Application Note 107

Embedded Software Development with ADS v1.2

Document number: ARM DAI 0107A

Issued: October 2002

Application Note 107
Embedded Software Development with ADS v1.2

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
October 2002	A	First release

Proprietary notice

Confidentiality status

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

Most embedded applications are initially developed in a prototype environment with resources that differ from those available in the final product. As such, it is important to consider the processes that enable your application to run on your target hardware.

The aim of this application note is to examine the processes involved in moving an embedded application from one that relies on the facilities of the development/debugging environment to a system that runs standalone on target hardware. In particular, this document illustrates some of the features of the ARM Developer Suite (ADS) v1.2, and suggests how they might be effectively used in this regard.

1.1 What is covered in this application note?

With ADS, several issues must be considered to move from an “out-of-the-box” build to a standalone embedded application:

- C library use of hardware
- Some C library functionality executes by using debug environment resources. If used, this functionality must be re-implemented to make use of target hardware.
- ADS has no inherent knowledge of the memory map of any given target. The image memory map must be tailored to the memory layout of the target hardware.
- An embedded application must perform some initialization before the main application can be run. A complete initialization sequence requires user-implemented code as well as ADS C library initialization routines.

This application note addresses each of the above issues. Also, some further considerations regarding the image memory map are highlighted.

1.2 Example code

To illustrate the topics covered in this application note, associated example projects are provided.

The Dhrystone benchmarking program provides the code base for the example projects. Dhrystone was chosen because it provides a simple, but non-trivial, main application that illustrates the topics described in this application note.

Included in the example are several directories, each containing a distinct build of Dhrystone. Each build provides an example of the techniques discussed in each successive chapter of the document. Specific information regarding each build can be found in the sections labeled Example Code in this document.

The example projects are tailored to run on the ARM Integrator development platform. However, the principles illustrated by the examples are applicable to any target hardware.

Note *The focus of this application note is not the Dhrystone program itself, but the steps that must be taken to enable it on a fully standalone system. For further discussion of Dhrystone as a benchmarking tool, see Application Note 93 – Benchmarking with ARMulator.*

2 A Default Build

When beginning to develop software for an embedded application, a user of ADS might not have technical specifications of their target hardware. Details of target peripheral devices, the memory map, or perhaps even the processor itself might be unknown or undecided.

To enable software development before such details are known or considered, the ADS tools have a default behavior that enables the user to start building and debugging application code immediately. It is useful to recognize this default behavior, in order to appreciate the steps necessary to move from a default build to a fully standalone application.

2.1 The ADS 1.2 C Library

2.1.1 Semihosting

In the ADS C Library, support for some ANSI C functionality is provided by the host debugging environment. The mechanism by which this is provided is termed semihosting.

Semihosting is implemented by a set of defined software interrupt (SWI) operations. When a semihosting SWI is executed, the debug agent identifies it and briefly suspends program execution. The semihosting operation is then serviced by the debug agent before code execution is resumed. Therefore, the task performed by the host itself is transparent to the program.

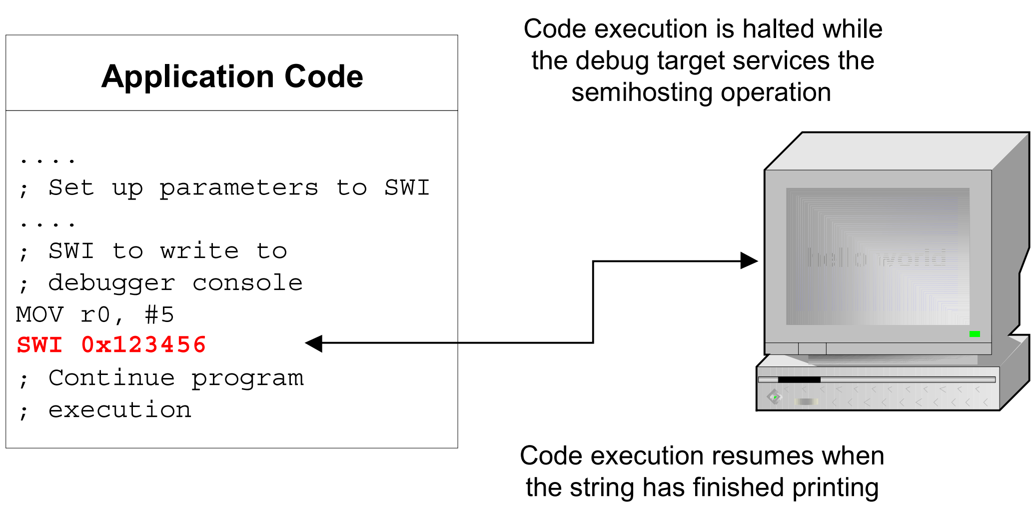

Figure 2-1 Example Semihosting operation

Figure 2-1 shows an example of semihosting operation, which prints a string to the debugger console.

Note *For more information on semihosting, see the ADS Debug Target Guide Section 5.*

2.1.2 C Library Structure

Conceptually, the C library can be divided into functions that are part of the ANSI C Language specification, and functions that provide support to this ANSI C level. This is illustrated in Figure 2-2.

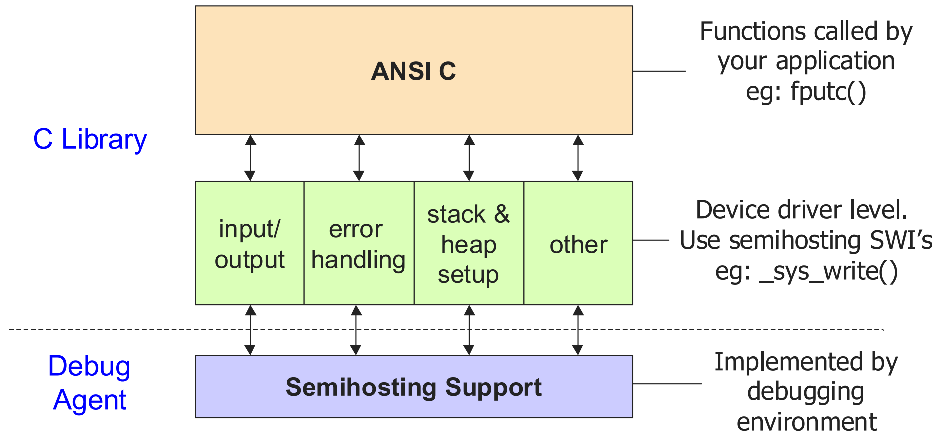

Figure 2-2 C Library Structure

Support for some ANSI C functionality is provided by the host debugging environment via a device driver level of support functions.

