Developing STM8 boot code with SDCC

Apr 18, 2017

I'm using the open source SDCC toolchain to develop an application for the STM8 microcontroller and part of that requires a custom bootloader (what ST's manuals refer to as User Boot Code or UBC) and application firmware. Here are some notes on how to use SDCC and stm8flash to develop and flash the bootloader and application.

The UBC concept itself is mostly a convention on STM8. The hardware does not do much with it aside from treating the UBC area of flash as write-protected (the idea is that boot code is not field-upgradeable in a typical product whereas we may wish to reflash the application firmware).

Boot process and interrupts

The STM8 uses option byte 1 to determine the size of the UBC (it's 0 by default meaning there is no boot code). Setting this to a non-zero size reserves a portion of flash (starting at `0x8000`) for the UBC. For example setting byte 1 to 4 reserves four 256 byte blocks or 1KB for the UBC.

It is up to the boot code to jump to the application. The STM8 CPU assumes that the interrupt vector table is located at `0x8000` so this single table must be shared between the UBC and the application.

Interrupt vector table

In SDCC, interrupt vectors look like:

```
void usart1_rx_irq(void) __interrupt(28)
{
}
```

Interrupt vectors must be implemented in the same translation unit (file) that implements `main()` and the `__interrupt` attribute is used to specify their IRQ number (which becomes an offset into the interrupt vector table).

The interrupt vector table is placed at the start of the program (`0x8000` by default, or whatever is set by using the `--code-loc` option). The UBC will be placed at `0x8000` along with its vector table but the application needs to be placed after the UBC (starting with its own vector table). So for a 1KB UBC (that is, option byte 1 is set to 4) we would build the application firmware with `--code-loc=0x8400` and we know that the interrupt vector table for the application is at `0x8400`.

The STM8 manual shows the offsets from the start of the vector table for each interrupt handler. For the above interrupt 28, the offset is 0x7b. Assuming the boot code does not need to do anything with interrupt 28, we could simply redirect to the application firmware's implementation of that interrupt handler. That is, in the UBC we would have something like:

```
void usart1_rx_irq(void) __interrupt(28)
{
  __asm jpf 0x8478 __endasm;
}
```

To connect the real interrupt 28 to the redirected handler in the application's table. The application's implementation of interrupt 28 would do whatever it is that is appropriate for handling that interrupt.

The UBC should redirect every single interrupt to the right location to provide equivalent functionality in the application. The table itself contains 4-byte entries:

- 0x00: the reset vector
- 0x04: trap handler
- 0x08: interrupt 0 (unused)
- 0x0C: interrupt 1 (FLASH)
- 0x10: interrupt 2 (DMA 0/1)

...and so on. As such, interrupt 28 is `4 * 28 + 8` or `120` which gives us the offset `0x78` and, if the application starts at `0x8400` the redirected vector is at `0x8478`.

We can calculate some offsets and addresses in code (or the preprocessor) to make life easier.

stm8flash

To flash the MCU:

- get the current option bytes content from the MCU
- modify that content to set the UBC size (and any other changes needed)
- write back the modified option bytes
- write the bootloader
- write the application

Using the STM8 discovery board (with STM8L151), we can read the option bytes from the MCU,

```
stm8flash -c stlink -p stm8l151c6 -s opt -r opt.bin
```

Then edit `opt.bin` as needed. Note that byte 0 must always be set to `0xAA` to keep the SWIM protocol usable. To write it back

```
stm8flash -c stlink -p stm8l151c6 -s opt -w opt.bin
```

To write the bootloader, `boot.ihx` to the default location (0x8000):

```
stm8flash -c stlink -p stm8l151c6 -w boot.ihx
```

And then to write the application to (for example) 0x8400:

```
stm8flash -c stlink -p stm8l151c6 -s 0x8400 -w fw.ihx
```

Self-programming the Flash

Warning: this is going to get very hacky!

