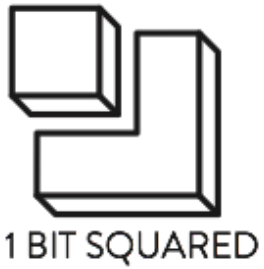
Embedded Programming with Black Magic and the Lights On

Piotr Esden-Tempski (@esden)



Introduction

Embedded systems programming has earned a bad reputation of being difficult to master. Especially in the open-source world, most people associate it with cut and pasted code that is difficult to debug. The usual tools we have to debug embedded systems are a blinking LED and if we are lucky **`printf`** statements through a serial port.

Modern CPUs like ARM come with an interface called JTAG or its two wire successor SWD or C-JTAG. These are becoming accessible in more platforms but either come with proprietary tools or require hours of research to get working.

In this workshop we will be using the Black Magic Probe JTAG/SWD debugging and programming tool and the 1Bitsy ARM Cortex-M4F development board. It is an open-source and open-hardware solution that makes programming and debugging of embedded systems plug & play and hassle-free.

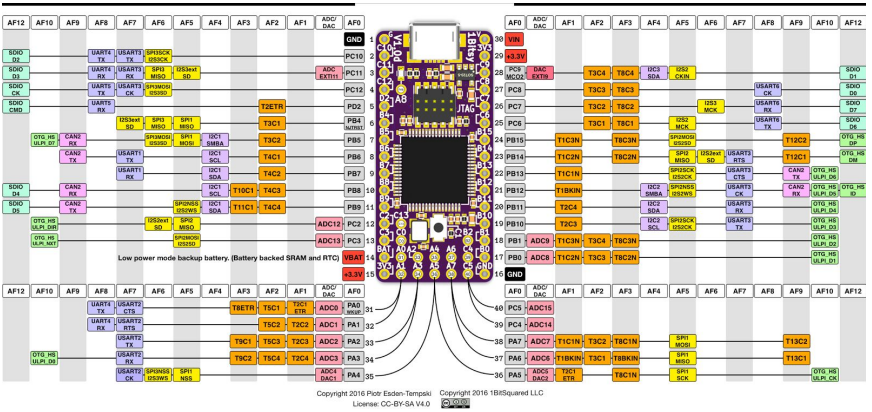
Note: Please observe the difference between the following characters and pay attention to them when copying code:

- **`l`**: small letter L
- **`1`**: the number one
- **`I`**: large letter i
- **`O`**: large letter O
- **`0`**: the number zero

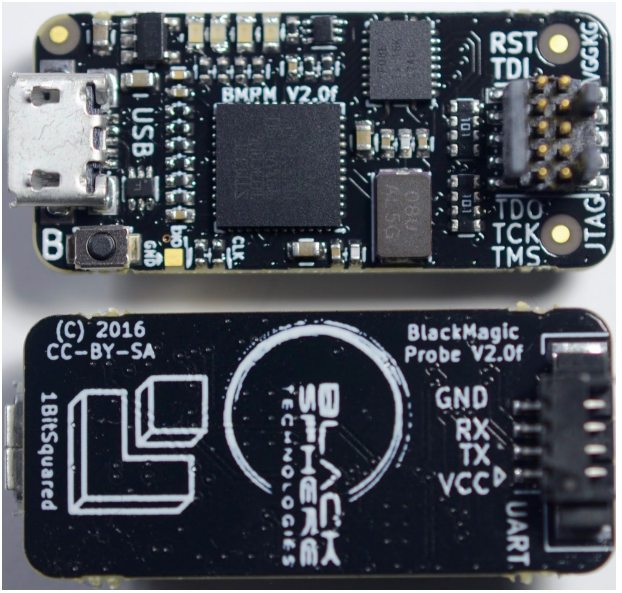
For more information about 1Bitsy you can visit http://1bitsy.org. If you want to learn more about Black Magic Probe visit http://github.com/blacksphere/blackmagic.

Hardware

1Bitsy: Our target hardware is a 1Bitsy. It is a small and powerful ARM microcontroller board based on the STM32F415 ARM Cortex-M4F CPU.



Black Magic Probe (BMP): To connect our target board to the computer we will be using the BMP. It is a small board that translates the JTAG and SWD protocols to the GNU Debugger remote target protocol. This removes the need for middleman software like OpenOCD.



Software

GNU Debugger (GDB): The standard debugger for the GNU toolchain but has been ported to many architectures. There are plenty of forks and GUI versions, but we will use the standard command-line version.
Instead of connecting GDB to a target process on your PC, we will connect GDB to the Black Magic Probe to debug via JTAG.
Since we're debugging an ARM target on an x86 system, we'll use **`arm-none-eabi-gdb`**
Note: If you are looking for the gdb version that has Python scripting compiled in you can also use the **`arm-none-eabi-gdb-py`**.

