The GRUB Switch
USB Device Firmware

Ruediger Willenberg

Version 1.4
June 27, 2022

This guide expects that you know how to use the Linux command-line in a console window and are familiar with commands to change directories, list files and start bash scripts. You might also need *super user* access for some steps, so make sure you have **`sudo`** rights.

Table of Contents

1 About the USB Storage Device firmware

1.1 Licensing

Our USB firmware was modified from code that *Atmel Corporation* open-sourced for the Application note *AVR273: USB Mass Storage Implementation on megaAVR with USB*.

The majority of original code was left intact or only has minor modifications and is under an Atmel-specific “Use with attribution”-style license, whose only major restriction is that the code can only be used in connection with Atmel devices. It MAY NOT be ported to other microcontroller architectures from competing vendors.
NOTE: Atmel has been part of Microchip Technology Inc. since 2016.

The source files added by the GRUB Switch project are under an MIT license:

```
2_usb_device/boards/*

2_usb_device/lib_mem/cf/bootfile_template.S

2_usb_device/lib_mem/cf/boot_choice.h
2_usb_device/lib_mem/cf/boot_choice.c

2_usb_device/lib_mem/cf/fat12_data.h
2_usb_device/lib_mem/cf/fat12_data.c
```

1.2 Functionality

A board properly programmed with the firmware appears as a flash drive on plugging it in. Specifically, it identifies as a drive with one FAT12 partition (the format used on old DOS floppy disks) with the minimum possibly capacity of 33792 bytes (66 sectors of 512 byte). This minimizes management effort and is perfectly sufficient for our purpose.

Files can not be randomly written to the drive, not even below that stated capacity. In fact, the drive will always show exactly 3 files:

`.entries.txt` is the only file that can actually be written, which is done to write a new boot configuration (a list and order of boot choices or entries and some display parameters) onto the board. The data is stored in the chip’s EEPROM, which has 1KByte or 2 sectors of capacity. However, entry files can actually not be larger than 960 bytes (which is sufficient for all practical cases), as we reserve 64 bytes for other stored parameters.

With an external pin, the board can be put into *Write Protect* mode, and then even **`.entries.txt`** is *read-only* to protect the boot entries from tampering.

`.bootpins.txt` is a read-only file that is generated on opening/reading and displays the state of the board's boot choice pins, depending on the configured choice mode:

- In *1-of-n mode*, which is the default, the file indicates with a 1 which pins have been tied to low-level/GND, the lowest order-one making the boot choice. The following example shows a situation where choice pin 3 has been connected:
  `11->1: 00000000100`
- In *binary mode*, which can be activated through another board pin, only the first 4 choice pins are sampled and connections are interpreted as the powers-of-two 1,2,4,8. The following example shows a situation where the lower three pins have been connected to low-level/GND, resulting in a boot choice of entry 7:
  `Binary pick:  0111`

`SWITCH.GRB` is the only file that is not declared "hidden" by either FAT attributes (for Windows) or a leading period in the name (for Linux). It is another *read-only*, dynamically generated file and it's content depends on the boot choice pins, the choice mode and the content of **`.entries.txt`**. It can be sourced on boot by the GRUB infrastructure as a script and will set a new GRUB menu default entry and a menu wait time of 0 seconds, thereby forcing the GRUB menu choice. If boot choice pins indicate no connection (choice 0), the file is empty and the GRUB menu is not cut off.

As to make sure that the PC is not buffering or caching an old version of **`SWITCH.GRB`**, the board will disconnect from the USB host and reconnect right away whenever a board pin changes state (including on change of the *Write Protect* status), forcing the PC to read the updated **`SWITCH.GRB`**

Because the file structure does not change except for the size of **`.entries.txt`**, almost the entire *File Allocation Table* (FAT) is immutable and actually stored in code flash memory.