For example, the ADS C library implements the ANSI C `printf()` family of functions by writing to the debugger console window. This functionality is provided by calling `__sys_write( )`, a support function that executes a semihosting SWI which results in a string being written to the console.

2.2 Default Memory Map

In an image where the memory map has not been described by the user, ADS places code and data according to a default memory map.

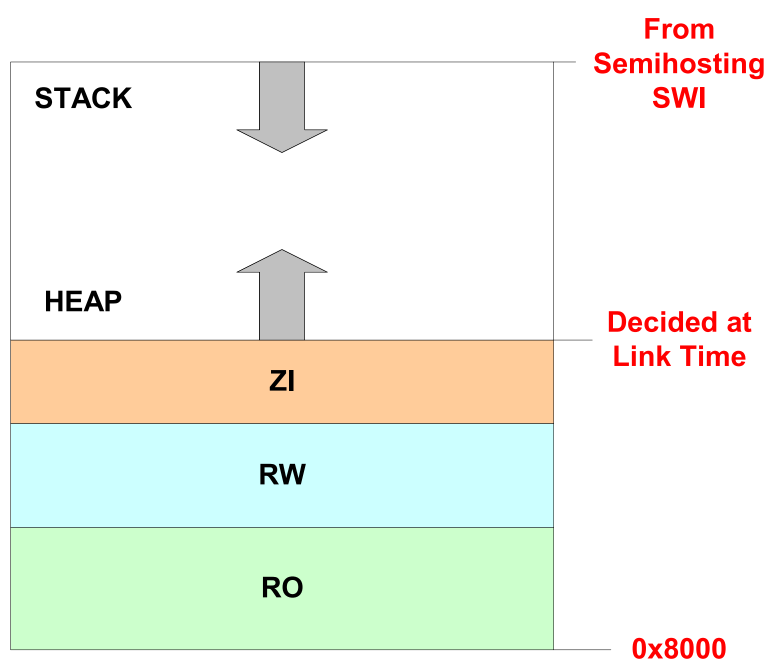

Figure 2-3 Default memory map

The default memory map (shown in Figure 2-3) is as follows:

- The image will be linked to load and run at address 0x8000. All RO (Read Only) sections are placed first, followed by RW (Read-Write) sections then ZI (Zero-Initialized) sections.

- The heap follows directly on from the top of the ZI section, so the exact location is decided at link time.
- The stack base location is provided by a semihosting operation during application startup. The value returned by this semihosting operation depends on the debug environment:
    - ARMulator returns the value set in the configuration file peripherals.ami. The default is 0x08000000.
    - Multi-ICE returns the value of the debugger internal variable `$top_of_memory`. The default is 0x00080000.

2.3 Linker Placement Rules

The linker observes a set of rules, shown in Figure 2-4, to decide where code and data is located.

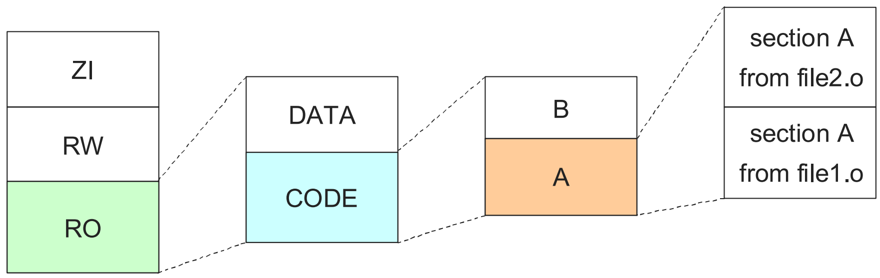

Figure 2-4 Linker placement rules

The image is organized first of all by attribute - RO precedes RW precedes ZI. Within each attribute code precedes data.

From there, the linker places input sections alphabetically by name. Input section names correspond with assembler area directives.

Within input sections, code and data from individual objects are placed in the order the object files are specified on the linker command line.

The user is not advised to rely on these rules if precise placement of code and data is required. Full control of placement of code and data is available via the scatterloading mechanism (discussed in Section 4.1).

Note *See section 3.2 of the ADS Linker and Utilities Guide for more information on placement rules.*

2.4 Application Startup

In most embedded systems, an initialization sequence executes to setup the system before executing the main task.

The default ADS initialization sequence is shown in Figure 2-5.

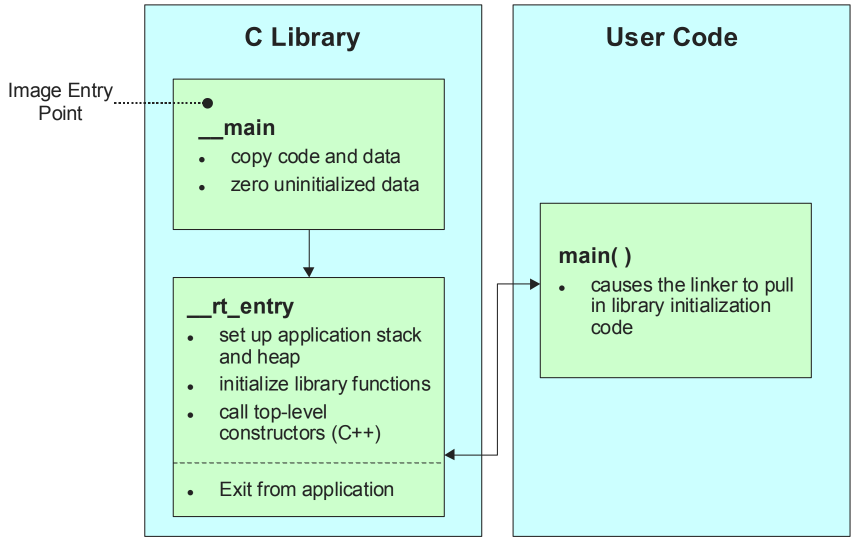

Figure 2-5 Default ADS Initialization sequence

At a high level, the initialization sequence can be divided into two functional blocks. `__main` is responsible for setting the run-time image memory map, whereas `__rt_entry` is responsible for initializing the C library.