GCC: GNU C Compiler is a program that translates a C program to binary. We will be using a cross compiler. It runs on your computer and compiles a binary for a different CPU (ARM) than the one you are running the compiler on (x86). Thus it is called a cross platform compiler. In our case we will be using the gcc-arm-embedded toolchain and the binary is called **`arm-none-eabi-gcc`.**

Text Editor: You can use pretty much any text editor you want. It is always nice to have syntax highlighting for your code but you can do programming with anything that can write an ASCII text file.

GNU Make: This is optional, but to make compilation easier you can use this tool. It will detect if files changed and recompile them only if necessary.

Setting up

1. Create a directory for your experiments.
   > mkdir ~/class-1
   > cd ~/class-1
2. Clone the libopencm3 low level hardware library that we will be using:
   > git clone https://github.com/libopencm3/libopencm3.git
3. Compile the stm32f4 libopencm3 library:
   > make -C libopencm3 lib TARGETS=stm32/f4
4. Open a text editor of your choosing and create a C program file named **program-1.c** with the following content:

```
#include <stdio.h>
int main(void)
{
    int a = 0;
    int b = 2;
    int result = a + b;

    /* this allows you to us printf and read
       through the GDB interface */
    initialise_monitor_handles();

    printf("The result is: %d\n", result);

    /* Never return we have nowhere to go*/
    while (1) {
        asm("nop");
    }
    return 0;
}
```

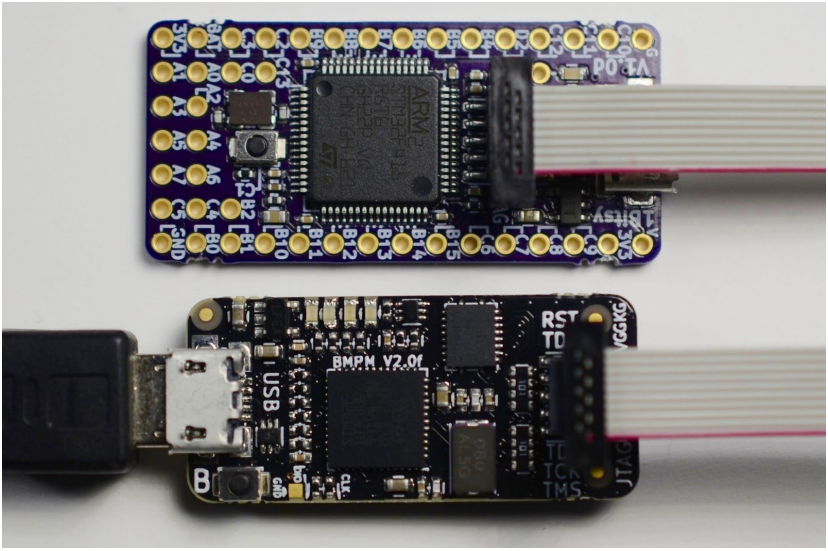
5. As opposed to a PC target that has a well defined memory layout, we also have to tell the linker how much memory we have available and where it is located. To provide that information to the linker we have to create a linker script file named **1bitsy.ld**:

```
MEMORY
{
rom (rx) : ORIGIN = 0x08000000, LENGTH = 1024K
ram (rwx) : ORIGIN = 0x20000000, LENGTH = 128K
}
INCLUDE libopencm3_stm32f4.ld
```

6. Compile your program:

```
> arm-none-eabi-gcc -g3 -mcpu=cortex-m4
  -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16
  --specs=rdimon.specs -nostartfiles
  -Llibopencm3/lib -Wl,-T,1bitsy.ld
  program-1.c -lopencm3_stm32f4
  -o program-1.elf
```

Now you can connect your Black Magic Probe to your 1Bitsy as shown in the picture and then connect the Black Magic Probe to your computer using the micro USB cable.



Connect and Download

Now start GDB, connect to the target, flash (“download”) the firmware and run our program:

1. Start GDB**:**
   > arm-none-eabi-gdb program-1.elf
2. Connect to the Black Magic Probe:
   (gdb) target extended-remote /dev/ttyACM0
3. Enable power supply to the target device:
   (gdb) monitor tpwr enable
4. Scan for connected CPUs:
   (gdb) monitor jtag_scan
5. Attach to the newly found CPU:
   (gdb) attach 1
6. Download the firmware:
   (gdb) load
7. Run the program:
   (gdb) run

Let’s have some fun

Now that we have a program running on our target hardware we should explore our environment. Here is a list of a few commands that are handy:

1. If you press **Control-C** you will interrupt the program execution and take over control.
2. **run** will start or restart the program.
3. **start** will set a temporary breakpoint at the beginning of the **main** function, then run the program and stop at the beginning of the **main** function. It will also restart the program if it is already running.
4. **continue** will resume the program at the point where it was interrupted.