One of the typical tasks of a bootloader is accepting a new application to write into the application section of the Flash. There isn't much Flash on a typical STM8 microcontroller so we're likely to implement a scheme that involves:

- the application is told to jump to boot code and accept a firmware update
- the boot code starts and, rather than jumping to the application, waits for new firmware
- new firmware is received and written to the Flash, block by block
- having completed the reflashing process, the boot code is told to boot the new application

That's generally easy enough to implement but on STM8 (and many other parts) an efficient block-oriented Flash operation requires us to be executing from RAM rather than in-place in the Flash. This requires us to place the actual code that performs the erase and write operation into RAM as well as the block of data that we wish to write.

Unlocking the Flash

The STM8 program flash is unlocked with the following sequence:

```
FLASH->PUKR = FLASH_RASS_KEY1;
FLASH->PUKR = FLASH_RASS_KEY2;
while (!(FLASH->IAPSR & FLASH_FLAG_PUL));
```

The Flash can be erased and written after that.

Calling a RAM function

Unfortunately SDCC is missing linker features to help us do this. We cannot tell the linker to place a routine in RAM and, to make matters worse, we cannot use a linker-derived symbol in our C code to implement a simple `memcpy()` of the function in question from Flash to RAM, nor can we learn the length of the function (all things that GCC and proprietary stm8 toolchains can do).

I decided to work around this limitation by taking my own hacky approach:

- write a dummy program that implements my RAM function, `flash_write_block` and compile it with SDCC like any other C program
- use the assembly output to locate the implementation of `flash_write_block` and retrieve the sequence of bytes (machine instructions) that make this function.
- save those bytes as a C array and include it in my bootloader program as static data.
- at boot, `memcpy()` that array to a location in RAM and call that location. This causes `flash_write_block()` to execute in RAM and then return.

We need a location for the "array" in RAM to jump to. SDCC provides an `__at` attribute to enable us to place a variable at a set location. The static array will have an underscore in front of its name (by my convention) so I decided on

```
__at(0x4400) char _flash_write_block_ram[sizeof(_flash_write_block)];
```

This function needs to know two things:

- the location of a block of data to write (128 bytes on my STM8 target)
- the destination to write to

The data to write must also be in RAM so I selected a fixed location in RAM to hold that block using SDCC's `__at` attribute:

```
__at(0x380) char data[128];
```

I decided to make the destination a "block number" where 0 is the first block of application firmware. My application starts after the UBC, for example at `0x8400` so block 0 is address `0x8400` and block 1 is `0x8480` (the next block). I can push this to the stack (as an argument to `flash_write_block()` or, since the data is in RAM anyway, we can make a RAM location to store this as well:

```
__at(0x37C) uint32_t block;
```

In the bootloader's `main()`, we simply copy from Flash to RAM:

```
memcpy((void *)0x4400, _flash_write_block_ram, sizeof(_flash_write_block));
```

then, whenever we need to call `flash_write_block()` in RAM, we simply write the data block and block number to the defined locations and call. For example,

```
block = 0; /* Write to the 0th block of the application */
__asm call 0x4400 __endasm
```

...and whatever is in `data` will be written to `0x8400`.

Implementing the RAM function

The STM8 reference manual describes several ways to program the Flash. I chose the following:

- erase the target block
- wait for Flash operation to finish
- request "fast" block programming (we'll program all 128 bytes, and the block was already erased)
- write all 128 bytes to the target block
- wait for Flash operation to finish

The data to write is at arbitrary RAM location `0x380` and the block number is at `0x37C`. We also need a loop counter for later (and SDCC does not support variable declarations mixed with code).

```
#include "stm8l15x.h"

#define BOOT_SIZE   0x400 /* This must match the UBC setting as well */
#define APP_BASE    (0x8000 + BOOT_SIZE)

void flash_write_block(void)
{
    unsigned i;
    uint32_t block = *(uint32_t *)0x37C;
    uint32_t addr = APP_BASE + (FLASH_BLOCK_SIZE * block);
    uint8_t *dest = (uint8_t *)(addr);
    uint8_t *data = (uint8_t *)0x380;
```