`__main` carries out code and data copying, and zeroing of ZI data. This step is only significant when the run-time location of code and data differs from that at load time (see section 4.1).

`__main` branches to `__rt_entry` (run-time entry). This sets up the application stack and heap initializes library functions and static data, and calls any constructors of globally declared objects (C++ only). `__rt_entry` then branches to `main()`, the entry to the user's application. When the main application has finished executing, `__rt_entry` hands control back to the debugger.

The function label `main()` has a special significance in ADS. The presence of a `main()` function forces the linker to link in the initialization code in `__main` and `__rt_entry`. Without a function labeled `main()` the initialization sequence is not linked in, and as a result, some standard C library functionality is not supported.

2.5 Example Code – Build 1

Build 1 is a default build of the Dhrystone Benchmark. As such, it adheres to the default ADS behavior described in this section.

To run this build on an Integrator, you must:

- ROM/RAM remapping must have been performed. This can easily be achieved by running the Boot Monitor (Switches 1 & 4 on).
- Set $top_of_memory to 0x40000, or fit a DIMM memory module. If this is not done, the stack (which defaults to 0x80000) may not be in valid memory.

3 Tailoring the C Library to your Target Hardware

By default the C library makes use of semihosting to provide device driver level functionality. A real embedded system makes use of target peripherals.

3.1 Retargeting the C Library

You can provide your own implementation of C Library functions that make use of target hardware, which are automatically linked in to your image in favor of the C library implementations. This process, known as retargeting the C library, is shown in Figure 3-1.

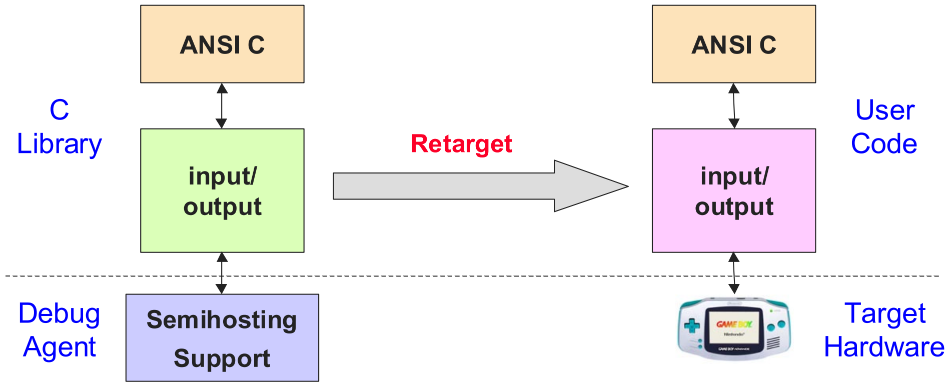

Figure 3-1 Retargeting the C library

For example, you might have a peripheral I/O device such as a UART, in which case you want to override the library implementation of `fputc()`, which writes to the debugger console, with one that outputs to the UART. Since this implementation of `fputc()` is linked in to the final image, the entire `printf()` family of functions print out to the UART.

An example implementation of `fputc()` is shown below.

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

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

This example simply redirects the input character parameter of `fputc()` to a serial output function `sendchar()`, which is assumed to be implemented in a separate source file. In this way, `fputc()` acts as an abstraction layer between target dependent output and the C library standard output functions.

3.2 Avoiding C Library Semihosting

In a standalone application, you cannot support semihosting SWI operations. Therefore you must be certain that no C library semihosting functions are being linked into your application.

To ensure that no functions which use semihosting SWIs are linked in from the C library, you must import the symbol `__use_no_semihosting_swi`. This can be done in any C or assembler source file in your project.

- `In a C module,` use the `#pragma` directive.

```
#pragma import(__use_no_semihosting_swi)
```

- `In an assemble module, use` the `IMPORT` directive.

  ```
  IMPORT __use_no_semihosting_swi
  ```

If there are still SWI-using functions being linked in, the linker reports the following error:

```
Error: Symbol __semihosting_swi_guard multiply defined
```

To identify which SWI-using functions are still being linked in, link with the `-verbose` switch. In the resulting output, C library SWI-using functions are tagged with `__I_use_semihosting_swi`.

```
Loading member sys_exit.o from c_a__un.l.
              definition:  _sys_exit
              reference :  __I_use_semihosting_swi
```

You must provide your own implementations of these functions.

It is important to note that the linker does not report any semihosting SWI-using functions in the user's own application code. An error only occurs if a semihosting SWI-using function is linked in from the C library.

Note *ADS 1.2 Compiler and Libraries Guide, Table 4-2 gives a full list of SWI-using C library functions.*

3.3 Example – Build 2

Build 2 of the example uses the Integrator platform's hardware for clocking and string I/O.

The following changes were made to Build 1 of the example project:

- C Library Retargeting

  A retargeted layer of ANSI C functions has been added. These include standard input/output functionality, clock functionality, as well as some additional error signaling and program exit.

- Target Dependent Device Driver

  A device driver layer that interacts directly with target hardware peripherals has been added.

To run this build on an Integrator:

- ROM/RAM remapping must have been performed. This can easily be achieved by running the Boot Monitor (Switches 1 & 4 on).
- Set $top_of_memory to 0x40000, or fit a DIMM memory module. If this is not done, the stack (which defaults to 0x80000) may not be in valid memory.

Note *The symbol `__use_no_semihosting_swi` is not imported into this project. This is because a semihosting-SWI is executed during C library initialization to set up the application stack and heap location. Retargeting stack and heap setup is covered in detail in section 4.2.*

Note *To see the output, a terminal or terminal emulator (such as Hyperterminal) must be connected to serial port A. The serial port settings should be set to 38400 baud, no parity, 1 stop bit and no flow control. The terminal should be configured to append line feeds to incoming line ends, and echo typed characters locally.*

4 Tailoring the Image Memory Map to your Target Hardware

4.1 Scatterloading

In a real embedded system, you almost certainly do not want to adhere to the default memory map provided by ADS. Your target hardware usually has several memory devices located at different address ranges. To make the best use of these devices, you will want to have separate views of memory at load and run-time.

