Posted by Stargirl Flowers on January 13, 2021 · view all posts

The most thoroughly commented linker script (probably)

While developing the firmware for Winterbloom's Castor & Pollux, I got very curious as to just what the Microchip/Atmel-provided linker script was doing.

If you've never heard of or seen a linker script before you're not alone. Most of us never even have to think about them, however, on memory constrained embedded devices it's not uncommon to need to modify the default linker script.

The linker script controls how `ld` combines all of your `.o` files into a single `.elf` and how that resulting `.elf` file gets loaded by the target processor.

So I was staring at this script that made absolutely no sense to me. It's filled with incantations and mysterious symbols and there's no indication of what they're for or where they come from.

So I did a **lot** of research and now I can present to you **the most thoroughly commented linker script**[1].

You can see this script in its entirety, comments and all, on GitHub. But if you'd like to read it here instead it's transcribed below.

Output format

Output format sets the ELF output format to use a specific BFD backend.

The first is the default BFD. The second and third arguments are used when big (-EB) or little (-EL) endian is requested.

Since the SAM D series are configured with only little endian support, "elf32-littlearm" is used across the board. This option seems to be included by Atmel/Microchip out of an abundance of caution, as arm-none-eabi-ld will do the right thing and use "elf32-littlearm" by default.

The list of acceptable values can be obtained using `objdump -i`.

References:

- https://sourceware.org/binutils/docs/ld/Format-Commands.html#Format-Commands
- https://sourceware.org/binutils/docs/ld/BFD.html
- https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.p
  Section 11.1.11, Cortex M0+ Configuration

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")                c
```

CPU memory configuration variables

These variables are used by the following "MEMORY" command to define the various memory spaces.

For the SAMD21G18A used by this project, the available Flash is 262kB and the available SRAM is 32kB.

This project also reserves 8kB for the bootloader and 1kB for "non-volatile memory" (NVM) - which is used by the application to store calibration and user settings.

References:

- https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.p Section 10.2, Physical Memory Map

```c
FLASH_SIZE = 0x40000;       /* 256kB */
BOOTLOADER_SIZE = 0x2000;   /* 8kB */
NVM_SIZE = 0x400;           /* 1kbB */
SRAM_SIZE = 0x8000;         /* 32kB */
```

ARM Cortex-M processors use a descending stack and generally require stack space to be set aside in RAM.

The application's behavior determines just how much stack space should be reserved. I generally start with 2kB (0x800) of stack space for Cortex-M0+ projects programmed in C .

You can analyze stack usage in GCC using the `-fstack-usage` flag and you can enable compiler warnings for stack usage with `-Wstack-usage=STACK_SIZE`.

References:

- https://embeddedartistry.com/blog/2020/08/17/three-gcc-flags-for-analyzing-memory-usage/
- https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/how-much-stack-memory-do-i-need-for-my-arm-cortex--m-applications
- https://gcc.gnu.org/onlinedocs/gnat_ugn/Static-Stack-Usage-Analysis.html
- https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

```c
STACK_SIZE = DEFINED(__stack_size__) ? __stack_size__ : 0x800;
```

Memory space definition

This section declare blocks of memories for specific purposes. Since an ARM's address space is generally split between Flash, SRAM, peripherals, and other regions, it's necessary to tell the linker where different types of data can go in the address space.

These blocks will be used in the `SECTIONS` command below.

References:

- https://sourceware.org/binutils/docs/ld/MEMORY.html#MEMORY
- https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.p Section 10.2, Physical Memory Map

```c
MEMORY
{
```

Start with the Flash memory region. On the SAMD21, Flash starts at the beginning of the address space (`0x00000000`) and is contiguous right up to the size of the Flash. Flash is marked a `rx` so that the linker knows that this space is read-only (`r`) and executable (`x`).

The "bootloader" section allows this firmware to work with the uf2 bootloader. The bootloader takes the first 0x2000 bytes of flash memory.

References:

- https://github.com/adafruit/uf2-samdx1#configuration

```
    bootloader (rx) : ORIGIN = 0x00000000, LENGTH = BOOTLOADER_SIZE                c
```

Following the bootloader is the flash memory used by the application, called "rom" here - even though it's flash, the name is just a name and doesn't carry special meaning.

The total length of the rom block is the MCU's flash size minus the bootloader's size and any space reserved for "non-volatile memory" by the application.