We now know where to write so let's start by erasing the block. This is done by requesting an erase operation and then writing `0` to the first word in the block:

```
    FLASH->CR2 |= FLASH_CR2_ERASE;
    *((uint32_t *)addr) = 0;
```

We then wait for the operation to finish:

```
    while (FLASH->IAPSR & (FLASH_IAPSR_EOP | FLASH_IAPSR_WR_PG_DIS));
```

Now we can request block programming (specifically the "fast" version):

```
    FLASH->CR2 |= FLASH_CR2_FPM;
```

To program, we simply copy `data` to `dest` (we can't call `memcpy()` since it is not in RAM):

```
    for (i = 0; i < FLASH_BLOCK_SIZE; i++)
        dest[i] = data[i];
```

That should do it! Now wait for the operation to finish:

```
    while (FLASH->IAPSR & (FLASH_IAPSR_EOP | FLASH_IAPSR_WR_PG_DIS));
}
```

and we are done. SDCC will label this function `_flash_write_block` in the assembly output. In my hacky scheme, I need a header file such as `ramfunc.h` that has something like:

```
#pragma once

static char _flash_write_block[] = { /* the instructions */ };
```

to make this whole thing work.

Retrieving instructions for RAM

Normally we could identify where in a binary the target function is implemented and indeed SDCC lets us know through the linker map file. That said, the linker does not make a binary (it makes an Intel hex by default) and I don't have much information beyond that to help me. After fighting with the SDCC toolchain a while I decided to "scrape" the assembly file for the dummy program implementing the `flash_write_block()` function to retrieve the machine instructions needed.

This isn't my proudest moment but I need to get things working and I didn't see a reliable path forward from the linker's output. The dummy program just needs:

```
#include "stm8l15x.h"

#define BOOT_SIZE   0x400 /* This must match the UBC setting as well */
#define APP_BASE    (0x8000 + BOOT_SIZE)

void flash_write_block(void)
{
  /* the implementation, shown above */
}

void main(void)
{
  flash_write_block(); /* I want to see it called */
  while (1);
}
```

Having compiled this dummy program, SDCC will leave us with a few interesting files, the most interesting to me being the listing (`.lst`) file.

The assembly listing will show the function starting with:

```
000000                        115 _flash_write_block:
```

That is, an underscore, the name, and a colon (it's a label). The last instruction in the function should of course be a `ret`:

```
00006F 81               [ 4]  193     ret
```

So we just need to grab the instructions (column 2 above, for instance `0x81` is the STM8 `ret` instruction) and write them into an array. A more complex line may look like:

```
000060 A3 00 80         [ 2]  184     cpw x, #0x0080
```

And our array should have `0xA3, 0x00, 0x80,` corresponding to this.

There are a few ways to do this but I wound up writing a quick and dirty Python script to do it. It takes a path to a `.lst` file and a function to "extract" and creates an output file (C header file) with the resulting instructions.

This again isn't my proudest moment but, hey, we're getting very hacky here:

```
def load(f, fname, outfile):
    found = False

    for line in f:
        fields = line.split()
        if found:
            if len(fields) >= 2:
                if fields[1] == '81':
                    found = False
                    write_out(fields, outfile, found)
            else:
                if fields[len(fields) - 1] == fname + ':':
                    outfile.write('#pragma once\n\nstatic char ' + \
                                  fname + '[] = { ')
                    found = True

    outfile.write('};')
```

The `load()` method above should capture everything given a function `fname` from an input file `f` and write a C header file to `outfile`. It's not doing much error checking at all and we assume that it's handed same input with a `0x81` instruction to finish things off.

For reference, the output file corresponding to the function I described here gives me, at this time:

```
#pragma once

static char _flash_write_block[] = { 0x52, 0x00, 0x4d, 0x03, 0x7C, 0x89, 0xEE, 0x
```

...and that does the trick.

We were unable to load Disqus. If you are a moderator please see our troubleshooting guide.