for blowing onto ROM, downloading into Flash or EPROM-Emulator etc.**
- **Other 'ROMmable' formats can also be generated by fromelf, e.g.**
  - Motorola 32 bit Hex (--m32)
  - Intel 32 bit Hex (--i32)
  - Byte Oriented Hex (--vhx)

Debugging ROM Images

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

- **Program the Flash device & power-up/reset the target**
  or
  If RAM at 0x0, load the binary image into RAM with RVD using the command:

```
readfile raw image.bin 0x0
```

- **Load the symbolic debug information from the ELF image into RVD:**
  - Either with the GUI :

    ‘File, Load Image’ and ensure the ‘Symbols Only’ checkbox is selected
  - Or on the command line

```
load/ni image.axf
```

Placing Stack and Heap in Scatterfile

- Your stack and heap can also be placed in your scatterfile in version of RVCT prior to RVCT 3.0
- To do this you can declare EMPTY regions in your scatter file and use their addresses in your implementation of `__user_initial_stackheap()`

```
LOAD_FLASH 0x24000000 0x04000000
{
  :
  STACK 0x13000 EMPTY -0x3000
  { }

  HEAP 0x15000 EMPTY 0x3000
  { }
}
```

- The heap grows up from 0x15000
- The stack grows down from 0x13000

Placing Stack and Heap in Scatterfile

- **The linker will generate symbols that point to the base and limit of each execution region**
  - Import these symbols to refer to them in your code

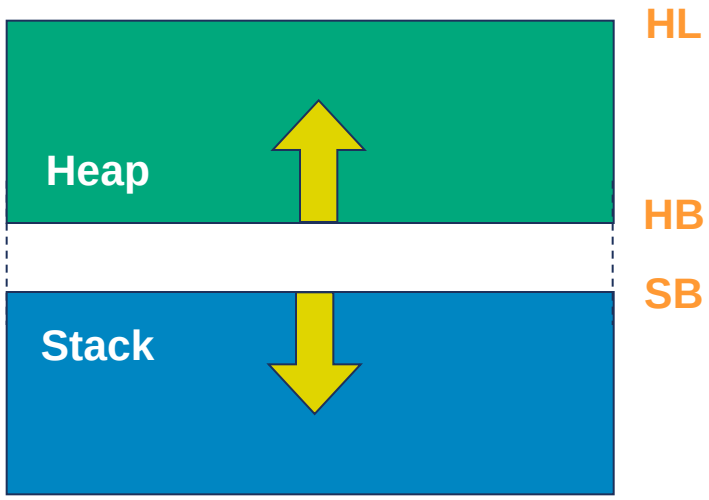
```
                IMPORT      ||Image$$STACK$$Base||
                IMPORT      ||Image$$HEAP$$Base||
                IMPORT      ||Image$$HEAP$$ZI$$Limit||

stack_base      DCD     ||Image$$STACK$$Base||
heap_base       DCD     ||Image$$HEAP$$Base||
heap_limit      DCD     ||Image$$HEAP$$ZI$$Limit||
```

- **Use DCD directive to place addresses as literals with meaningful names**

Placing Stack and Heap in Scatterfile

- In the reset handler the stack pointer (r13) is normally set up. It is passed as a parameter to `__user_initial_stackheap` in r1

```
        IMPORT __use_two_region_memory
        EXPORT __user_initial_stackheap

__user_initial_stackheap
        LDR r0, heap_base
        ;Inherit SB from reset handler
        LDR r2, heap_limit
        ;
        BX lr
```



- This example of `__user_initial_stackheap()` implements a two region memory model
  - Must import `__use_two_region_memory` so that the heap is checked against the heap limit and not the stack pointer

Placing Stack and Heap in Scatterfile

This slide provides an implementation of UISH that conforms with the example memory map. This example demonstrates the following features…

- Uses labels to place heap (reset handler also uses labels)
- Inherits SP value from execution environment (reset handler)
- Two region memory model

We can use the linker generated symbols directly in our implementation of UISH. In this case, we simply load the values of bottom_of_heap and top_of_heap into r0 and r2 directly.

