Get the most out of the linker map file

09 Jul 2019 by **Cyril Fougeray**

In this article, I want to highlight how simple linker map files are and how much they can teach you about the program you are working on.

Firmware engineers rarely reach for the map file generated by their build process when debugging. Yet, the answer sometimes lies in that particular file.

The map file provides valuable information that can help you understand and optimize memory. I highly recommend keeping that file for any firmware running in production. This is one of the few artifacts I keep in my CD pipeline.

The map file is a symbol table for the whole program. Let's dive into it to see how simple it is and how you can effectively use it. I will try to illustrate with some examples, all described with GNU binutils.

Generating the map file

To get started, make sure you generate the map file as part of your build.

Using GNU binutils, the generation of the map file must be explicitly requested by setting the right flag. To print the map to `output.map` with LD:

```
LDFLAGS += -Wl,-Map=output.map
```

As an example for a simple program, you would link the compilation units using those commands:

```
# Compile/assemble source files without linking
$ gcc -c sourcefile1.c sourcefile2.c
# Link together following the linker script specs and outputs map file with the link resu
$ ld -Map output.map -T linker_script.ld -N -o output.elf sourcefile1.o sourcefile2.o
```

Most compilers have an option to enable the same kind of file, `--map` for the ARM compiler for example.

Blinky

👨‍🦳 What's better than our good old friend to explain the basics of the map file?

In order to learn about map files, let's compile a simple LED-blink program, and modify it to add a call to `atoi`. We will then use the map file to dissect the differences between both programs. The main file, available here for you to play with, can be used as a replacement for the Blinky example from the nRF5 SDK. Looking at the `main.c` file, 3 configurations are possible: `NO_ATOI`, `STD_ATOI`, `CUSTOM_ATOI`.

Let's build the simplest version of that project (`NO_ATOI`), equivalent to this piece of code:

```c
#include <stdbool.h>
#include <stdint.h>
#include "nrf_delay.h"
#include "boards.h"

/**
 * @brief Function for application main entry.
 */
int main(void)
{
    /* Configure board. */
    bsp_board_init(BSP_INIT_LEDS);

    /* Toggle LEDs. */
    while (true)
    {
        for (int i = 0; i < LEDS_NUMBER; i++)
        {
            bsp_board_led_invert(i);
            nrf_delay_ms(300);
        }
    }
}
```

Compiling:

```
    $ make
    Assembling file: gcc_startup_nrf52840.S
    Compiling file: boards.c
    Compiling file: main.c
    Compiling file: system_nrf52840.c
    Linking target: _build/nrf52840_xxaa.out
       text        data     bss     dec     hex filename
       1704         108      28    1840     730 _build/nrf52840_xxaa.out
    Preparing: _build/nrf52840_xxaa.hex
    Preparing: _build/nrf52840_xxaa.bin
    DONE nrf52840_xxaa
```

The generated map file is 563-line long 😮, even though it does nothing more than blink LEDs. That many lines cannot be left unseen, there must be some serious information in there…

Now let's modify our program to add a call to `atoi`. Instead of using an integer directly for the delay, we'll encode it as a string and decode it with `atoi`. Here is the code using the configuration `STD_ATOI`:

```c
#include <stdbool.h>
#include <stdint.h>
#include "nrf_delay.h"
#include "boards.h"
#include "stdlib.h"

static const char* _delay_ms_str = "300";

/**
 * @brief Function for application main entry.
 */
int main(void)
{
    /* Configure board. */
    bsp_board_init(BSP_INIT_LEDS);
    int delay = atoi(_delay_ms_str);

    /* Toggle LEDs. */
    while (true)
    {
        for (int i = 0; i < LEDS_NUMBER; i++)
        {
            bsp_board_led_invert(i);
            nrf_delay_ms(delay);
        }
    }
}
```

After compilation, the whole program goes from 1840 bytes to 2396 bytes.

```
       text        data     bss     dec     hex filename
       2256         112      28    2396     95c _build/nrf52840_xxaa.out
```

We expected more code to come with calling `atoi`, but a 30% increase in our program size is huge! 🤔

Now that I have two map files, I want to know the differences between the two.

