

51QE Bootloader

Neil Klingensmith *

August 7, 2012

1 Purpose Statement

This bootloader was written in order to allow the use of the GNU toolchain for development on the MCF51QE family of microcontrollers. Since the free¹ version of CodeWarrior imposes a code size limitation that makes it impractical for developing large applications. Furthermore, the GNU tools offer a build system that make them independent of cumbersome IDEs. While it is possible to use the CodeWarrior tools without the IDE, it is not commonly done.

Furthermore, the P&E development tools require proprietary software to load code onto their target devices. This limits the usefulness of third-party development tools such as GNU's `gcc`.

2 User Interface

The bootloader's interface is displayed to the user using the 51QE's UART1 operating at 115200 BAUD, 8 data bits, no parity, one stop bit, and no flow control. The user interface is shown in Figure 1. To display the user interface, use a terminal emulator such as `minicom` on a GNU/Linux operating system.

*Send correspondence to `naklingensmi@wisc.edu`.

¹As in beer. No version of CodeWarrior is free as in speech at the time of this writing.

51QE Bootloader

- 1) Load code into RAM
- 2) Load code into flash
- 3) Execute code from RAM
- 4) Execute code from flash
- 5) Erase flash

Figure 1: **Bootloader Main Menu.** The bootloader presents developers with five options for programming and executing code. The serial port settings are 115200/8/N/1.

The bootloader is distributed with an example project to be compiled with the GNU toolchain. The example project includes linker scripts for flash and RAM targets as well as startup code and an application skeleton.

All programs loaded onto the QE device must be transmitted in S-Record format. The example project's makefile generates an S-Record output using the `m68k-elf-objcopy` command. The S-Record file format is well-documented on Freescale's website for those who are interested.

2.1 Running Code from RAM

Running code from RAM is convenient because programming is fast and does not wear out the memory cells as it does with flash. Also, the startup code is simpler because data does not

need to be copied from nonvolatile memory into volatile memory. However, the contents of RAM are not persistent through device resets or power cycles.

A linker script named `linkerscript-ram.ld` for loading code into RAM is included in the example project. This file places the interrupt vector table at address `0x800000`, and it places the code (labeled `.text`) directly following the vector table. The program's data is placed above the `.text` section (labeled `.rodata`, `.data`, and `.bss`). For a more detailed explanation of linker scripts, please refer to section 3.2.

Select option (2) from the bootloader's menu to load a program into flash. Once option (2) is selected, the bootloader is ready to receive an S-Record at full speed. No line or character delays need to be inserted when loading to RAM. The S-Record loaded to RAM must be compiled as a RAM target. Any S-Record lines that do not fall within the RAM memory space (`0x800000` - `0x802000`) will be ignored. In order to generate a RAM target, type `make ram` at the command prompt from within the example application directory.

The startup code is common for both flash and RAM, and is located in `start.sx`². For the RAM target, the `load_vectors` and `load_data` sub-routines are not assembled because the data sections do not need to be copied into RAM from flash.

In the RAM vector table, the initial program counter must be located at address `0x800004`. This is the address that the bootloader reads when option (3) is selected from the bootloader's menu (see Figure 1). In the example project,

²Files ending with the `.sx` extension are passed through the preprocessor before being assembled, while files ending with the `.s` extension are sent straight to the assembler.

address `0x800004` points to the `start` function, which initializes the stack pointer and the status register before calling `main`. It is the application's responsibility to relocate the vector table to the beginning of RAM by writing `0x800000` to the VBR. This is also done by `start` in the example project.

2.2 Running Code from Flash

Running code from flash is somewhat less convenient than running code from RAM because the flash reprogramming procedures are more complicated. However, the 51QE devices include much more nonvolatile memory than RAM, so larger programs can be loaded and run from flash.

Before programming flash, it must be erased. This can be accomplished by selecting option (5) from the bootloader menu (Figure 1). The bootloader only erases flash in the application section (above address `0x2000`). Memory below `0x2000` contains the bootloader's code, and is not erased.

After erasing flash, select option (2) to reprogram the device with an S-Record. Since the flash block programming algorithm is slow, short delays must be inserted after each line³. The shell script `xmit_srec.sh` inserts the appropriate line delays while programming flash. In the example application's makefile, the make program target calls `xmit_srec.sh` with the appropriate parameters to erase and program flash over the `/dev/ttyUSB0` UART.

