Our BBC micro:bit v2 linker script

James Geddes

August 28, 2024

1 Overview

This file explains (and is also the source of) the linker script we have developed for the micro:bit version 2, at least as far as we have developed it, which is currently not very far.

To extract (or "tangle," in literate-programming parlance) the script itself, run `make` in this directory. That probably won't work on your machine, but it's a start . . .

So far, we have discovered that we need to describe the memory layout of the microbit and also to explain to the linker where in memory to put various elements of the program (known as "sections"). The memory layout is largely determined by the design of the Arm Cortex M4 CPU with some details specific to the Nordic nRF528333 system-on-a-chip which contains that CPU.

2 The linker script

Our overall script, in the file `microbit-v2.ld`, is:

```
/*
    BBC micro:bit v2 (woefully incomplete) linker script
    Written by the Hut23 Compiler Club
*/
<<Memory layout>>
<<Input and output sections>>
```

The following sections describe each part in more detail.

3 Memory layout

The Cortex M4 is a 32-bit machine, meaning that it can in principle address 2^{32} bytes, or 4 GB, of memory. However, the nRF52833 only has 512 kB of flash memory and 128 kB of RAM. These are mapped to the 4 GB of "virtual" memory space as described in the nRF52833 product specification (section 4.2.3, figure 3).

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 512K
  RAM (wx)   : ORIGIN = 0x20000000, LENGTH = 128K
}
```

Listing 1: Memory layout of the Nordic nRF528333

There are lots of other bits of memory – for example, the memory-mapped peripheral registers – but (I think) we don't need to write them down here because the linker doesn't need to arrange data to fit in those areas. Instead, they are at fixed, known, locations, and therefore we can specify them in header files. (On the other hand, perhaps they should go here?)

4 Input and output sections

The content (either code or data) of the linker's input and output files are assigned to regions called "sections." One job of the linker is to map the input sections to the output sections and to write in the output file where the output sections are to end up in memory.

There are three conventional sections which are produced, for example, by the C compiler. These are `.text` (which holds code), `.data` (which holds "initialised data", which starts with a given value), and `.bss` (which holds "uninitialised data", which starts off as zero). The reason for a `.bss` section is that the final output file can be made smaller by not storing in it the actual zeros, merely noting how much space they will need. When the program starts, one of its first jobs will be to actually zero the memory locations (and we will have to write code to do this).

The script above gathers together all `.text` sections in all input files (that's what `*(.text)` means) and groups them together in a single output section, also labelled `.text`, which is itself to be placed in the flash memory. In other words, code in the input file will end up written into memory from 0x0 onwards.

```
  /* This is all wrong */
  SECTIONS {
  .text : { *(.text) } >FLASH
}
```

Listing 2: Input and output sections

This script is wrong for lots of reasons. One is that in fact the first part of memory has to hold data that the CPU will use. On reset, the CPU loads the stack pointer with the address found in the four bytes starting at 0x0, and then commences executing code at the address in the four bytes starting at 0x4. In addition, the memory from 0x8 up possibly to 0x400 should contain pointers to interrupt handlers. (That's how the CPU knows what code to run when interrupts happen.)