Digging into the map files

You can check the differences in this file: std_atoi_map.diff. In the following parts, I'll use snippets to explain the different sections of the map file.

Archives linked

What? An archive in my binary? I'm only blinking an LED!

Don't be afraid, the good news is that the map file does mention the specific symbols which caused the archive members to be brought in, in the first lines of the file.

Here are my first lines:

```
    Archive member included to satisfy reference by file (symbol)

    /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/arm-none-eabi/
                              /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc
```

The format is:

```
    The archive file location (compilation unit)
                        The compilation unit referencing the archive (symbol called)
```

It means that a `crt0` file is calling the `exit` function from `exit.o` included in `libc_nano.a`.

The reason is not in the scope of this article, but there are indeed standard libraries that are provided by your toolchain (here the GNU toolchain). Those are available to provide standard functions such as `atoi`. In that example, I specified to the linker to use the `nano.specs` file. That's why standard functions all come from `libc_nano.a`. You can read more about this on lb9mg.no.

Now, comparing the two generated map files, the first difference spotted is that some other archive members are included in the program: `atoi`, which itself needs `_strtol_r` which itself needs `_ctype_`:

```
    /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/arm-none-eabi/
                              _build/nrf52840_xxaa/main.c.o (atoi)
    /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/arm-none-eabi/
                              /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc
    /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/arm-none-eabi/
                              /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc
```

We now have a better sense of the files actually included into our program, and the reason why they are there. Let's explore what else is inside the file!

Memory configuration

The most straightforward pieces of information in the map file are the actual memory regions, with location, size and access rights granted to those regions:

```
    Memory Configuration

    Name             Origin             Length             Attributes
    FLASH            0x0000000000001000 0x00000000000ff000 xr
    RAM              0x0000000020000008 0x000000000003fff8 xrw
    *default*        0x0000000000000000 0xffffffffffffffff
```

Linker script and memory map

Following the memory configuration is the **Linker script and memory map**. That one is interesting as it gives detailed information about the symbols in your program. In our case, it first indicates the `text` area size and its content (`text` is our compiled code, as opposed to `data` which is program data). Here, the interrupt vectors (under the section `.isr_vector`) are present at the beginning of the executable, defined in `gcc_startup_nrf52840.S`:

```
    Linker script and memory map

    .text           0x0000000000001000      0x8c8
     ***(.isr_vector)**
     .isr_vector    0x0000000000001000      0x200 _build/nrf52840_xxaa/**gcc_startup_nrf5
                    0x0000000000001000                __isr_vector
     *(.text*)
     .text          0x0000000000001200       0x40 /usr/local/Cellar/arm-none-eabi-gcc/8-2
     .text          0x0000000000001240       0x74 /usr/local/Cellar/arm-none-eabi-gcc/8-2
                    0x0000000000001240                _mainCRTStartup
                    0x0000000000001240                _start
     .text          0x00000000000012b4       0x3c _build/nrf52840_xxaa/gcc_startup_nrf528
                    0x00000000000012b4                Reset_Handler
                    0x00000000000012dc                NMI_Handler

    [...]

     .text.**bsp_board_led_invert**
                    0x00000000000012f0       **0x34** _build/nrf52840_xxaa/**boards.c.o**
                    0x00000000000012f0                bsp_board_led_invert
```

Those lines give us the address of each function and its size. Above, you can read the address of `bsp_board_led_invert`, coming from `boards.c.o` (compilation unit of `board.c` as you guessed) which has a size of 0x34 bytes in the `text` area. That way, we are able to locate each function used in the program.