When writing code to be executed by the bootloader from flash, the application's vector table must be located at address `0x2000`. Since address `0x2000` is not a permissible value for the VBR, the application's startup code must copy

³20 ms line delays are sufficient.

the vector table to 0x800000 before enabling interrupts.

3 Other Stuff

This section discusses details of the linker script and startup code used in the 51QE bootloader. These issues are normally taken for granted by application programmers. Since the bootloader requires modifications to `startcf.c` and `Project.lcf`, a short explanation of the modifications is in order.

3.1 Example Application

3.2 Sections in an Executable File

The linker script explanations discussed in the document frequently refer to “sections” of code and data such as `.text`, `.bss`, and so fourth. These sections are essentially queues that the executable file gives to the loader about where different parts of the program should be placed. When programming a PC, the absolute addresses of the sections are generally unimportant because they are reassigned at run time by the operating system and the hardware. In other words, if a program runs on a PC five times, it may be placed in five different locations in physical memory.

However, bare metal applications like the ones written for microcontrollers must specify the exact locations of their code and their data. This is because there is no operating system present to load the code into memory, and no virtual addressing hardware to translate relative addresses into physical addresses at runtime. In an embedded compiler, the burden of code location falls on the linker and the loader, and the programmer specifies the addresses using a linker script.

To understand the need for a loader, we can examine the typical short assembly file shown in Figure 2. This assembly file contains two types of information: code and data. The data is declared first using a `ds.1` directive. A few lines of assembly code follow.

Most computers treat code and data in fundamentally different ways. For example, many systems assume that memory regions containing code cannot be written to, or that writing to them requires some special procedures. The data sections are often further subdivided into read-only data (declared as `const` in C), initialized, and uninitialized data. For this reason, they are treated separately by the linker and the loader, and they are stored separately in the executable file.

In the case of microcontrollers, code and data are stored in different types of memory which are mapped to different addresses in the address space. The linker and loader must be made aware of the storage capabilities of the different memory types and their location in the address space in order to properly locate code. The utility of a linker script is to outline the memory map of a target system for the linker and loader, and to tell it where each section (code and data) should be placed.

3.3 Memory Map

Traditionally, the data segment is placed at the bottom of RAM, and the stack starts at the top, growing downward toward the data. Since the bootloader may be required to load application code or data at the bottom of RAM, data stored there may be overwritten. Therefore, the bootloader locates all its global variables above the stack, at the top of RAM. A memory map for the bootloader is shown in Figure 3.

```

array:
ds.l 4

function:
link a6,#-4

lea array, a0
clr.l (a0)+
clr.l (a0)+
clr.l (a0)+
clr.l (a0)

unlk a6
rts

```

Figure 2: **A Simple Assembly File.**

3.4 Start Code

To conserve code space, the bootloader does not initialize its global variables when it boots up. Instead, values must be initialized by application code before they are used.

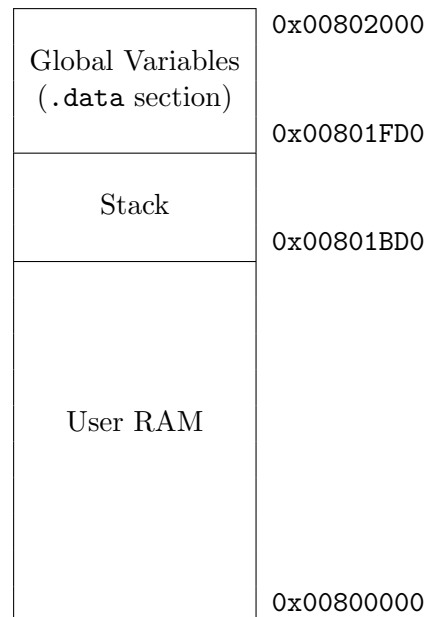


Figure 3: **Bootloader RAM Memory Map.** The bootloader uses the top end of RAM for global variable storage in order to keep the lowest region of RAM free. The top of RAM is unlikely to be overwritten by program data because it is the region that programs generally use for the stack.