Scatterloading enables the user to describe the load-time and run-time location of code and data in memory in a textual description file known as a scatter file. The scatter file is passed to the linker on the command line using the `-scatter` switch. For example:

```
armlink -scatter scat.scf file1.o file2.o
```

The scatter file describes to the linker the desired location of code and data at both load-time and run-time, in terms of addressed memory regions. Scatterloading regions fall into two categories:

- Load Regions which contain application code & data at reset/load time.
- Execution Regions which contain code and data while the application is executing. One or more execution regions are created from each load region during application startup.

All code and data in the image falls into exactly one load region, and one execution region.

During startup, C library initialization code in __main carries out the copying and zeroing of code and data necessary to move from the image load view to the execute view.

4.1.1 Scatter File Syntax

Scatter file syntax reflects the functionality provided by scatterloading itself.

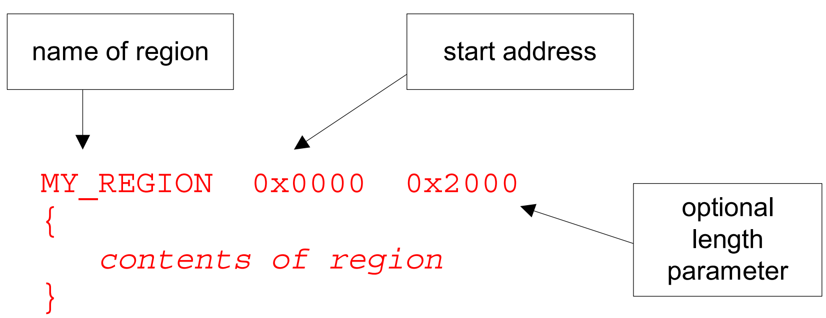

Figure 4-1 Scatter file syntax

A region is defined by a header tag that contains, as a minimum, a name for the region and a start address. Optionally, a maximum length and various attributes can be added. Open and closed curly braces delimit the contents of a region.

The contents of the region depend on the type of region.

- Load regions must contain at least one execution region. In practice, there are usually several execution regions per load region.
- Execution regions must contain at least one code or data section. These are usually source or library object files. The wildcard (*) syntax can be used to group all sections of a given attribute not specified elsewhere in the scatter file.

Note *For a more detailed description of scatter file syntax, see Chapter 5 of the ADS Linker and Utilities Guide.*

4.1.2 Simple Scatterloading Example

Figure 4-2 illustrates a simple example of scatterloading.

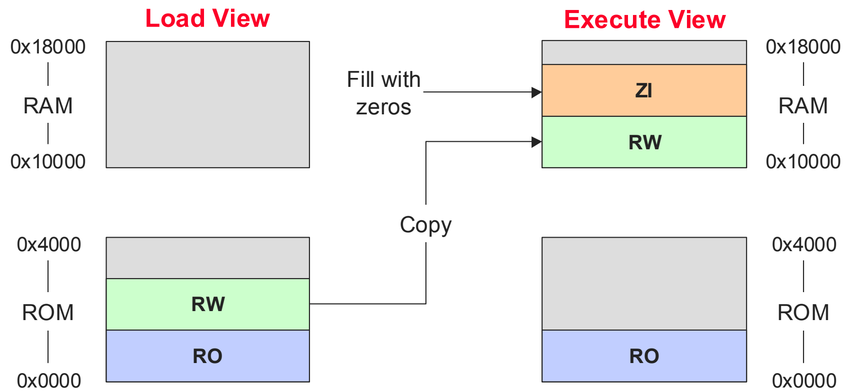

Figure 4-2 Simnple scatterloading example

This example 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.

Below is the scatter description file that describes the above memory map.

```
LOAD_ROM 0x0000 0x4000
{
     EXE_ROM 0x0000 0x4000      ; Root region
     {
          * (+RO)               ; All code and constant data
     }

     RAM 0x10000 0x8000
     {
          * (+RW, +ZI)          ; All non-constant data
     }
}
```

4.1.3 Placing Objects in a Scatter File

In most images, you will want to control the placement of specific code and data sections, rather than grouping all attributes together as in the previous example. This can be done by specifying individual objects directly in the scatter file, rather than relying only on the wildcard syntax.

Note *The ordering of objects within a scatter file execution region does not affect their ordering in the output image. The linker placement rules described in Section 2.3 apply to each execution region.*

To override the standard linker placement rules, we can use the +FIRST and +LAST scatterloading directives. A typical example is placing the vector table at the beginning of an execution region:

```
LOAD_ROM 0x0000 0x4000
{
  EXEC_ROM 0x0000 0x4000
  {
    vectors.o (Vect, +FIRST)
    * (+RO)
```



```
    }
    ; more exec regions…
}
```

In this scatter file, we ensure that the area `Vect` in vectors.o is placed at address 0x0000.

4.1.4 Root Regions

A root region is an execution region whose load address is equal to its execution address. Each scatter file must have at least one root region.

One restriction placed on scatterloading is that the code and data responsible for creating execution regions (ie: copying and zeroing code and data) cannot itself be copied to another location. As a result, the following sections must be included in a root region:

- **`__main.o`** - contains the code that copies code/data
- **`Region$$Table`** and **`ZISection$$Table`** - sections which contain the addresses of the code/data to be copied.

Because the above sections are attributed as read-only, they are grouped by the `* (+RO)` wildcard syntax. As a result, if `* (+RO)` is specified in a non-root region, the above must be explicitly declared in a root region.

An example is shown below:

```
LOAD_ROM 0x0000 0x4000
{
     EXE_ROM 0x0000 0x4000 ; root region
     {
          __main.o (+RO)        ; copying code
          * (Region$$Table)     ; RO/RW addresses to copy
          * (ZISection$$Table)  ; ZI addresses to zero
     }

     RAM 0x10000 0x8000
     {
          * (+RO)               ; all other RO sections
          * (+RW, +ZI)          ; all RW and ZI sections
     }
}
```

Failing to include `__main.o`, `Region$$Table`, and `ZISection$$Table` in a root region results in the linker generating an error message.