In this example, r1 (stack base) is deliberately left unchanged. We can do this because the library init code passes the current value of SP into r1 before calling UISH. Because we have already set up the these values in our reset handler (init.s) we don't need to change them in our UISH.

This example of UISH implements a two region memory model, consistent with the memory map shown earlier. You can see in the code that the symbol use_two_region_memory is imported so that the heap is checked against the heap limit rather than the stack pointer.

Placing Stack and Heap in Scatterfile

Equivalent C implementation...

```
__value_in_regs struct __initial_stackheap
__user_initial_stackheap(
        unsigned R0, unsigned SP, unsigned R2, unsigned
unused)
{
    struct __initial_stackheap config;
    config.heap_base = heap_base;
    config.heap_limit = heap_limit;
    return config;
}
```

NB: This assumes that the linker generated symbols have been imported into the C module and that the appropriate #defines or variable declarations have been made.

Example Preprocessed Scatter File

```
#! armcc -E
;; Preprocess this scatter file using
;; armcc's preprocessor and #define's
;; from stack.h
#include "stack.h"
FLASH FLASH_ADDRESS FLASH_SIZE
{
   FLASH FLASH_ADDRESS
   {
     init.o (Init, +First)
     * (+RO)
   }
   32bitRAM RAM_ADDRESS
   {
     vectors.o (Vectors, +First)
     handlers.o (+RO)
   }
```

```
#ifdef TWO_REGION
  ARM_LIB_HEAP HEAP_BASE EMPTY
    HEAP_SIZE
  {
  }
  ARM_LIB_STACK STACK_BASE EMPTY
    -STACK_SIZE
  {
  }
#else
  ARM_LIB_STACKHEAP HEAP_BASE EMPTY
    STACK_SIZE + HEAP_SIZE
  {
  }
#endif
}
```

- Example scatter file that uses value defined in stack.h

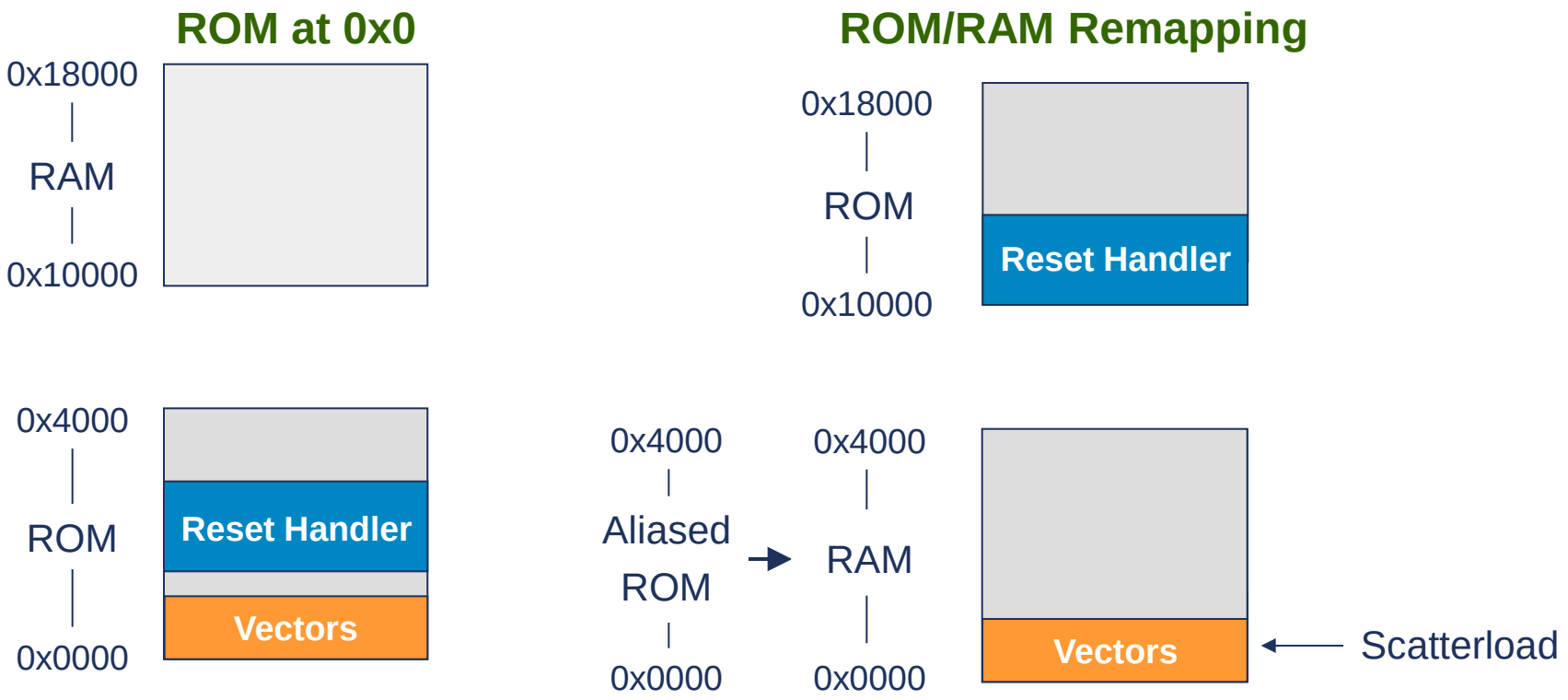
Example Preprocessed Scatter File

```
/* Define memory addresses */
#define FLASH_ADDRESS  0x24000000
#define FLASH_SIZE     0x04000000
#define RAM_ADDRESS    0x0000
/* #define TWO_REGION */

/* Stack & heap settings for one and two region models */

/* Stack start address */
#define STACK_BASE  0x20200000
/* Length stack grows downwards */
#define STACK_SIZE  0x8000
/* Heap start address */
#define HEAP_BASE   0x20100000
/* Heap length */
#define HEAP_SIZE   (0x100000-STACK_SIZE)
```