1.3 Code structure overview

General startup and operation code, multi-tasking between USB und storage management and other required tasks are found in the directories

```
./main/
./conf/
./modules/
./lib_mcu/
```

Both the overall flash storage handling and our customized GRUB Switch code are placed in

./lib_mem/cf/

A few words about specific files:

bootfile_template.S includes the text file **grub-switch/bootfiles/template** into code flash memory, and corresponding address labels so it can be read from C code. It holds the large static portion of the **SWITCH.GRB** script. Only the setting of a few environment variables is dynamically prepended to it when serving the file for reading, based on boot choice and the configuration in **.entries.txt**.

fat12_data.h and **fat12_data.c** provide most of the File Allocation Table and static and dynamic content of the three files. As mentioned before, the FAT and therefore the names, attributes and sector locations of the three files are largely static data. Only the size of the **SWITCH.GRB** and **.entries.txt** is modified dynamically, as is the *read-only* attribute of **.entries.txt** based on Write-Protect status.

boot_choice.h and **boot_choice.c** implement initialization of all I/O pins, reading of boot choice and automatic USB detach/reattach following a state change of an input pin.

cf.h and **cf.c** hold the modified functions for sector-wise write and read. The **cf_write_sector()** function implements writing of up to 960 bytes of **.entries.txt** (if Write-Protect is disabled) and adjusts file size in the FAT. The **cf_read_sector()** function relays read accesses to the responsible file-specific functions in **fat12_data.c**.

The directory

./boards/include/

holds specific header files for each ATmega**u4 boards currently supported (see the section on adding new boards for details). The file **boards.h** includes all board files and provides the preprocessor macros that facilitate easy customization for specific boards.

2 Pre-built firmware images

The directory **`./prebuilt_images/`** holds pre-built *Intel Hex Format* images, based on the currently released sources, for all *ATmega**u4*-based boards that are supported through a corresponding header file. Currently these are:

Arduino Micro

Adafruit ItsyBitsy 32u4 (3 and 5V versions)

Sparkfun Pro Micro and clones (3 and 5V versions)

DFRobot Beetle (DFR0282 with Atmega 32u4)

Teensy 2.0

Our Custom PCB design hardware (see documentation in **`3_custom_hardware.pdf`**)

Furthermore, the directory holds two bash scripts to facilitate programming (see next section).

Individual programming information and wiring diagrams to connect various switch options are listed both in **`quickstart.pdf`** as well as in individual text files in **`./boards/`**

3 Programming a firmware image onto a board

The process to flash a firmware image into the *ATmega**u4* MCU depends on what bootlader has been pre-installed into its protected bootloader flash section. Chips by default start the programmed firmware on power-up, in our case the GRUB Switch firmware. To start the bootloader requires specific actions. For easier programming, we have provided two bash scripts in **`./prebuilt_images/`** (see the end of each subsection).

3.1 DFU bootloader (factory default, non-Arduino)

The factory-delivered chip has a so called *DFU bootloader* installed, which conforms to a USB standard for devices that can be reprogrammed. The **/RST** and **/HWB** pins have to be cycled in a specific timing and order to enter the bootloader. Bootloader mode is then only left by command, reset or power-off. Among our currently supported boards, only our own custom hardware design uses the DFU bootloader.
To program such chips, the free tool **dfu-programmer** needs to be installed. For example in Ubuntu Linux, a version 0.6.1 package is available for install with

```
sudo apt-get install dfu-programmer
```

Alternatively you can download, make and install the most recent 0.7.2 version from

https://sourceforge.net/projects/dfu-programmer/files/latest/download

Both versions work and are automatically supported by an automated script we supply (see below). If not using the script, you need slightly different commands depending on the version installed. You can check the version with

```
dfu-programmer --version
```

To enter the bootloader when another firmware has already been programmed before, you can press the **PROG** button on our board. In DFU bootloader mode, a chip device will connect and identify either as

"*Atmel Corp. atmega32u4 DFU bootloader*" with USB VID 0x03eb, PID 0x2ff4.
or
"*Atmel Corp. atmega16u4 DFU bootloader*" with USB VID 0x03eb, PID 0x2ff3.

depending on which of the two possible chips has been used on the hardware.

Calling the **lsusb** tool will show both this information as well as (decimal!) bus and device numbers, for example:

```
lsusb
```

(...other USB devices…)
```
Bus 002 Device 017: ID 03eb:2ff4 Atmel Corp. atmega32u4 DFU bootloader
```
(...other USB devices...)

Caution: The device number changes everytime the chip reconnects.