4.2 Placing the Stack and Heap

Scatterloading provides a method for specifying the placement of code and statically allocated data in your image. We now look at how to place the application stack and heap.

4.2.1 Retargeting __user_initial_stackheap( )

The application stack and heap are setup during C library initialization. We are able to tailor stack and heap placement by retargeting the routine responsible for stack and heap setup. In the ADS C library, this routine is `__user_initial_stackheap()`.

The diagram below shows the C library initialization process with a retargeted `__user_initial_stackheap()`.

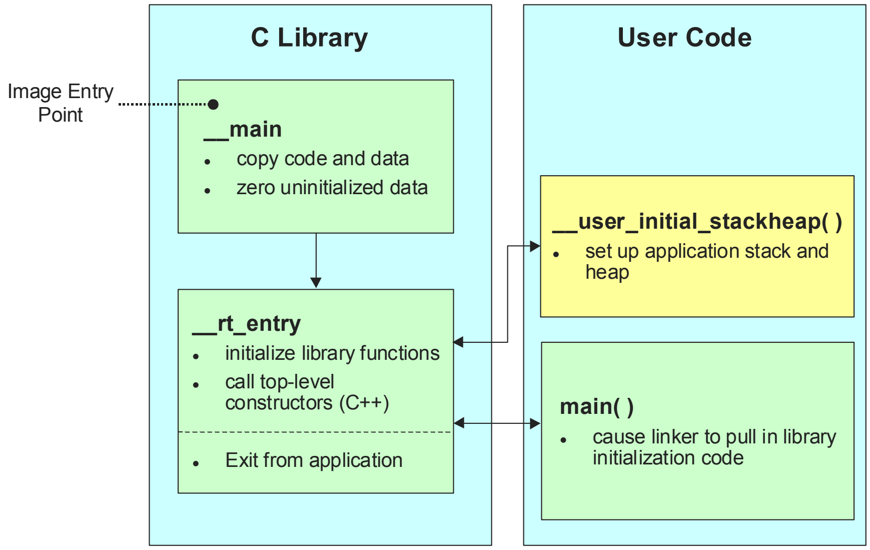

Figure 4-3 Retargeting __user_initial_stackheap()

`__user_initial_stackheap` can be coded in C or ARM assembler. It must return the following parameters:

- heap base in r0
- stack base in r1
- heap limit in r2 (if required)
- stack limit in r3 (if required)

You must re-implement `__user_initial_stackheap` if you are scatterloading your image. Otherwise, the linker will generate the following error:

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

Note *In ADS v1.1, no error message is generated. Instead, the heap base is located (often inappropriately) at address 0x0000.*

4.2.2 Run-time Memory Models

ADS provides two possible run-time memory models. In the default model, the application stack and heap grow towards each other in the same region of memory. This is called the one-region model. In this case, the heap is checked against the value of the stack pointer when new heap space is allocated (that is, when `malloc()` is called).

On the other hand, 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 stack use only. To inform ADS that you wish to use a two-region model, you must import the symbol `use_two_region_memory`. The heap is then checked against a dedicated heap limit, which is set up by `__user_initial_stackheap`.

In both run-time memory models, the stack grows unchecked by default. You can optionally enable software stack checking in your image by compiling all modules with the compiler switch `-apcs /swst`. If you are using a two-region model, you must also specify a stack limit in your implementation of `__user_initial_stackheap`.

Note *Enabling software stack checking introduces a substantial code size and performance overhead, since the value of the stack pointer must be checked against the stack limit with each function call.*

4.2.3 Example Implementations

One-region model

```
EXPORT __user_initial_stackheap

__user_initial_stackheap
     LDR r0, =0x20000 ;HB
     LDR r1, =0x40000 ;SB
     ; r2 not used (HL)
     ; r3 not used (SL)
     MOV PC, LR
```

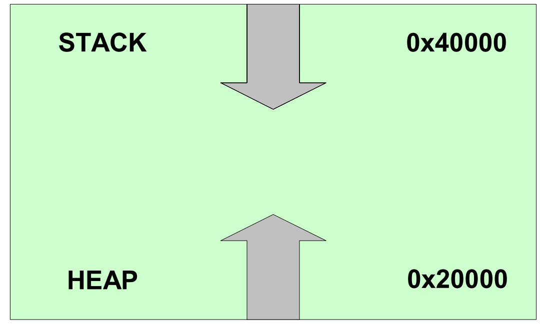

Figure 4-4 One Region Model

The above example of `__user_initial_stackheap` implements a simple one-region model, where the stack grows down from address 0x40000, and the heap grows up from 0x20000. The routine simply loads the appropriate values into the registers r0 and r1, and then returns. r2 and r3 remain unchanged, because a heap limit and stack limit are not used in a one-region model.

Two-region model

```
IMPORT __use_two_region_memory
EXPORT __user_initial_stackheap

__user_initial_stackheap
     LDR r0, =0x28000000 ;HB
     LDR r1, =0x40000 ;SB
     LDR r2, =0x28080000 ; HL
     LDR r3, =0x20000 ;SL
     MOV PC, LR
```

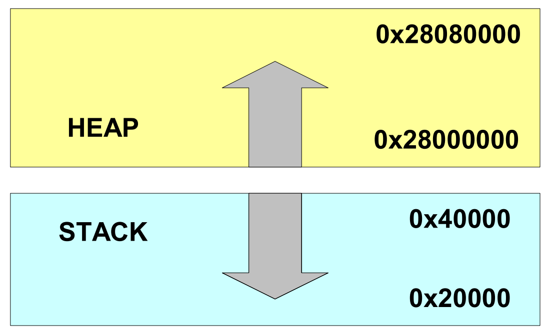

Figure 4-5 Two Region Model

The above example implements a two-region model. The stack grows down from 0x40000 towards a limit of 0x20000. To make use of this stack limit, all modules using this implementation must be compiled for software stack checking. The heap grows up from 0x28000000 to 0x28080000.

