AVR32795: Using the GNU Linker Scripts on AVR UC3 Devices

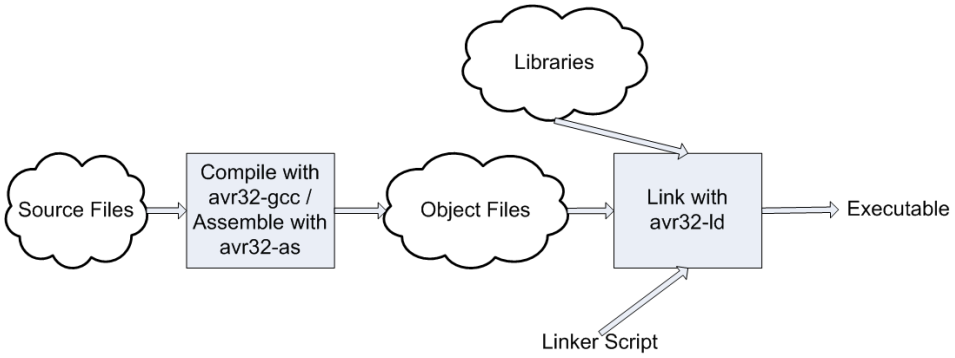
Features

- **Basic GNU linker script concepts**
- **32-bit AVR® UC3™ GNU linker scripts**
- **Controlling the location of functions and variables in the flash**

1 Introduction

This document highlights the main purpose of the GNU linker script, which is to control the location of code and variables in the executable.

Figure 1-1. AVR GNU toolchain build steps.



32-bit AVR® Microcontrollers

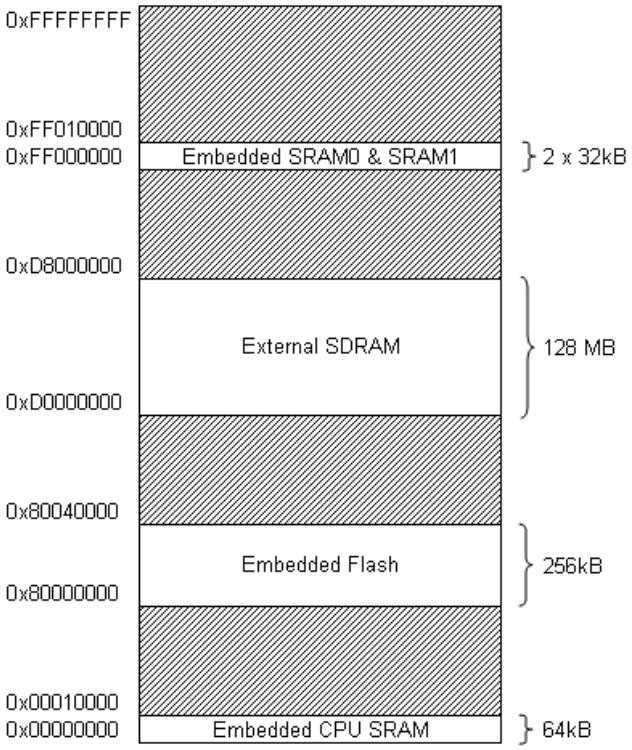
Application Note

Rev. 32158A-AVR-01/11

2 Memory map

The Atmel® AVR UC3 microcontroller architecture has a 32-bit memory space and separate memory types (program and data) connected with distinct buses. Such a memory architecture allows the processor to access both program and data memories at the same time. Each memory type has its own address space.

Figure 2-1. Example of the Atmel AT32UC3A3256 memory map.



3 Basic linker script concepts

3.1 Sections

The linker combines input files (object file format) into a single output file (executable).

Each object file has, among other things, a list of sections. We refer to a section in an input file as an input section. Similarly, a section in the output file is an output section.

Each section in an object file has a name and a size. Most sections also have an associated block of data (the section contents).

3.2 Section properties

A section may be marked as loadable, which means that its contents should be loaded into memory when the executable is run.

A section with no contents may be allocatable, which means that an area in memory should be set aside, but nothing in particular should be loaded there (and, in some cases, this memory must be zeroed out).

A section which is neither loadable nor allocatable typically contains some sort of debugging information.

3.3 VMA and LMA

Every loadable or allocatable output section has two addresses. The first is the VMA, or virtual memory address. This is the address the section will have when the output file is run. The second is the LMA, or load memory address. This is the address at which the section will be loaded. In most cases the two addresses will be the same.