My constant string `_delay_ms_str` is obviously included in the program as it is initialized. Read-only data are saved as `rodata` and kept in the `FLASH` region as specified in the linker script (stored in Flash and not copied in RAM as it is constant). I can find it under that line:

```
    .rodata.main.str1.4
                      0x00000000000017b8        0x4 _build/nrf52840_xxaa/main.c.o
```

I also noticed that the inclusion of `_ctype_` added 0x101 bytes of read-only data in the `text` area 😯.

```
    .rodata._ctype_
                    0x00000000000017c0      **0x101** /usr/local/Cellar/arm-none-eabi-gcc
                    0x00000000000017c0                **_ctype_**
```

As standard libraries are open-source (link), we can easily find out why it's taking that much space. I dived into the inner-working of `atoi` (`atoi_r` in its **reentrant** version, see below), which calls directly `strtol_r`:

```c
int
_DEFUN (_atoi_r, (s),
  struct _reent *ptr _AND
  _CONST char *s)
{
  return (int) _strtol_r (ptr, s, NULL, 10);
}
```

When it comes to `strtol_r`, it is actually more complex than only converting characters into an integer because if performs type checking, using `ctype`. The way `ctype` works is by using a table where the ASCII symbol types are stored into an array. Here are the main parts of `ctype`, annotated with my comments:

```c
// Flags indicating the symbol type
#define _U  01    // Uppercase letter
#define _L  02    // Lowercase letter
#define _N  04    // Numeric
#define _S  010
#define _P  020   // Ponctuation
#define _C  040   // Control characters
#define _X  0100  // Hexadecimal
#define _B  0200

// ASCII table types
#define _CTYPE_DATA_0_127 \
  _C, _C, _C, _C, _C, _C, _C, _C, \
  _C, _C|_S, _C|_S, _C|_S,  _C|_S,  _C|_S,  _C, _C, \
  _C, _C, _C, _C, _C, _C, _C, _C, \
  _C, _C, _C, _C, _C, _C, _C, _C, \
  _S|_B,  _P, _P, _P, _P, _P, _P, _P, \
  _P, _P, _P, _P, _P, _P, _P, _P, \
  _N, _N, _N, _N, _N, _N, _N, _N, \
  _N, _N, _P, _P, _P, _P, _P, _P, \
  _P, _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U, \
  _U, _U, _U, _U, _U, _U, _U, _U, \
  _U, _U, _U, _U, _U, _U, _U, _U, \
  _U, _U, _U, _P, _P, _P, _P, _P, \
  _P, _L|_X,  _L|_X,  _L|_X,  _L|_X,  _L|_X,  _L|_X,  _L, \
  _L, _L, _L, _L, _L, _L, _L, _L, \
  _L, _L, _L, _L, _L, _L, _L, _L, \
  _L, _L, _L, _P, _P, _P, _P, _C

// _ctype_ structure takes 257 (0x101) bytes, as seen on the map file (.rodata._ctype_)
_CONST char _ctype_[1 + 256] = {
  0,
  _CTYPE_DATA_0_127,
  _CTYPE_DATA_128_255
};
```

Interestingly, the addition of `atoi` not only increased our code size (the `text` area), but also our data size (the `data` area). With the diff file, I can easily discover data that was before discarded by the linker:

```
    .data._impure_ptr
                                                            0x0000000000000000
```

Now you may have noticed the function names finishing by `_r`, when calling `strtol_r` for example. That suffix indicates reentrancy. Documentation about reentrancy can be found in newlib source code. To sum-up, reentrant functions can be called even when the same function is already in execution in another process, without intervening the execution. From the documentation:

```
Each function which uses the global reentrancy structure uses the global
variable _impure_ptr, which points to a reentrancy structure.
```

In our case, we need that new global variable to call the reentrant function: `atoi_r`.