Note *`__use_two_region_memory` is imported using the assembler `IMPORT` directive.*

Note *Both examples above are suitable for the Integrator system.*

4.3 Example Code – Build 3

Build 3 of the example implements scatterloading, and contains a retargeted `__user_initial_stackheap`.

The following modifications were made to build 2 of the example project:

- Scatterloading

  A simple scatter description file is passed to the linker.

- Retargeted `__user_initial_stackheap`

You have the option of selecting either a one-region or a two-region implementation. The default build is one-region. The two-region implementation can be selected by defining two_region at the build step.

- Avoiding C library Semihosting

To run this build on an Integrator, ROM/RAM remapping must have been performed. This can easily be achieved by running the Boot Monitor (Switches 1 & 4 on).

The symbol `__use_no_semihosting_swi` is imported into this build, because there are no longer any C library semihosting functions present in the image.

Note *In order to avoid using semihosting for clock(), this is retargeted to read the Real Time Clock (RTC) on the Integrator AP. This has a resolution of one second, so the results from Dhrystone will not be precise. This mechanism is improved in Build 4.*

Note *It is important to disable all 'Vector Catch' and semihosting if you are using an ARM7 core based target. Otherwise the debugger will interpret the execution of instruction between address 0x0 and 0x1C as exceptions, and report this in a dialogue box. This can be set via the 'Options -> Configure Processor' menu.*

5 Reset and Initialization

Until now, we have assumed that execution begins at `__main`, the entry point to the C library initialization routine. In fact, any real embedded application performs some system-level initialization at startup. This section discusses this in more detail.

5.1 Initialization Sequence

Figure 5-1 shows a possible initialization sequence for an ARM-based embedded system.

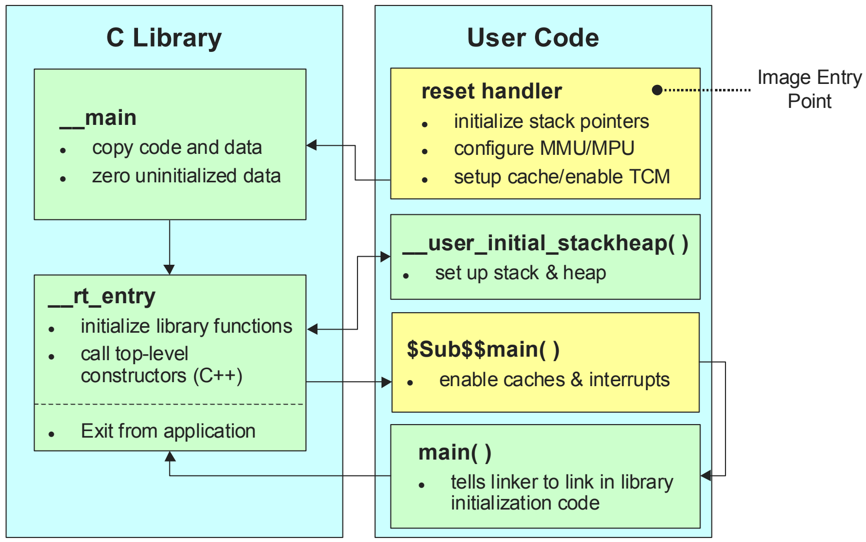

Figure 5-1 Initialization sequence

To Figure 5-1 we have added a reset handler, which executes immediately at system startup. We also have a block of code labeled `$Sub$$main()`, which executes immediately before entering the main application.

The reset handler is a short module coded in assembler that is executed on system reset. As a minimum, your reset handler initializes stack pointers for the modes that your application is running in. For cores with local memory systems, (that is, cache and/or tightly coupled memory), some configuration must be done at this stage in the initialization process. After executing, the reset handler typically branches to `__main` to begin the C library initialization sequence.

There are some components of system initialization, for example the enabling of interrupts, which are generally performed after the C library initialization code has finished executing. The block of code labeled `$Sub$$main()` performs such tasks immediately before the main application begins executing.

Section 5.2 describes the various components of the initialization sequence in more detail.

5.2 The Vector Table

All ARM systems have a vector table. The vector table does not form part of the initialization sequence as such, but it must be present for any exception to be serviced.

```
            AREA Vectors, CODE, READONLY

      IMPORT Reset_Handler
; import other exception handlers

      ; …
```

```
        ENTRY

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
```

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

The FIQ handler is placed at address 0x1C directly. In this way, we avoid having to execute a branch to the FIQ handler, so optimizing FIQ response time.

Note *The vector table is marked with the label ENTRY. This effectively tells ADS that this code is a possible entry point, and so it cannot be removed from the image at link time. You must select one of the possible image entry points as the true entry point to your application using the –entry linker option. See the ADS Linker and Utilities Guide section 3.1.4 for more information.*

5.3 Memory Setup

5.3.1 ROM/RAM Remapping

Note *We assume in this section that the ARM core begins fetching instructions at 0x0000, which is the norm for ARM core based systems. Some ARM cores can be configured to begin fetching instructions from 0xFFFF0000.*

One important consideration to make is what sort of memory your system has at 0x0000, the address of the first instruction executed.

Clearly, you require a valid instruction at 0x0000 at startup, so you have to have non-volatile memory located at 0x0000 at the moment of reset.

A simple way to achieve this is to have ROM located at 0x0000. However, there are some drawbacks to this configuration. Access speeds to ROM are generally slower than to RAM, and your system might suffer if there is too great a performance penalty when branching to exception handlers. Also, locating the vector table in ROM does not enable you to modify it at run time.