In the following examples, please substitute ****** with **32** or **16** depending on the chip and use the commands for the installed version of *dfu-programmer*. The bus and device numbers must be used - without leading zeroes – when programming. First, the current flash programming should be erased (remember 2,17 are example numbers):

```
dfu-programmer  atmega**u4:2,17  erase            (v.0.6.x)
dfu-programmer  atmega**u4:2,17  erase --force    (v.0.7.x)
```

(sudo might be required on your installation)

The actual programming is initiated with

```
dfu-programmer  atmega**u4:2,17  flash FIRMWARE.hex     (all versions)
```

The device can be induced to restart and reconnect with the flash programming:

```
dfu-programmer  atmega**u4:2,17  reset      (v.0.6.x)
dfu-programmer  atmega**u4:2,17  launch     (v.0.7.x)
```

If you have programmed a device with the GRUB Switch firmware, it should identify with VID 1209, PID 2015 and either “The GRUB SWITCH” or "InterBiometrics", which is the company that donated the 1209 USB VID range for community use:

```
lsusb
Bus 002 Device 018: ID 1209:2015 The GRUB SWITCH
```

For more details, read the **dfu-programmer** manual with

```
man dfu-programmer
```

3.1.1 Easy DFU programming

To simplify the process, we have built a bash script that automates the bus scanning and the three calls to the programmer. After entering **`./prebuilt_images/`**, call

```
sudo ./program_grubswitch_dfu.sh  GS_CUSTOMHW_REVC_32U4.hex
```

or

```
sudo ./program_grubswitch_dfu.sh  GS_CUSTOMHW_REVC_16U4.hex
```

(Note: There is also still build support and prebuilt `*_REVB_*.hex` images for Custom Rev. B in the repository as there are still a few of those older boards out there).

The script asks you to put the board into bootloader mode. For the Custom HW board, this is accomplished by pressing the board's PROG button. The chip resets and reconnects as a DFU class USB device. Press any key to commence flashing the firmware image onto the chip.

After successful programming, the board should reconnect as a flash drive with the name **BOOTTHIS**. Your Linux GUI might auto-mount the drive or offer to do so.

3.2 Arduino-compatible bootloaders

Most commercially available boards support being programmed through the Arduino IDE. Arduino application firmwares include a mechanism by which no button needs to be pushed on the boards to initiate programming, but that requires that the application firmware includes special supporting code. We are not indulging this requirement in our GRUB Switch firmware.

However, each of these boards can enter bootloader mode by connecting the **/RST** button to **0V/GND** twice in short order. *Arduino Micro* and *ItsyBitsy* have an actual button for this, *Pro Micro* and *DFR Beetle* need the corresponding pads to be shorted with a wire or soldered connection.

In each of these boards, the bootloader quits and enters the application firmware again after 5-10 seconds, so bootloader entry needs to be initiated right before programming (see below).

To program these boards, the free tool **avrdude** needs to be installed, for example under Ubuntu with

```
sudo apt-get install avrdude
```

(**avrdude** is also used by Arduino IDE).

When putting a board into bootloader mode, it will identify as a USB UART, usually as `/dev/ttyACM0` . You can check through the file date if a special **tty** has just turned up:

```
ls -l /dev/tty* | grep "/dev/tty[^S0-9]"
```

Remember that the bootloader will yield to the regular board program after a few seconds and the **tty** device will be gone again.

Before the bootloader timeout, avrdude can initiate programming with

```
avrdude -v -F -p atmega32u4 -c avr109 -P /dev/ttyACM0 -Uflash:w:FIRMWARE.hex:i
```

For more details, read the avrdude manual with

```
man avrdude
```

3.2.1 Easy *avrdude* programming

To simplify the process, we have built a bash script that automates the UART identification scanning and the calls to the programmer. After entering **`./prebuilt_images/`**, call

```
sudo ./program_grubswitch_avrdude.sh FIRMWARE.hex
```

(sudo might not be required on every machine, but doesn't hurt)

The script asks you to put the board into bootloader mode by pushing the button respectively grounding **/RST** twice in short order The chip enters bootloader mode and remains in it for about 5-10 secs, then returns to normal program operation.

Make sure to press any key inside this time margin to commence flashing the firmware image onto the chip.

After successful programming, the board should reconnect as a flash drive with the name **BOOTTHIS**. Your Linux GUI might auto-mount the drive or offer to do so.