```
    rom (rx) : ORIGIN = 0x00002000, LENGTH = FLASH_SIZE - BOOTLOADER_SIZE - NVM_SIZE c
```

The "nvm" block is space set aside for the application to store user settings and calibration data in the MCU's flash.

The block is located right at the end of the flash space. This is useful because it means that it says in a fixed location regardless of how much flash space the application takes up in "rom". Explicitly defining this section also lets the linker ensure that application code doesn't overwrite the data in this region.

This block is marked as read-only (`r`) because flash can not be written in the same way as normal memory, however, the application can use the SAMD's NVM peripheral to write data in this region.

```
    nvm (r) : ORIGIN = FLASH_SIZE - NVM_SIZE, LENGTH = NVM_SIZE                    c
```

The "ram" block is mapped to the CPU's SRAM and it's where the stack, heap, and all variables will go.

For the SAMD21, SRAM starts at 0x20000000 and is contiguous for the size of the SRAM.

```
    ram (rwx) : ORIGIN = 0x20000000, LENGTH = SRAM_SIZE                            c
}
```

Sections

The sections command tells the linker how to combine the input files into an output ELF and where segments belong in memory.

The linker takes a set of input files containing the "input sections" and uses this to map them to "output sections" which are placed in the output ELF file.

While the most important sections to think about here are the ones that'll be placed into the memory (segments) some sections are just placed in the output ELF for debugging.

References:

- https://sourceware.org/binutils/docs/ld/SECTIONS.html#SECTIONS

```
SECTIONS                                                                    c
{
```

The text segment contains program code and read-only data.

References:

- https://developer.arm.com/documentation/dui0101/a/ Page 5, Segments
- http://www.sco.com/developers/gabi/latest/ch4.sheader.html#special_sections

```
    .text :                                                                 c
    {
```

This segment must be 4-byte aligned as defined in ARM ELF File Format specification.

```
        . = ALIGN(4);                                                       c
```

The vector table defines the initial stack pointer and interrupt/exception routines for the ARM CPU and device peripherals. Every Cortex-M project needs this.

For the SAM D series the vector table is expected to be at address 0x00000000 after reset. Since flash memory starts at 0x00000000, the first values in flash should be the vector table.

When defining the vector table in code you must use `__attribute__ ((section(".vectors")))` to tell GCC to place the vector table into the section named ".vectors" in the input object file so that the linker can find it.

Note that since this project uses the UF2 bootloader, this actually gets placed at the beginning of the program's flash area (0x2000). The Cortex-M allows changing the vector table after initialization, so the startup script sets the Vector Table Offset Register (`SCB->VTOR`) to `_sfixed` during its intialization. The `_efixed` symbol is unused but included for completeness.

References:

- https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.p Secion 8.3.3, Fetching of Initial Instructions
- https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf Section B1.5.3, The vector table Section B3.2.5, Vector Table Offset Register, VTOR
- startup_samd21.c

```
        _sfixed = .;                                                        c
        KEEP(*(.vectors .vectors.*))
```

Include code and read-only data sections from all input files.

By default, GCC places all program code into a section named ".text" and read-only data (such as const static variables) into a section named ".rodata" in the input object files. This naming convention is from the ELF ABI specification.

GCC generates three "flavors" of sections in object files:

- `.{section}`: the basic section.
- `.{section}.*`: sections generated by `-ffunction-sections` and `-fdata-sections` so that each function/data has a unique section.
- `.gnu.linkonce.{type}.*`: sections generated by GCC so the linker can remove duplicates. Seems to be related to Vague Linking.

References:

- http://www.sco.com/developers/gabi/latest/ch4.sheader.html#special_sections
- https://gcc.gnu.org/onlinedocs/gcc/Vague-Linkage.html
- https://stackoverflow.com/questions/5518083/what-is-a-linkonce-section

```
    *(.text .text.* .gnu.linkonce.t.*)                                    c
    *(.rodata .rodata* .gnu.linkonce.r.*)
```

C & C++ runtime support

The following sections are for the C/C++ runtime. These are generally used by crt0.

References:

- https://en.wikipedia.org/wiki/Crt0

Initializers

- C++ Runtime: initializers for static variables.
- C Runtime: designated constructors

For C++, handles variables at file scope like this:

```
int f = some_func()
```

For C, handles functions designated as constructors:

```
void intialize_thing(void) __attribute__((constructor));
```

Executed by the C runtime at startup via `__libc_init_array`.

References:

- https://github.com/gcc-mirror/gcc/blob/master/libgcc/crtstuff.c
- https://sourceware.org/git/?p=newlib-cygwin.git;a=blob;f=newlib/libc/misc/init.c;

- https://gcc.gnu.org/onlinedocs/gccint/Initialization.html
- https://developer.arm.com/documentation/dui0475/h/the-arm-c-and-c---libraries/c---initialization--construction-and-destruction
- https://stackoverflow.com/questions/15265295/understanding-the-libc-init-array

```c
    . = ALIGN(4);
    KEEP(*(.init))
    . = ALIGN(4);
    __preinit_array_start = .;
    KEEP (*(.preinit_array))
    __preinit_array_end = .;

    . = ALIGN(4);
    __init_array_start = .;
    KEEP (*(SORT(.init_array.*)))
    KEEP (*(.init_array))
    __init_array_end = .;
```

Finalizers

- C++ runtime: destructors for static variables.
- C runtime: designated finializers

For C, handles functions designated as destructors:

```c
void destroy_thing(void) __attribute__((destructor));
```

References:

- https://sourceware.org/git/?p=newlib-cygwin.git;a=blob;f=newlib/libc/misc/fini.c

```c
    . = ALIGN(4);
    KEEP(*(.fini))

    . = ALIGN(4);
    __fini_array_start = .;
    KEEP (*(.fini_array))
    KEEP (*(SORT(.fini_array.*)))
    __fini_array_end = .;
```

C++ runtime: static constructors

References:

- https://gcc.gnu.org/onlinedocs/gccint/Initialization.html
- https://github.com/gcc-mirror/gcc/blob/master/libgcc/crtstuff.c

```
    . = ALIGN(4);                                                    c
    KEEP (*crtbegin.o(.ctors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*crtend.o(.ctors))
```

C++ runtime: static destructors and atexit()

Note that in usual practice these aren't ever called because the program doesn't exit - except when powered off or reset.

References:

- https://gcc.gnu.org/onlinedocs/gccint/Initialization.html

```
    . = ALIGN(4);                                                    c
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*crtend.o(.dtors))

    . = ALIGN(4);
    _efixed = .;
  } > rom
```

ARM exception handling

ARM defines several special sections for exception handling.

These are require for C++ and for C programs that try to examine backtraces.

- `exidx` is used to contain index entries for stack unwinding.
- `extab` names sections containing exception unwinding information.

Essentially, each function that can throw an exception will have entries in the exidx and extab sections.

References:

- https://developer.arm.com/documentation/ihi0038/b/
- https://stackoverflow.com/a/57463515

```
  .ARM.extab : {                                                     c
     *(.ARM.extab* .gnu.linkonce.armextab.*)
  } > rom

  .ARM.exidx : {
     PROVIDE (__exidx_start = .);
     *(.ARM.exidx* .gnu.linkonce.armexidx.*)
     PROVIDE (__exidx_end = .);
  } > rom
```

Relocate

The `.relocate` section includes mutable variables that have a default value and specially marked functions that should execute from RAM.

This data is stored in ROM but is referenced from RAM. The program/runtime must copy the data from ROM to RAM on reset, hence, "relocate".

Performance sensitive/critical functions can also be placed in RAM using this section:

```c
#define RAMFUNC __attribute__((section(".ramfunc")))
void fast_function(void) RAMFUNC;
```

In other linker scripts you might see this named as the `.data` section. That's what the ELF specification calls for, but the Microchip-provided SAM D startup scripts expect `.relocate`.

This also sets the symbol `_etext` to the start of the relocation segment in flash. The startup script copies the data starting at `_etext` to `_srelocate` and ends when it reaches `_erelocate`. The `_etext` name is a bit unfortunate since it's not the end of the text segment, but rather the start of the read-only copy of the relocate section in flash. If I wrote the startup script I would have named these symbols differently.

References:

- http://www.sco.com/developers/gabi/latest/ch4.sheader.html#special_sections
- https://www.sourceware.org/binutils/docs/ld/Output-Section-LMA.html#Output-Section-LMA
- startup_samd21.c