An example of when the LMA and VMA might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (a technique often used to initialize global variables in a ROM-based system). In this case, the ROM address would be the LMA and the RAM address would be the VMA.

3.4 Symbols

Every object file also has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information.

The compilation of a C or C++ program into an object file will generate a defined symbol for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

3.5 Well-known sections

.text: usually contains the code, and is usually loaded to a non-volatile memory, such as the internal flash.

.data: initialized data; usually contains initialized variables.

.bss: usually contains non-initialized data.

NOTE The section, properties, VMA, and LMA are available in an object file or output file by using the avr32-objdump program with the `-h` option. For example:

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .reset        00002004  80000000  80000000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .got          00000000  0000001c  80003634  0000501c  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, DATA
  2 .init         0000001a  80002004  80002004  00003004  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .text         00001074  80002020  80002020  00003020  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .exception    00000200  80003200  80003200  00004200  2**9
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .fini         00000018  80003400  80003400  00004400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .rodata       00000208  80003418  80003418  00004418  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .dalign       00000004  00000004  00000004  00000000  2**0
                  ALLOC
  8 .ctors        00000008  00000008  80003620  00005008  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .dtors        00000008  00000010  80003628  00005010  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 10 .jcr          00000004  00000018  80003630  00005018  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 11 .data         00000820  0000001c  80003634  0000501c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 12 .bss          0000015c  00000854  00000854  00000000  2**2
                  ALLOC
 13 .heap         0000e650  000009b0  000009b0  00000000  2**0
                  ALLOC
```

NOTE The symbols are available in an object or output file by using the avr32-nm program, or by using the avr32-objdump program with the `-t` option.

4 Default versus specific linker script

The linker always uses a linker script. If none is explicitly supplied, the linker will use a default script that is compiled into the linker executable. The AVR GNU toolchain default linker scripts are under the directory:

C:/Program Files/Atmel/AVR Tools/AVR(32) Toolchain/avr32/lib/ldscripts/

Other linker scripts can be supplied by using the `-T` command line option (or the long form: `-script=<file>`). When this is done, the linker script specified will replace the default linker script.

Extensions of the GNU toolchain linker scripts and their meanings:

`.x`: default linker script, for “regular” executables.

`.xbn`: default linker script used when the `-N` option is specified; mix text and data on the same page; don't align data.

`.xn`: default linker script used when the `-n` option is specified; mix text and data on the same page.

`.xr`: default linker script used when the `-r` option is specified; link without relocation.

`.xu`: default linker script used when the `-Ur` option is specified; link without relocation, create constructors.

`.xw`: linker script to use for writable .rodata section.

5 GNU linker script walkthrough

`/* … */` : Comments

`OUTPUT_FORMAT("elf32-avr32", "elf32-avr32", "elf32-avr32")`:
The `OUTPUT_FORMAT` command names the BFD format to use for the output file. In this case, this is strictly equivalent to `OUTPUT_FORMAT("elf32-avr32")`

`OUTPUT_FORMAT(default, big, little)`:
`OUTPUT_FORMAT` is usable with three arguments to use different formats based on the `-EB` and `-EL` command line options.

This permits the linker script to set the output format based on the desired endianness.

`OUTPUT_ARCH(avr32:uc)`: Specify a particular output machine architecture.

`ENTRY(_start)`: Set the first instruction to execute in a program (called the entry point).

5.1 MEMORY command

Describes the location and size of blocks of memory in the target.

```
MEMORY
{
  FLASH (rxai!w) : ORIGIN = 0x80000000, LENGTH = 256K
  INTRAM (wxa!ri) : ORIGIN = 0x00000004, LENGTH = 0x0000FFFC
  USERPAGE : ORIGIN = 0x80800000, LENGTH = 0x00000200
}
```

Name used in the linker script to refer to that memory region

Optional list of attributes

- `r`: read-only section
- `w`: r/w section
- `x`: executable section
- `a`: allocatable section
- `i`: initialized section
- `!`: invert the sense of the following attributes

The linker will use the region attributes to select the memory region for the output section that it creates (if not explicitly mentioned later in the script).

Once a memory region is defined, the linker script can direct the linker to place specific output sections into that memory region.

5.2 PHDRS command

The ELF object file format uses program headers, aka segments. The program headers describe how the program should be loaded into the target memory.