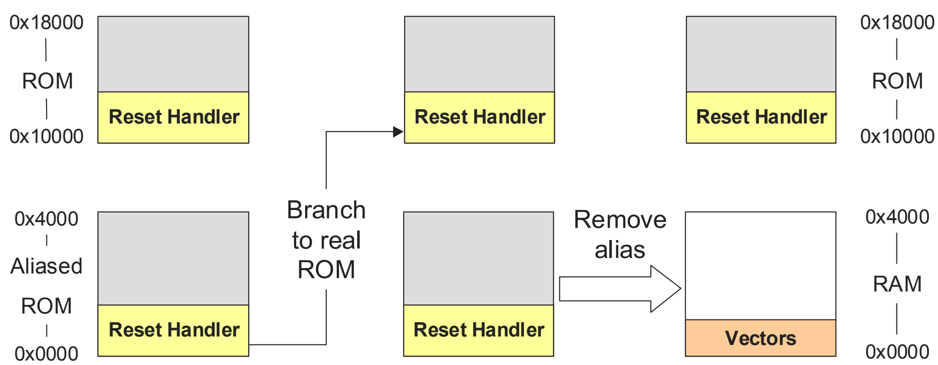

Figure 5-2 ROM/RAM Remap

Another possible solution is shown above. ROM is located at address 0x10000, but this memory is aliased to zero by the memory controller at reset. Following 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 0x0000, so that exceptions can be serviced.

```
; --- 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
```

The above code shows how you might implement ROM/RAM remapping in an ARM assembler module. The constants shown here are specific to the Integrator platform, but the same method is 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 done because the label `instruct_2` is located at the real ROM address.

After this step, the alias of ROM is removed by flipping the remap bit of the Integrator Core Module control register.

The above code is normally executed immediately after system reset. Remapping must be completed before C library initialization code can be executed.

Note *In systems with memory management units (MMUs), remapping can be implemented through MMU configuration at system startup.*

5.3.2 Local Memory Setup Considerations

Many ARM processor cores have on-chip memory systems, such as caches, tightly coupled memories (TCMs), memory management units (MMUs) and memory protection units (MPUs). Such devices are normally setup and enabled during system startup. As such the initialization sequence of cores with local memory systems requires special consideration.

As we have seen, C library initialization code in `__main` is responsible for setting up the execution time memory map of the image. Therefore, the processor core's run-time memory view must be set up before branching to `__main`. Essentially, this means that any MMU or MPU must be set up and enabled in the reset handler.

TCMs must also be enabled before branching to `__main` (normally before MMU/MPU setup), because you generally want to scatterload code and data into TCMs. As a side issue, you must be careful that you do not have to access memory that is masked by the TCMs when they are enabled.

One final issue to note is that you run the risk of cache coherency issues if caches are enabled before branching to `__main`. Code in `__main` copies code regions from their load address to their execution address – essentially treating instructions as data. As a result, some instructions can be cached in the data cache, in which case they are not visible to the instruction path.

You can avoid such coherency issues easily if you simply enable caches after the C library initialization sequence finishes executing.

5.3.3 Scatterloading and Memory Setup

In a system where the reset-time memory view of the core is altered, either through ROM/RAM remapping or MMU configuration, the scatterloading description file must describe the image memory map after remapping has taken place.

For example the scatter file for the example in Section 5.3.1, resembles the following:

```
LOAD_ROM 0x10000 0x8000
{
     EXE_ROM 0x10000 0x8000
     {
          reset_handler.o (+RO, +FIRST)
          …
     }

     RAM 0x0000 0x4000
     {
          vectors.o (+RO, +FIRST)
          …
     }
}
```

The load region LOAD_ROM is placed at 0x10000, because this indicated the load address of code and data after remapping has occurred.

5.4 Stack Pointer Initialization

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

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

Offset_FIQ_Stack         EQU     0
Offset_IRQ_Stack         EQU     Offset_FIQ_Stack + Len_FIQ_Stack
…

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
        SUB     sp, r0, #Offset_FIQ_Stack

        MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
        SUB     sp, r0, #Offset_IRQ_Stack
        …

; Set up stack limit if needed
        LDR     r10, stack_limit
```

In the above example, the stacks are located at `stack_base`. This symbol can be a hard coded address, or it can be defined in a separate assembler source file and located by a scatter file. Details of how this is done are given in section 6.2.

The example allocates 256 bytes of stack for FIQ and IRQ mode, and you can do the same for any other execution mode. To set up the stack pointers, simply enter each mode (interrupts disabled) and assign the appropriate value to the stack pointer. If we are to make use of software stack checking, you to have set up a stack limit here as well.

Stack pointer and/or stack limit values set up in the reset handler are automatically passed as parameters to `__user_initial_stackheap` by C library initialization code. These values therefore must not be modified by `__user_initial_stackheap`.

The following implementation of `__user_initial_stackheap` can be used with the stack pointer setup routine above.

```
        IMPORT heap_base
        EXPORT __user_initial_stackheap

__user_initial_stackheap

; heap base could be hard coded, or placed by scatter file
    LDR   r0,=heap_base
    ; r1 contains SB value
    MOV   pc,lr
```

5.5 Hardware Initialization - $Sub$$main( )

In general, it is beneficial to separate all system initialization code from the main application. However, some components of system initialization, for example enabling of caches and interrupts, must occur after executing C library initialization code.

We can make use of the $Sub and $Super function wrapper symbols to effectively insert a routine that is executed immediately before entering the main application. Essentially, this mechanism enables us to extend functions without altering the source code itself.

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

void $Sub$$main(void)
{
  cache_enable();             // enables caches
  int_enable();               // enables interrupts
  $Super$$main();             // calls original main()
}
```

Above is an example of how $Sub and $Super can be used in this way. The linker replaces the function call to `main()` with a call to `$Sub$$main()`. From there we can call a routine that enables caches, and another to enable interrupts.

The code branches to the real `main()` by calling `$Super$$main()`.

Note *More information on $Sub and $Super can be found in the ADS 1.2 Linker and Utilities Guide.*

5.6 Execution Mode Considerations

It is important to consider what mode the main application will run in. Your choice affects how you implement system initialization.

A lot 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, and enabling interrupts.

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

If you wish to run your application in User mode, you can only change to User mode after completing the necessary tasks in a privileged mode. The most likely place to do this is in `Sub$$main()`.

An important issue to note is that `__user_initial_stackheap` must set up the application mode stack. Because of this, you must exit your reset handler in system mode (which uses the User mode registers). `__user_initial_stackheap` then executes in system mode, and so the application stack and heap are still set up when User mode is entered.

5.7 Example Code – Build 4

Build 4 of the example can be run standalone on the Integrator platform.

The following modifications were made to build 3 of the example project:

- Vector Table

  A vector table was added to the project, and placed by the scatter file.