3.3 Teensy 2.0

Teensy 2.0 uses its own customized bootloader and programming software for a fairly simply programming process. You can find all installation and programming instructions here:

https://www.pjrc.com/teensy/loader_linux.html

https://www.pjrc.com/teensy/loader_cli.html

After successful programming, the board should reconnect as a flash drive with the name **BOOTTHIS**. Your Linux GUI might auto-mount the drive or offer to do so.

3.4 Steps after programming the firmware

A successfully flashed board can be mounted and will appear with the drive name **BOOTTHIS**. If you display hidden files, you should see all three FAT12 files described before.

If you have previously used the configuration tool
`grub-switch/1_shell_scripts/CONFIGURE_GRUBswitch.sh`

to generate a GRUB Switch configuration (**`grub-switch/bootfiles/.entries.txt`**), you can copy this file onto the drive yourself, or you can use option "**4**" in the **CONFIGURE_GRUBswitch** menu to auto-mount, write the file and unmount the drive again.

Individual wiring diagrams to connect various switch options are listed both in **`quickstart.pdf`** as well as in individual text files in **`./boards/`**

4 How to build the firmware yourself

4.1 Prerequisites

The device firmware is intended to be built with the free GNU toolchain under Linux (though it can be made to work on other OSes). At the minimum, you need to install the packages for the GCC AVR compiler and the AVR standard libraries, for example under Ubuntu with

```
sudo apt-get install gcc-avr
```

```
sudo apt-get install avr-libc
```

Further tools are needed to program various boards (see below)

If you have not already downloaded the GRUB Switch repository itself, there are two ways to obtain it:

- Via **git clone**:
  - Go to your preferred working directory
  - Clone the repository with

    ```
    git clone  https://github.com/rw-hsma-fpga/grub-switch.git
    ```
  - All GRUB Switch files are now available in the path

    ```
    ./grub-switch
    ```
- Via **download**:
  - Download to your preferred directory: https://github.com/rw-hsma-fpga/grub-switch/archive/refs/heads/master.zip
  - Unzip with GUI-tool or on the command line with

    ```
    unzip grub-switch-master.zip
    ```
  - All GRUB Switch files are now available in the path

    ```
    ./grub-switch-master
    ```

All further examples will assume you are starting in the parent directory above `grub-switch`

4.2 Building firmware images

The root directory for the device firmware can be entered with

```
cd grub-switch/2_usb device/
```

For a first run at building the firmware, just call *make*:

```
make
```

A successful run should indicate that an image for the TEENSY20 (Teensy 2.0 board) has been built, as that is the default set in the *Makefile*. Three files are being generated in the local directory: An *ELF* file, a file in *Intel Hex format* and a memory map report. Generally, `*.hex` files will be used for device programming:

```
GS_TEENSY20.elf   GS_TEENSY20.hex   GS_TEENSY20.map
```

All intermediate build files are found in the `./build/` path which can be cleaned with

```
make clean
```

To build for another target board, specify the *BOARD* parameter:

```
make BOARD=TARGETBOARD
```

The target board name is always a capitalized version of the corresponding board header file that can be found in `./boards/include/` . As an example, the header file for the *Arduino Micro* is `./boards/include/arduino_micro.h` and the image for it is built with

```
make BOARD=ARDUINO_MICRO
```

which generates

```
GS_ARDUINO_MICRO.elf   GS_ARDUINO_MICRO.hex   GS_ARDUINO_MICRO.map
```

If you use a board with an ATmega16U4 (rare!), you have to specify that MCU in small letters:

```
make  BOARD=SOME16U4BOARD  MCU=atmega16u4
```

There is a bash script to auto-generate all images with corresponding headers in `./boards/include/`, which can be called with

```
./make_all_boards
```

If you want to store the generated images as prebuilt images, you have to hand-copy them to the `./prebuilt_images/` directory with

```
cp *.hex ./prebuilt_images
```

Do not delete the `./prebuilt_images/` directory even when you want to re-generate all images, as it also holds bash scripts for programming the images onto the boards.