- Example stack.h to be used in scatter file preprocessing

ROM or RAM at 0x0?

- We need a valid instruction at address 0x0



- This functionality can be coded in the same module as the reset handler

ROM or RAM at 0x0?

One important consideration to make is what sort of memory your system will have at 0x0000 (assumed for this slide to be the location of the vector table).

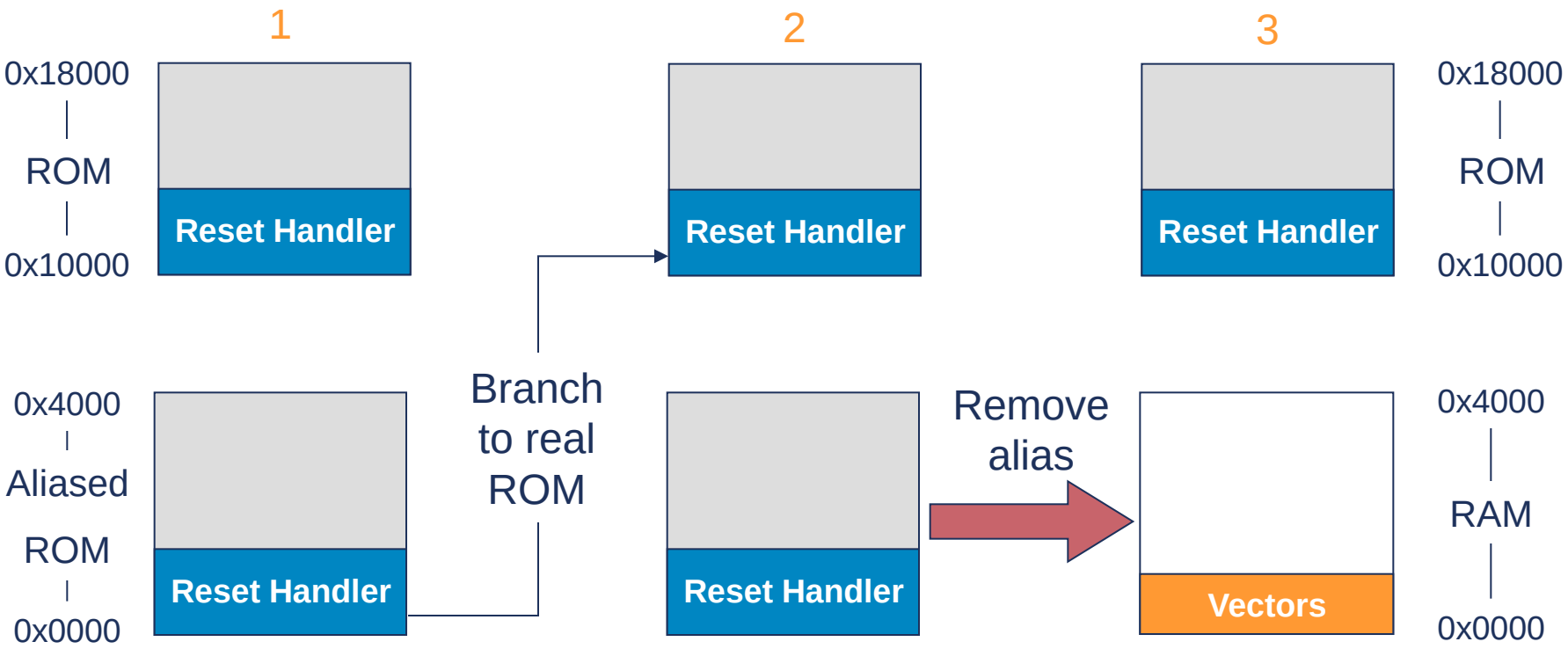
Clearly, we need a valid instruction at zero at startup. So we need to have non-volatile memory located at zero at the moment of reset.