- Reset Handler

  The reset handler is added in init.s. Two separate modules, responsible for TCM and MMU setup respectively, are included in the ARM926EJ-S build. These are excluded from the ARM7TDMI build, which will run on Integrator systems with any core. ROM/RAM remapping occurs immediately after reset.

- $Sub$$main( )

  For the ARM926EJ-S build, Caches are enabled in $Sub$$main( ) before entering the main application.

- Embedded Scatter File

  An embedded scatter file is used, which reflects the post-remapping view of memory.

The batch files for both of these builds produce a binary file suitable for downloading into the Integrator AP application Flash at address 0x24000000. This can be achieved via the ‘File -> Flash Download menu’. A separate application note is available which describes this process.

A precise timer is implemented using a timer on the AP motherboard. This generates an IRQ, and a handler is installed which increments a counter every 1/100 second.

6 Further Memory Map Considerations

6.1 Locating hardware addresses in the scatter file

So far, we have described the placement of code and data in a scatter file, but the location of target hardware peripherals and the stack and heap limits are assumed to be hard coded in source or header files. It would be beneficial to locate all information pertaining to the target's memory map in our scatter file so removing all references to absolute addresses from our source code.

6.1.1 Locating Target Peripherals in the Scatter File

Conventionally, addresses of peripheral registers are hard-coded in project source or header files. One can also declare structures that map on to peripheral registers, and place these structures in the scatter file.

For example, a target could have a timer peripheral with two memory mapped 32-bit registers. Below is a C structure that maps on to these registers.

```
struct {
    volatile unsigned ctrl; /* timer control */
    volatile unsigned tmr;  /* timer value   */
} timer_regs;
```

To place this structure at a specific address in the memory map, create a new execution region to hold the structure.

```
LOAD_FLASH 0x24000000 0x04000000
{
 ; …
 TIMER 0x40000000 UNINIT
 {
     timer_regs.o (+ZI)
 }
 ; …
}
```

The above scatter file locates the `timer_regs` structure at 0x40000000.

It is important that the contents of these registers are not initialized to zero during application startup, because this is likely to alter the state of your system. Marking an execution region with the `UNINIT` attribute prevents ZI data in that region from being zero-initialized.

6.2 Locating the Stack and Heap in the Scatter File

In many cases, it is preferable to specify the location of the stack and heap in the scatter file. This has two main advantages:

- All information about the memory map is kept in one file.
- Changes to the stack and heap will only require re-linking, not recompiling.

This section describes two methods for implementing this.

6.2.1 Placing symbols explicitly

With ADS v1.2, this is the simplest of the two methods

Section 5.4 refers to the symbols stack_base and heap_base as reference symbols that can be placed in a scatter file. To do this, create symbols labeled `stack_base` and `heap_base` in an assembler module (the same can be done for the stack and heap limits in a two-region memory model).

```
        AREA   stacks, DATA, NOINIT
        EXPORT stack_base

stack_base        SPACE   1

        AREA   heap, DATA, NOINIT
        EXPORT heap_base

heap_base         SPACE   1

        END
```

These symbols can be located each in their own execution region in the scatter file.

```
LOAD_FLASH 0x24000000 0x04000000
{
 ; …
 HEAP 0x20000 UNINIT
 {
     stackheap.o (heap)
 }

 STACKS 0x40000 UNINIT
 {
     stackheap.o (stacks)
 }
 ; …
}
```

The above scatter description file places the heap base at 0x20000 and the stack base at 0x40000. The stack and heap base locations can now be easily altered by editing the addresses of the respective execution regions.

6.2.2 Utilizing linker generated symbols

With ADS v1.2, this method requires that the stack and heap size are specified in an object file. To some extent this negates the advantages described in section 6.2. However this is the easiest method to migrate to the recommended method for future ARM tools, where additional features will avoid this requirement.

First, define areas of an appropriate size for the stack and heap in an assembler source file, for example, stackheap.s. The space directive can be used to reserve a zeroed block of memory. Setting the ‘NOINIT’ area attribute prevents this zeroing (during development, you might choose to zero-initialize the stack so that the maximum stack usage can be seen). Note that labels are not required in this source file.

```
      AREA stack, DATA, NOINIT
      SPACE   0x3000 ; Reserve stack space

      AREA heap, DATA, NOINIT
      SPACE   0x3000 ; Reserve heap space

      END
```

These sections can then be placed in their own execution region in the scatter file.

```
LOAD_FLASH 0X24000000 0x04000000
{
      :
      STACK 0x1000 UNINIT  ; length = 0x3000
      {
            stackheap.o (stack) ; stack = 0x4000 to 0x1000
      }

      HEAP 0x15000 UNINIT  ; length = 0x3000
      {
            stackheap.o (heap) ; heap = 0x15000 to 0x18000
      }
}
```

The linker generates symbols that point to the base and limit of each execution region, which can be imported into the retargeting code to be used by __user_initial_stackheap. This code can be made more readable by using the DCD directive to give these values more meaningful names:

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

      stack_base  DCD     ||Image$$STACK$$ZI$$Limit||
      stack_limit DCD     ||Image$$STACK$$ZI$$Base||

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

The above files are suitable to place the heap base at 0x15000 and the stack base at 0x40000. The stack and heap base locations can now be easily altered by editing the addresses of the respective execution regions.

6.3 Example Code – Build 5

Build 5 of the example is equivalent to Build 4, but with all target memory map information located in the scatter file as described in section 6.2.1:

- Scatter File Symbols

  Symbols to locate the stack, heap, and peripherals are declared in assembler modules.

- Updated Scatter File

  The embedded scatter file from build 4 is updated to locate the stack, heap, data TCM, and peripherals.

6.4 Example Code – Build 6

Build 6 of the example is equivalent to Build 5, but with all target memory map information located using linker generated symbols relating to the scatter file as described in section 6.2.2.

7 References

For additional information see:

- Example code in <ads_install>\Examples\embedded directory
- ADS 1.2 Developer Guide

  Chapter 6 – Writing Code for ROM
- ADS 1.2 Compilers and Libraries Guide

  Chapter 4 – The C and C++ Libraries
- ADS 1.2 Debug Target Guide

  Chapter 5 – Semihosting
- ADS 1.2 Linker and Utilities Guide

  Chapter 5 – Using Scatterloading Description Files.