```c
    .relocate :
    {
       . = ALIGN(4);
       _srelocate = .;
       *(.ramfunc .ramfunc.*);
       *(.data .data.*);
       . = ALIGN(4);
       _erelocate = .;
    } > ram AT> rom

    _etext = LOADADDR(.relocate);
```

BSS

The BSS section reserves RAM space for declared but uninitialized variables.

This is zeroed out by the startup script. The start-up script zeros out the area of RAM starting at `_szero` and ending at `_ezero`.

This includes `COMMON` which is a bit of a legacy section. GCC defaults to `-fno-common` these days so there shouldn't be anything in there, but it's included for completeness.

References:

- http://www.sco.com/developers/gabi/latest/ch4.sheader.html#special_sections
- https://en.wikipedia.org/wiki/.bss
- https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html Section -fcommon
- startup_samd21.c

```
.bss (NOLOAD) :                                                    c
{
   . = ALIGN(4);
   _szero = .;
   *(.bss .bss.*)
   *(COMMON)
   . = ALIGN(4);
   _ezero = .;
} > ram
```

Stack space

Cortex-M stacks grow down, so the stack starts at `_estack` and grows towards `_sstack`. The startup script sets the vector table's stack pointer to `_estack` on startup. `_sstack` is ususued but included for completeness.

The ARM procedure call standard (AAPCS) requires the stack to be aligned on an eight byte boundary.

References:

- startup_samd21.c
- https://developer.arm.com/documentation/ihi0042/e/ Section 5.2.1.2, Stack constraints at a public interface

```
.stack (NOLOAD):                                                   c
{
   . = ALIGN(8);
   _sstack = .;
   . = . + STACK_SIZE;
   . = ALIGN(8);
   _estack = .;
} > ram
```

Heap space

If the program uses the malloc and the heap, then `_heap_start` can be used as the start of the heap. If the program doesn't use the heap then the `_heap_start` symbol is unused and could be removed. With `-specs=nano.specs`, the `_sbrk` syscall has to be implemented for malloc to work:

```
extern int _heap_start;                                            c

void *_sbrk(int incr) {
    static unsigned char *heap = NULL;
    unsigned char *prev_heap;

    if (heap == NULL) {
        heap = (unsigned char *)&_heap_start;
    }

    prev_heap = heap;
    heap += incr;

    return prev_heap;
}
```

Another memory layout strategy is to place the stack at the end of RAM and the heap after bss. That way the heap can grow upwards towards the stack and the stack can grow downwards to the heap. However, I'm not a big fan of that approach- it's possible for the heap to overwrite the stack. Leaving the entire end of RAM available as the heap works well for my purposes.

References:

- https://en.wikipedia.org/wiki/Sbrk
- https://interrupt.memfault.com/blog/boostrapping-libc-with-newlib
- https://embeddedartistry.com/blog/2017/02/15/implementing-malloc-first-fit-free-list/

```c
    . = ALIGN(4);
    PROVIDE (_heap_start = .);
    _end = . ;
}
```

Absolute symbol definitions

This section defines some useful absolute symbols for the application to use.

Symbols for the settings section in the NVM memory block.

Gemini uses the first half of the NVM block for user settings. This symbol is used by the settings module to know where to load and save settings.

References:

- gem_settings_load_save.c

```c
_nvm_settings_base_address = ORIGIN(nvm);
_nvm_settings_length = LENGTH(nvm) / 2;
```

Symbols for the calibration/look-up table in the NVM memory block.

Gemini uses the other half of the NVM block to store the factory- calibrated look-up table for translating ADC -> frequency/DAC codes.

References:

- gem_voice_param_table_load_save.c

```c
_nvm_lut_base_address = ORIGIN(nvm) + LENGTH(nvm) / 2;
_nvm_lut_length = LENGTH(nvm) / 2;
```

---

1. Probably. ↩

👋 Hey, if you found this article useful I would love to hear from you. If you loved it you can consider tipping me on Ko-fi or sponsoring me on GitHub. I don't get paid for this content, so kind words and support encourage me to create more!

Stargirl Flowers
thea.codes · theacodes · theavalkyrie
Atlanta, Georgia