5 Adding a new board as a build target

The device firmware sources are designed to make it easy to add support for another new board with the *ATmega32u4/16u4* chips. With proper information in the form of a schematic, you only need to copy, rename and modify one file and add a line to another. For both actions you should go to the `2_usb_device/boards/include/` path, for example with:

```
cd grub-switch/2_usb_device/boards/include
```

5.1 Making a board header file and changing named tokens

The best way to produce a valid header file for your board is to copy an existing file and modify it properly. Let's say you board is called **Foobar+** and we're using the *Arduino Micro* header file as a template, then you start with

```
cp arduino_micro.h foobar_plus.h
```

Open the file with your favored text editor, for example

```
nano foobar_plus.h
```

You should then change all capitalized mentions of ARDUINO_MICRO in tokens to capitalized FOOBAR_PLUS, so the following lines at various positions in the file

```
#ifndef ARDUINO_MICRO_H
#define ARDUINO_MICRO_H
```

```
#ifdef TARGET_BOARD_ARDUINO_MICRO
```

```
#endif   // TARGET_BOARD==ARDUINO_MICRO
#endif   // ARDUINO_MICRO_H
```

should turn into

```
#ifndef FOOBAR_PLUS_H
#define FOOBAR_PLUS_H
```

```
#ifdef TARGET_BOARD_FOOBAR_PLUS
```

```
#endif   // TARGET_BOARD==FOOBAR_PLUS
#endif   // FOOBAR_PLUS_H
```

NOTE: If your board does not use an ATmega32u4, but an ATmega16u4 (rare), you should add a `_16u4` suffix to your board file name and `_16U4` to preprocessor tokens:

```
cp arduino_micro.h foobar_plus_16u4.h
#define FOOBAR_PLUS_16U4_H
```

5.2 Adjusting the clock frequency (relates to supply voltage)

*ATmega**u4* MCUs serving as a USB device need one of two external crystals or oscillators:

- If the chip is supplied with a **VCC** of **+5V**, it can be clocked with **8 or 16MHz**
- If the chip is supplied with a **VCC** of **+3.3V**, it needs to be clocked with **8MHz**

There are no other combinations of supply voltage and external frequency that work. The internal oscillator in the **-RC** versions of the chip is not accurate enough for USB2.0 timing.

The chip sadly does not autodetect its environment and needs to be told which divider to use for its USB timing (therefore there are some boards like the *Pro Micro* that have almost identical 3V and 5V versions, but different crystal frequencies), but they need this one different value in their firmware image.

You need to set the **FOSC** macro in the header file to the proper clock value in **kHz**:

```
#define FOSC 8000    // for 8MHz clocking
```
or
```
#define FOSC 16000   // for 16MHz clocking
```

5.3 Pin assignments

*ATmega**u4* name all their port pins with digital GPIO capability with letter/digit combinations starting from **PB0** up to **PF7**. They are almost all multiplexed with other pin functions, but a specific board will usually only expose those GPIO pins as I/O solder pads that are not required to operate as something else.

Every board will have a different set and order of exposed pins and it is up the header file writer to put the required *GRUB Switch* functions into a practical order (if in doubt, look at the pinout assignments and switching diagrams for the already supported boards). The pads might also use other naming schemes, like the Arduino one, so you DO need a schematic to be sure which pad is connected to which MCU pin.

For each pin function we want to declare, we have defined a pair of macros ***FUNCTION*_PORT** and ***FUNCTION*_PIN** that require you to separately specify the letter (port) and the digit (pin) of the assigned pin.

All pins defined for input functions (pin level sampled by the firmware for some purpose) have a pull-up resistor activated so that they are by default a logical high-level (1). This way both input levels can be chosen by connecting it to **0V/GND** (0) or leaving it unconnected (1).

5.3.1 Setting the LED pin and active level

Most boards have at least one LED that can be switched by a GPIO pin. Our firmware uses that LED to indicate the writing of a new boot configuration (which can take up to 2 seconds) and it also flashes briefly on USB power-up.

On the *Arduino Micro*, the LED is connected to Pin **PC7**, so the definitions look like this:

```
#define LED_PORT  C
#define LED_PIN   7
```

(The P for port is left out) You need to adjust this to the letter/number combination for your board's LED port.

Boards can also differ in whether the LED lights up on a high level (1) or a low level (0) output. On most boards, the former is true: The LED's *anode* is connected to the pin, and the *cathode* is tied to **0V** (there'll be a series resistor somewhere). But on some boards, the LED's *cathode* is connected to the pin and the *anode* is tied to **VCC**. If that is the case, you need to add and extra definition

```
#define LED_IS_LOW_ACTIVE
```

and the firmware will make the proper adjustment.