```
PHDRS
{
  FLASH PT_LOAD;
```

This program header describes a segment to be loaded from the file

```
  INTRAM_ALIGN PT_NULL;

  INTRAM_AT_FLASH PT_LOAD;

  INTRAM PT_NULL;

  USERPAGE PT_LOAD;

}
```

Indicates an unused (for loading) program header

To place an output section in a particular segment, use the `:phdr` output section attribute.

When the executable is programmed to target, only the loadable segments should be programmed.

NOTE There are types other than `PT_LOAD` and `PT_NULL` (refer to the GCC linker scripts documentation for details).

5.3 SECTIONS command

Tells the linker how to map input sections into output sections, and how to place the output sections in memory.

```
SECTIONS
{
  sections-command
  sections-command
  ...
}
```

Each `SECTIONS` command may be one of the following:

- an `ENTRY` command
- a symbol assignment
- an output section description
- an overlay description

5.3.1 Symbol assignment

A value can be assigned to a symbol. This will define the symbol as a global symbol.

Example 1:

```
/* Use a default stack size if stack size was not defined. */

__stack_size__ = DEFINED(__stack_size__) ? __stack_size__ : 4K;
```

Example 2:

```
PROVIDE (__executable_start = 0x80000000);

. = 0x80000000;
```

Indicates an unused (for loading) program header

NOTE The special symbol "." is the location counter.

NOTE

If the address of an output section is not specified, the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. At the start of the SECTIONS command, it equals zero by default.

5.3.2 Output section description

Most programs consist only of code, initialized data, and uninitialized data. These will be in the .text, .data, and .bss sections, respectively. For most programs, these are also the only sections that appear in the input files.

5.3.2.1 Sections

```
SECTIONS
{
. = 0x80000000;
.text : { *(.text) }
. = 0x00000000;
.data : { *(.data) }
.bss : { *(.bss) }
}
```

Set the value of the location counter

List the names of the input sections that should be placed into this output section: “all .text input sections in all input files”

Section name

The first line inside the SECTIONS command of the above example sets the value of the special symbol “.”, which is the location counter. If any address of an output section is specified in some other way, the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. At the start of the SECTIONS command, the location counter has the value 0.

The second line defines an output section, .text. Within the curly braces after the output section name, it lists the names of the input sections that should be placed into this output section.

The “*” is a wildcard, which matches any file name. The expression *(.text) means all .text input sections in all input files.

Because the location counter is 0x80000000 when the output section .text is defined, the linker will set the address of the .text section in the output file to be 0x80000000.

The remaining lines define the .data and .bss sections in the output file. The linker will place the .data output section at address 0x00000000.

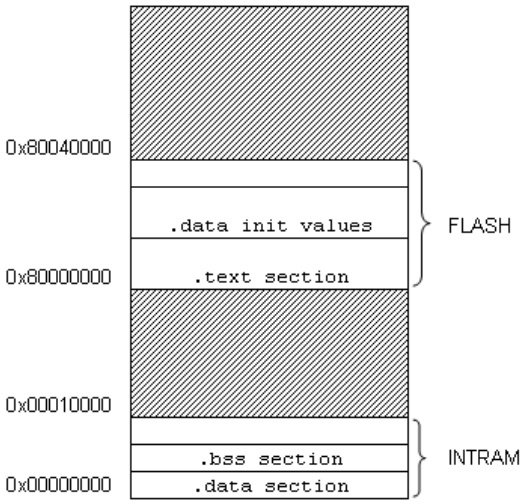
After the linker places the .data output section, the value of the location counter will be 0x00000000 plus the size of the .data output section.

The effect is that the linker will place the .bss output section immediately after the .data output section in memory.

The linker will ensure that each output section has the required alignment, by increasing the location counter if necessary.

In this example, the specified addresses for the .text and .data sections will probably satisfy any alignment constraints, but the linker may have to create a small gap between the .data and .bss sections.

Figure 5-1. Memory mapping.



5.3.2.2 Text