One last piece of information to keep in mind: initialized variables have to be kept in Flash but they appear in `RAM` in the map file because they are copied into RAM before entering the `main` function. Here, symbols `__data_start__` and `__data_end__` keep track of the area used in RAM to keep initialized variables. Those values are stored in Flash starting at `0x00000000000018d0`:

```
    .data           0x0000000020000008       0x70 load address 0x00000000000018d0
                    0x0000000020000008                __data_start__ = .
    [...]
                    0x0000000020000078                __data_end__ = .
```

Discarded sections

Functions and variables that are compiled to be included into the program aren't always part of the final binary if the linker doesn't find any reference to them. They are removed but still appear in the map file under the `Discarded input sections` part. As an example, here are some functions defined in `boards.c` that are never called and thus discarded:

```
 .text.bsp_board_led_state_get
                0x0000000000000000       0x28 _build/nrf52840_xxaa/boards.c.o
 .text.bsp_board_led_on
                0x0000000000000000       0x24 _build/nrf52840_xxaa/boards.c.o
```

Common symbols

That section does not appear in our map file but it deserves its own paragraph.

Common symbols are non-constant global variables that can be used everywhere in your code. You might know that using global variables is usually not a good practice as they make the code harder to maintain. Indeed, the scope is global and each extern module can modify the value of any global variable, which must be taken into account when accessed. Isolation of variables into a module, using the `static` keyword, is often better to make sure the module creating the variable is entirely responsible of its state.

Now if you want to make your program safer and prevent accessibility of some global variables, go take a look into that map file section. If some variables don't need to be declared as global, you might want to convert them to static variables.

Several usages of the map file are possible. Most of the time, you will have an address and you will want to resolve the function behind. It can be the Program Counter in the Hard Fault handler for example. Some other times you will be debugging some undefined behavior to finally find out that your program is accidentally writing into an out-of-bounds array. Whenever you have the ELF file, `arm-none-eabi-nm` is pretty useful for those things too, and it comes with options to sort symbols by size, check out this article from François.

But some other times, it will be useful even before you have an executable ready…

Debugging a linking error

Map files are created while the built code (`.o` files) is linked together which means it can be used to resolve errors in the linking process. I remember working on a bootloader that was contained in a few Flash pages. At some point, I wanted to use `atoi` but the bootloader didn't compile anymore because there was no more space available.

Using the previous example, let's say I now have only 0x800 bytes of Flash. Compiling the first example, without `atoi`, there are no issues. The second example however would overflow our small Flash:

```
    $ make
    Compiling file: main.c
    Linking target: _build/nrf52840_xxaa.out
    /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/arm-none-eabi/
    /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/arm-none-eabi/
    /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/arm-none-eabi/
    collect2: error: ld returned 1 exit status
    make: *** [_build/nrf52840_xxaa.out] Error 1
```

That's annoying! `atoi` is such a simple function… But as we have seen, it takes more Flash than we would expect using `libc_nano.a`.

Let's try to to implement our own version of `atoi`, it's not that difficult after all. Here is the result after compilation (config `CUSTOM_ATOI`):

```
    Linking target: _build/nrf52840_xxaa.out
       text        data     bss     dec     hex filename
       1740         108      28    1876     754 _build/nrf52840_xxaa.out
```

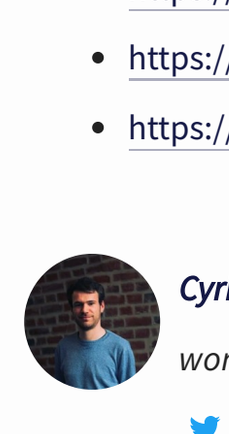
Way better! The code can now be crammed into 0x800 bytes to meet our (fake) requirements 🥳.

Reading the map file will teach you a lot about the code you are working on, and it is the first step to better Firmware. Some tools are available to parse map files and have a summarized view of your program. I should mention Adafruit's, using that source code, for example.

Feel free to share your story with any interesting usage of the map files.

Sources

- https://www.embeddedrelated.com/showarticle/900.php
- https://stackoverflow.com/questions/755783/whats-the-use-of-map-files-the-linker-produces
- https://sourceware.org/binutils/docs/ld/Options.html#index-_002d_002dprint_002dmap
- https://www.oreilly.com/library/view/programming-embedded-systems/0596009836/ch04.html



*Cyril Fougeray is a freelance embedded software engineer you can hire **through that page**. Previously he worked on embedded software at Equisense and Spire.*

Start Discussion 0 replies