The easy way to achieve this is to have ROM located at zero, as shown on the left. The problem with this setup is that accessing ROM is generally slow, and your system may suffer if there is too great a performance penalty when branching to exception handlers.

One possible solution is shown on the right. Real ROM is at address 0x10000. This memory is aliased to zero by the memory controller at reset. At reset, code in the reset handler branches to the real address of ROM. The memory controller then removes the aliased ROM, so that RAM is shown at address 0x0000. In __main, the vector table is copied into RAM at zero, so that exceptions can be taken when interrupts are enabled.

Additional advantages of locating a vector table in RAM are (a) you can change it dynamically at run-time and (b) you can set multiple breakpoints on it using software breakpoints. (Not an issue for systems with vector catch).

ROM/RAM Remapping



1. At reset, ROM is aliased to address 0x0000
2. We branch to real ROM at address 0x10000
3. It is now safe to remove the alias of ROM at zero to expose RAM - the vector table can then be copied to 0x0000

ROM/RAM Remapping

- The following can be coded in the same source file as the reset handler

```
; Integrator CM control reg
CM_ctl_reg       EQU  0x1000000C     ; Address of CM Control Register
Remap_bit        EQU  0x04           ; Bit 2 is remap bit of CM_ctl

        ENTRY

; On reset, an alias of ROM is at 0x0, so jump to 'real' ROM.
        LDR     pc, =Instruct_2

Instruct_2
; Remap by setting Remap bit of the CM_ctl register
        LDR     r1, =CM_ctl_reg
        LDR     r0, [r1]
        ORR     r0, r0, #Remap_bit
        STR     r0, [r1]

; RAM is now at 0x0.
; The exception vectors must be copied from ROM to RAM (in __main)

; Reset_Handler follows on from here
```

- This functionality can also be implemented by an MMU

ROM/RAM Remapping

This example shows how you might implement ROM/RAM remapping in an ARM assembler module. The constants shown here are specific to the Integrator AP platform, but the general idea should be applicable to any platform that implements ROM/RAM remapping in a similar way.

The first instruction is a jump from aliased ROM to real ROM. This can be easily done since the label instruct_2 will be located at the real ROM address.

After this step, removing the alias of ROM is done easily by writing to the control register of the Integrator core module.

The reset handler could easily follow on from this code.

==========

This code (the first instruction) must be position independent (because it actually executes at a different address than its load address).

Note that functionality similar to this can be implemented using an MMU. See initializing cached cores for more details.