```
.text           :
{
  *(.text .stub .text.* .gnu.linkonce.t.*)
  KEEP (*(.text.*personality*))
  /* .gnu.warning sections are handled specially by elf32.em.*/
  *(.gnu.warning)
} >FLASH AT>FLASH :FLASH =0xd703d703
```

The program may use external libraries (for example, gcc libraries, newlib) that use specific input sections

Set the fill pattern for an entire section

Assign a section to a previously defined segment (cf PHDRS command)

Specify a memory region for the .text section LMA

Assign a section to a previously defined region of memory (cf. MEMORY command)

NOTE

When link-time garbage collection is in use (-gc-sections), it is often useful to mark sections that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with `KEEP()`.

6 Examples

6.1 Controlling the location of functions and variables in the flash

6.1.1 Process flow

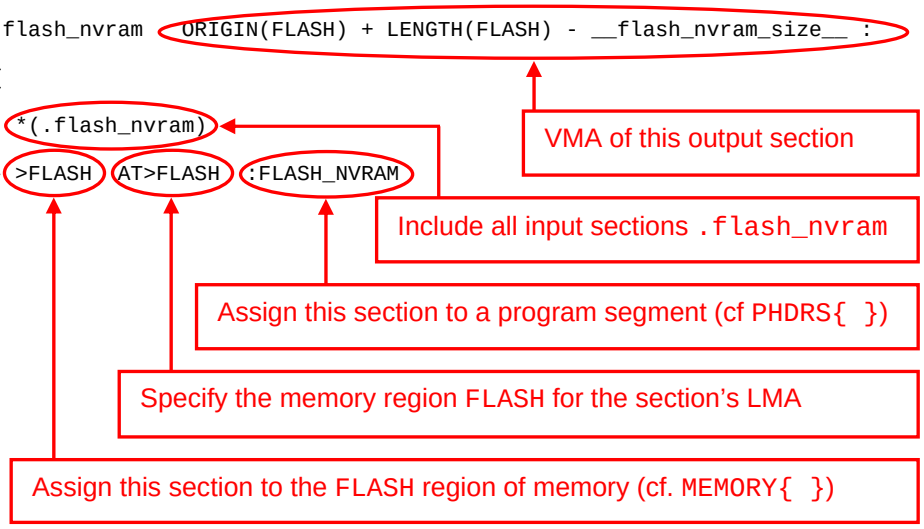
A. Define a custom section, `.flash_nvram`, located in flash:

A.1. Handle a default or pre-defined size for this section:

```
/* Use a default flash NVRAM size if flash NVRAM size was not defined. */
__flash_nvram_size__ = DEFINED(__flash_nvram_size__)?__flash_nvram_size__:4K;
```

A.2. Describe the output section, `.flash_nvram`:

```
.flash_nvram  ORIGIN(FLASH) + LENGTH(FLASH) - __flash_nvram_size__ :
{
  *(.flash_nvram)
} >FLASH  AT>FLASH  :FLASH_NVRAM
```



B. Locate a variable in a custom section, `.flash_nvram`:

- Extensions to the C language family:
  - Specifying attributes of variables

```
__attribute__((__section__(".flash_nvram"))) static int flash_nvram_data;
```

C. Locate a function in a custom section, `.flash_nvram`:

- Extensions to the C language family:
  - Declaring attributes of functions

```
__attribute__ ((__section__(".flash_nvram"))) void Func(void) {…}
```

NOTE Placing a variable or a function at a specific address has to be done through the linker script (place the custom section at the specific address).

6.1.2 Related examples in the software framework

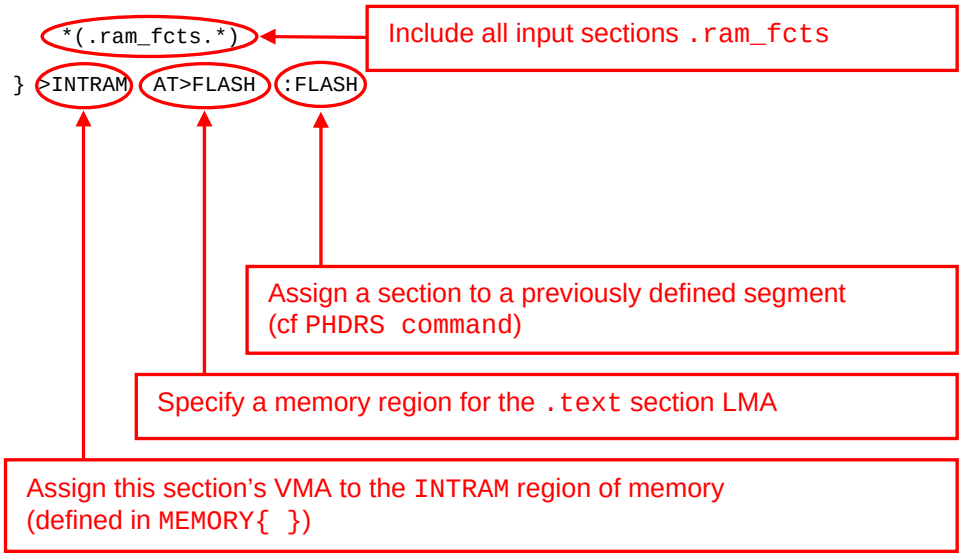
- drivers/flashc/flash_example/: Controlling the location of a variable in flash
- drivers/cpu/mpu/example/: Controlling the location of a function in flash

6.2 Controlling the location of functions in internal RAM

6.2.1 Process flow

A. Define a custom output section, `.ram_fcts`, with VMA in RAM and LMA in flash:

```
.ram_fcts       :
{
   *(.ram_fcts.*)
} >INTRAM AT>FLASH :FLASH
```



B. Locate a function in a custom section, `.ram_fcts`:

- Extensions to the C language family:
  - Declaring attributes of functions

```
__attribute__ ((__section__(".ram_fcts"))) void Func(void) {…}
```

C. The startup routine is responsible for copying the `.ram_fcts` LMA (in flash) to the `.ram_fcts` VMA (in `INTRAM`), as is done for the `.data section`.

6.2.2 Related examples

Examples of a startup routine implementation can be found in the AVR Software Framework under utils/startup_files/gcc/ in ASF v1 and under utils/startup/startup_uc3.S in ASF v2.

An example using the same method can be found here:

AVR32749 Application note: Software Workaround Implementation for the Erratum Flash Read-after-Write for AT32UC3A0512 / AT32UC3A1512 Revision E, H and I.

6.3 Controlling the location of variables and the heap in external SDRAM

6.3.1 Process flow

A. Linker script customization

A.1. Adding the external SDRAM memory (`MEMORY{}`)

```
MEMORY
{
  FLASH (rxai!w) : ORIGIN = 0x80000000, LENGTH = 0x00040000
  INTRAM (wxa!ri) : ORIGIN = 0x00000004, LENGTH = 0x0000FFFC
  ERAM0 (wxa!ri) : ORIGIN = 0xFF000000, LENGTH = 0x00008000
  ERAM1 (wxa!ri) : ORIGIN = 0xFF008000, LENGTH = 0x00008000
  SDRAM (wxa!ri) : ORIGIN = 0xD0000000, LENGTH = 0x02000000
  USERPAGE : ORIGIN = 0x80800000, LENGTH = 0x00000200
}
```

A.2. Adding the external SDRAM segments (`PHDRS{}`)

```
PHDRS
{
  FLASH PT_LOAD;
  INTRAM_ALIGN PT_NULL;
  INTRAM_AT_FLASH PT_LOAD;
  INTRAM PT_NULL;
  SDRAM_AT_FLASH PT_LOAD;
  SDRAM PT_NULL;
  USERPAGE PT_LOAD;
}
```

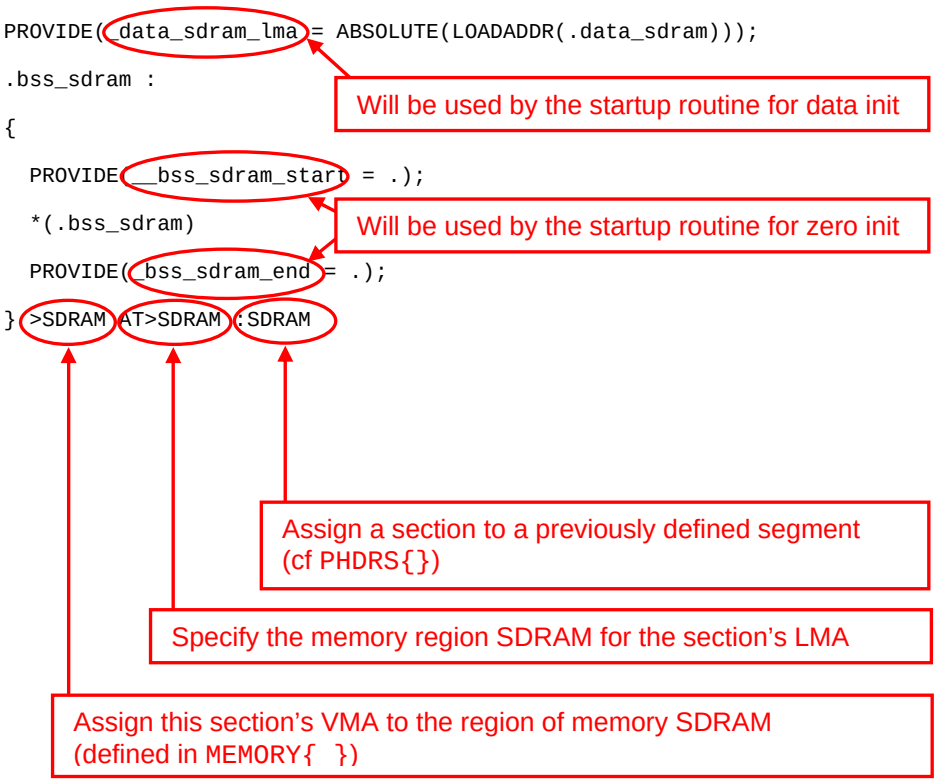
A.3. Defining two output sections for variables placed in external SDRAM (`.data_sdram`, `.bss_sdram`)