5.3.2 /WR_PROT and /BINARY_MODE pins

Two other special pin functions are required for proper firmware operation.

If the **/WR_PROT** pin is tied to **low-level/0V/GND**, the firmware doesn't allow to overwrite the current boot configuration file. If it's left unconnected, changing the boot configuration is possible. Our *Arduino Micro* configuration uses **PF6**, so the macros are

```
#define WRPROT_PORT  F
#define WRPROT_PIN   6
```

Adjust accordingly to your preferred Write-Protect pin.

When the **/BINARY_MODE** pin is tied to **low-level/0V/GND**, the firmware interprets the first four *choice pins* (see below) as a binary number, therefore enabling boot choices from 0..15 (0x0..0xF). If the pin is unconnected, the firmware uses the default *1-of-n* mode. Our *Arduino Micro* configuration uses **PF1**, so the macros are

```
#define MODE_PORT  F
#define MODE_PIN   1
```

Adjust accordingly to your preferred Binary mode pin.

5.3.3 Auxiliary low-level(0V) pins.

It is often helpful to have the **/WR_PROT** and **/BINARY_MODE** pads neighbouring a **0V/GND** pad, so a jumper or a short solder bridge can be used to activate either function. However, GND pads are limited in number and this is not always feasible. We have therefore defined two macro pairs that make regular GPIO pins outputs that are permanently set to **low-level/0V**. They can then be used to tie down the mode pins, or even connect the *choice pins* (see below) to them instead of to a real **0V/GND** pad. Our *Arduino Micro* configuration uses **PF7** and **PF0**, which are indeed adjacent to **/WR_PROT** and **/BINARY_MODE** on that board:

```
#define AUX0_PORT1  F
#define AUX0_PIN1   7

#define AUX0_PORT2  F
#define AUX0_PIN2   0
```

Adjust accordingly to your pins. If you don't need two such pins and are running out of otherwise unused MCU pins, you can even both declare them to the same location without any harm (the output direction and 0-value is just set twice).

5.3.4 Choice pins

Up to 11 pins can be declared to indicate boot choices by tying them to **0V/GND**. For the default *1-of-n*-mode (**/BINARY_MODE** not active), the lowest-order pin tied to 0 indicated the boot choice; if no choice pin is connected, GRUB Switch goes to the GRUB menu. In *Binary mode* (**/BINARY_MODE** tied to 0), only the choice pins 1..4 are read and interpreted as powers of two 1,2,4,8 when they're tied to GND.

Our *Arduino Micro* configuration uses 11 pads from **PD1** to **PD6** for these choice pins:

```
#define  CHOICE_PORT1       D
#define  CHOICE_PIN1        1

#define  CHOICE_PORT2       D
#define  CHOICE_PIN2        0
[…]
#define  CHOICE_PORT10      B
#define  CHOICE_PIN10       7

#define  CHOICE_PORT11      D
#define  CHOICE_PIN11       6
```

Adjust accordingly to your board's pad order. If not enough pins or pads are free for 11 choices, all the upper choices can be assigned to the same pin with no harm (it should of course be unconnected, and all these choices will then never be selected).

We recommend again to look at the connection patterns and assignments for the boards that are already supported for inspiration. They can be found in the **`quickstart.pdf`** as well as in the form of ASCII text files in the **`./boards/`** directory.

When all the modifications have been made and checked again, you can save the new header file.

5.4 Add header file to `boards.h` and try building an image

Open the existing file **`2_usb_device/boards/include/boards.h`** with your favourite text editor, e.g.

```
nano boards.h
```

and add your new header file to the list of includes, for example

```
#include "boards/include/foobar_plus.h"
```

Save and close the file. The adding process is finished. After returning to the **`2_usb_device`** directory with

```
cd ../../
```

it should now be possible to compile the new firmware by specifying the new board code in capitalized form, for example

```
make BOARD=FOOBAR_PLUS
```

which will produce the requisite ELF, HEX and MAP files. Because our bash script parses the **`boards.h`** file, calling

```
./make_all_boards
```

will also include a build for the new board's firmware image.