```
. = ORIGIN(SDRAM);
.data_sdram    ORIGIN(SDRAM) :
{
  PROVIDE(_data_sdram = .);
  *(.data_sdram)
  . = ALIGN(8);
  PROVIDE(_edata_sdram = .);
} >SDRAM AT>FLASH :SDRAM_AT_FLASH
```

Set the location counter to SDRAM start address

Set the VMA of `.data sdram`

Will be used by the startup routine for data init



NOTE Because the program header `SDRAM` was defined as `PT_NULL`, the `.bss_sdram` section won't be loaded to target (which is ok because the `.bss` section is supposed to hold uninitialized data). And so the `AT>SDRAM` is unnecessary and ignored. This section will just be zeroed out in the startup routine.

NOTE Definitions:

`_data_sdram`: VMA address of the start of the `.data_sdram` section

`_data_sdram`: VMA address of the end of the `.data_sdram` section

`_data_sdram_lma`: LMA start address of the `.data_sdram` section

```
>SDRAM AT>FLASH :SDRAM_AT_FLASH
```

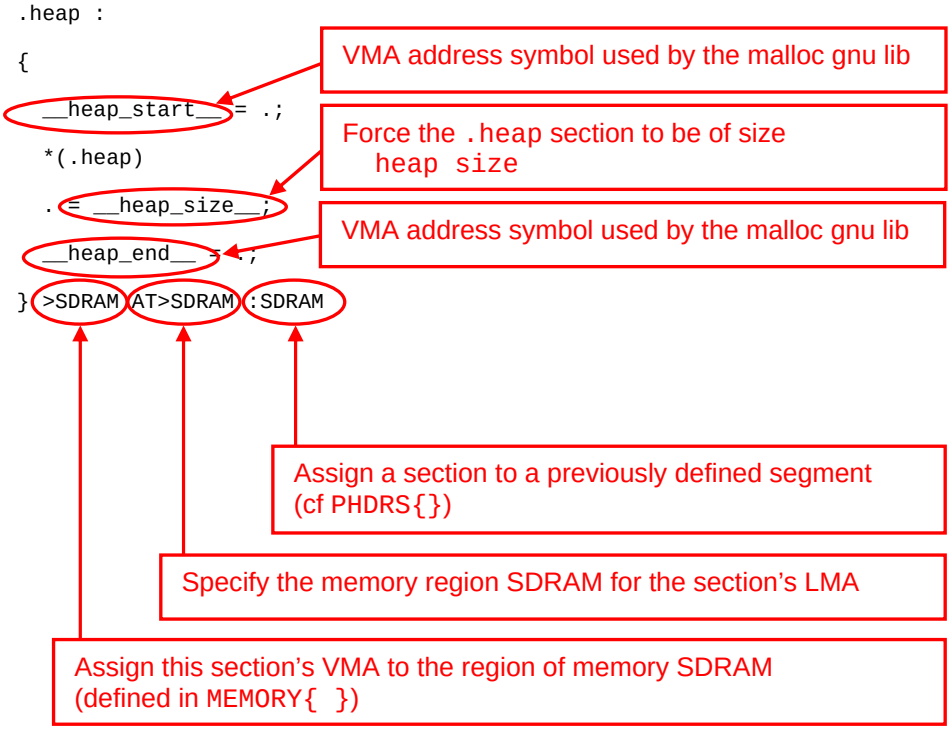
`>SDRAM`: VMA of `.data_sdram` in external SDRAM

`AT>FLASH`: LMA of `.data_sdram` (in flash)

`:SDRAM_AT_FLASH`: specified as loadable in `PHDRS{}` (for the data init value)

A.4. Specifying the size and location of the heap to external SDRAM

```
.heap :
{
  __heap_start__ = .;
  *(.heap)
  . = __heap_size__;
  __heap_end__ = .;
} >SDRAM AT>SDRAM :SDRAM
```



NOTE Because the program header SDRAM was defined as PT_NULL, the .heap section won't be loaded to target. And so the AT>SDRAM is unnecessary and ignored.

B. Initialization of the SDRAM controller:

- When: before the first SDRAM access, which is performed during the startup process (for .data_sdram and .bss_sdram sections initialization)
- How: using the startup customization API (_init_startup()), which is called by the startup routine before doing the SDRAM access

C. Startup routine customization

C.1. Call the startup customization function:

call _init_startup: this is when the SDRAM controller must be initialized

C.2. .data_sdram and .bss_sdram sections initialization:

- Load initialized external SDRAM data having a global lifetime from the .data_sdram LMA section using the symbols previously defined in the linker script (_data_sdram and _edata_sdram (the VMA addresses), _data_sdram_lma (the LMA start address))

- Clear uninitialized external SDRAM data having a global lifetime in the `.bss_sdram` section using the symbols previously defined in the linker script (`_bss_sdram_start` and `_bss_sdram_end` (VMA addresses))

D. Using dynamic allocation: use malloc() and free() “as usual”

E. Assigning an initialized variable to external SDRAM:

```
__attribute__((__section__(".data_sdram")))
static int AllGoodChildrenGoToHeaven[7] = { 1,2,3,4,5,6,7 };
```

F. Assigning a non-initialized variable to external SDRAM:

```
__attribute__((__section__(".bss_sdram")))
static int HelloGoodbye;
```

6.3.2 Details (documentation and source code)

AVR32733 application note: Placing data and the heap in external SDRAM.

7 Specific linker script examples

- The AVR Software Framework provides generic linker scripts under utils/linker_scripts/ or specific linker scripts for some examples; these scripts have a .lds extension (the extension doesn't matter).
- drivers/flashc/flash_example/: Controlling the location of variables in flash
- drivers/cpu/mpu/example/: Controlling the location of a function in flash
- Application Note AVR32733: Placing data and the heap in external SDRAM

8 Frequently asked questions

Q: How can I use my own linker script instead of the default one inside a 32-bit AVR project?

A: By default, avr32-gcc uses the default linker script from the AVR GNU toolchain.

To use your own linker script in a 32-bit Atmel AVR32 Studio® project, use the following procedure:

- Copy your linker script in the root of your project (use the import command, or simply copy/paste)
- Open the Project Properties view (Properties item from the contextual menu)
- Select the Tool Settings tab
- Select the AVR32/GNU C Linker -> Miscellaneous item
- Add -T../my_linker_script.lds to the linker flags command line

Your project is now ready to link with your own linker script.

Q: How can I declare a variable at a specific location of the flash memory?

A: To do so with GCC, a specific section must be created at link time.

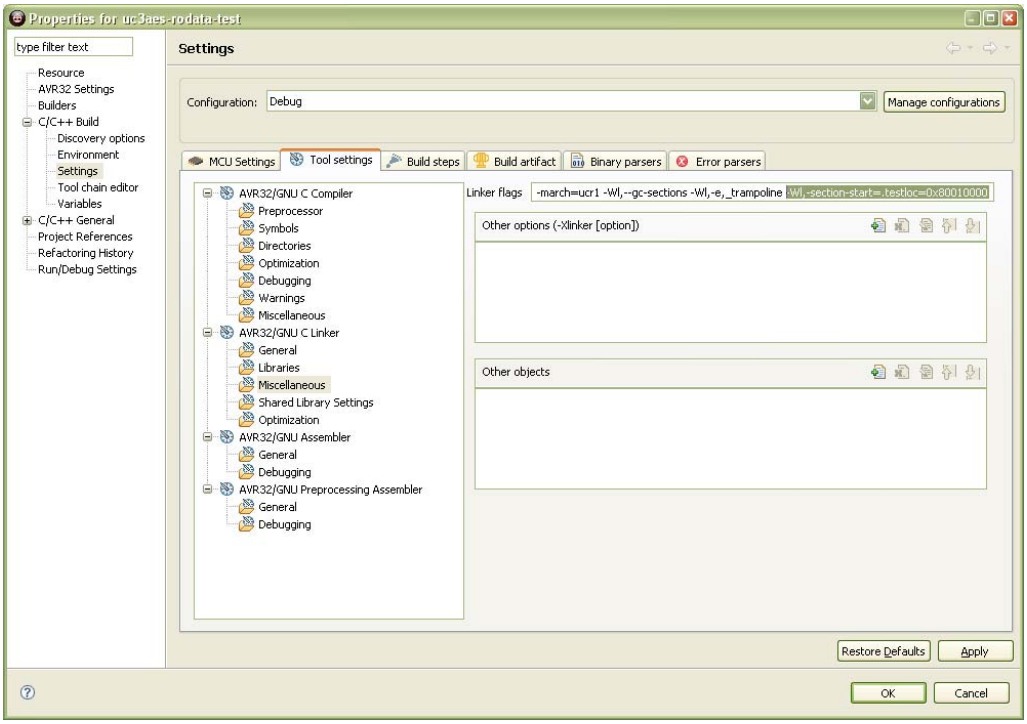
Here is an example of how to place a string variable at address 0x80010000:

1. Declare the variable, specifying location in the `.testloc` section:

```
const char string[] __attribute__ ((section (".testloc"))) = "String at fixed address";
```

2. Create the section in the linker options:
   - Open the Project Properties: highlight the project name and press Alt+Enter
   - Select the C/C++ Build / Settings category
   - Select the Tool Settings tab
   - Expand the AVR32/GNU Linker and highlight the miscellaneous item
   - In the Linker Flags field, add the -Wl,-section-start=.testloc=0x80010000 option

Refer also to the screen shot below.



Q: When compiling the ARV Software Framework examples, I got the following kinds of warnings:

ld: uc3a0512-ctrlpanel.elf: warning: allocated section '.dalign' not in segment

ld: uc3a0512-ctrlpanel.elf: warning: allocated section '.bss' not in segment

ld: uc3a0512-ctrlpanel.elf: warning: allocated section '.heap' not in segment

ld: uc3a0512-ctrlpanel.elf: warning: allocated section '.stack' not in segment

What are these warnings?

A: These are normal warnings, and you don't need to care about them. Here is an explanation of these warnings:

When using the default linker scripts provided with avr32-gcc, the ELF LOAD program headers are generated automatically from the output sections, including BSS and the stack which are only allocated areas.

In its current revision, avr32program programs allocate LOAD program headers that do not have to be filled with data from the ELF file, which wastes time. This avr32program behavior will be changed in a future release, but until this is achieved, the linker scripts provided with AVR Software Framework are modified to place the allocated-only output sections in NULL ELF program headers, which are ignored by avr32program, explaining the warnings when linking.

Q: How can I create a 32-bit AVR32 Studio project from an existing standalone project that has its own makefile and linker script?

A: Here is the step-by-step procedure to import your existing project into 32-bit AVR32 Studio and reuse the makefile and linker scripts of the project:

1. Create an empty project:
   - Open the New wizard selection: menu File -> New -> Other
   - Expand the C folder and highlight the AVR32 C project (Make) item
   - Click Next to open the New Project wizard
   - Select the Target MCU from the list, enter a project name, and click Finish
2. Add the existing source code and makefile:
   - Open the New wizard selection again: menu File -> New -> Other
   - Expand the General folder and highlight the Folder item
   - Click Next to open the New Folder wizard
   - Click on Advanced>>, and check the Link to folder in the file system box
   - Browse to the location of your existing stand-alone project, and click Finish
3. Create a make target:
   - In the Project Explorer view, browse to the folder that contains the makefile
   - Right-click on the makefile file, and select the Create make target item
   - Enter a name in the Target Name: field (for example, Build), and click Create

NOTE Other targets could be added by repeating the above steps.

NOTE The make target creations depend on the keywords defined in the makefile:

4. Build the make target:
   - In the Project Explorer view, right-click on the project name, and select the Build make target item
   - Select the make target to build, and click Finish

9 References

The official GNU ld linker documentation:
http://sourceware.org/binutils/docs-2.19/ld/index.html
http://sourceware.org/binutils/docs-2.19/ld/Scripts.html#Scripts

Using the GNU Compiler Collection:
http://gcc.gnu.org/onlinedocs/: online HTML or PDF document

10 Support

Atmel has several support channels available:

- Web portal: http://support.atmel.no/ All Atmel microcontrollers
- Email: avr@atmel.com All AVR products
- Email: avr32@atmel.com All 32-bit AVR products

Please register on the web portal to gain access to the following services:

- Access to a rich FAQ database
- Easy submission of technical support requests
- History of all past support requests
- Register to receive Atmel microcontroller newsletters

11 Table of Contents

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Milennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chou-ku, Tokyo 104-0033
JAPAN
Tel: (+81) 3523-3551
Fax: (+81) 3523-7581