5. **`exit`** will quit GDB. You can also do that by pressing the **`Control-D`** keys on your keyboard.

Exercises

1. Restart the program using the **`start`** command. Try to see how variables are being assigned and called using the **`next`** and **`print <variable name>`** GDB commands. If you are lost and don't know where inside your program you are, run the **`list`** or **`list <line number>`** command.
2. While running the program try to inspect the environment variables using the **`info locals`**.
3. Try modifying the program output using the **`set variable <variable name>=<number>`** command.
4. Try setting your own breakpoint in the program using the **`break <line number>`**. Now when you restart or resume your program, GDB will show you a prompt as soon as the execution of your program reaches the line you requested.
5. List your breakpoints with **`info breakpoints`**. You can disable or delete them using **`disable breakpoints`** or **`delete breakpoints`**. You can choose which one to disable or delete if you provide the breakpoint number at the end of the commands. Try to set a breakpoint just before the **`printf`** command is executed and see how you can quickly get to the place you want to be.

Program #2

Let's change things up a bit and write a new program. This program is blinking an LED instead of outputting text through the JTAG interface.

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
void delay(int ticks)
{
  int i;
  for (;ticks!=0; ticks--) {
    for (i=0; i<10000; i++) {
      asm("nop");
    }
  }
}
int main(void)
{
  int blink_count = 0;

  rcc_periph_clock_enable(RCC_GPIOA);
  gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,
                  GPIO_PUPD_NONE, GPIO8);
  gpio_set(GPIOA, GPIO8);

  while(1) {
    blink_count += 1;
    if ((blink_count ^ 4) == 0) {
      delay(100);
    }
    delay(10);
    gpio_toggle(GPIOA, GPIO8);
  }
  return 0;
}
```

You can compile, download, and run the program the same way as the previous one. Do that and you will see the LED blinking. You can use the commands you learned in the exercises before to explore the program and see how the LED is switching on and off when you step through the code.

To make things simpler you can also write a small **`Makefile`**:

```
CC=arm-none-eabi-gcc
CFLAGS=-g3 -mcpu=cortex-m4 -mthumb -mfloat-abi=hard \
       -mfpu=fpv4-sp-d16 --specs=rdimon.specs \
       -Ilibopencm3/include -DSTM32F4
LDFLAGS=-nostartfiles -Llibopencm3/lib \
        -Wl,-T,1bitsy.ld -lopencm3_stm32f4

all: program-2.elf

%.elf: %.o
	${CC} $< ${LDFLAGS} -o $@

clean:
	rm -f *.o *.elf
```

Note: Make sure to insert <TAB> characters marked in gray. The backslashes need to be the last character on the line, they allow you to break a single line and continue it on the next. Aka. escape newline.

This way you can easily recompile your program by running the **`make`** command.

To save time when starting GDB, you can create a **`.gdbinit`** file. This way you don't have to run all the same commands every time you start GDB. You can either create this file in your home directory or where your program is located. The following example file has a few additional commands. They are useful in everyday life with GDB:

```
set history file ~/.gdb_history
set history save on
set target-async on
set confirm off
set mem inaccessible-by-default off
target extended-remote /dev/ttyACM0
monitor version
monitor jtag_scan
attach 1
```

Note: Instead of **`jtag_scan`** you can also use **`swdp_scan`**. This tells the Black Magic Probe if it should use the JTAG or SWD protocols to talk to the target. This is important in cases where the target does not support one of the protocols or you don't have all the signals connected to be able to use the full JTAG interface.

Exercises:

1. Step through the code and see how you can either step over functions using the **`next`** command or step into a function using the **`step`** command.
2. See how you can leave a function by using the **`finish`** command.
3. Set a watchpoint on the **`blink_count`** variable using the **`watch blink_count`** command. Then let the program run using the **`continue`** command and see how it acts similar to a breakpoint that interrupts the code execution every time the variable value changes.
4. See how you can make complex expressions using the **`print <expression>`** command. For example **`print blink_count ^ 4`**, and then format your output with **`printf "my formatting %d", blink_count ^ 4`**.
5. Now combine the two previous exercises with the **`commands`** feature. You can watch a variable and print the condition output by writing the following sequence of commands:

   ```
   (gdb) watch blink_count
   (gdb) commands
   > printf "blink extend condition %d\n",
     blink_count ^ 4
   > end
   ```

6. Try using all you learned so far to find out why the LED blink delay extension triggers only once instead of every four cycles. After finding the solution, edit the program source code in a separate window without exiting GDB. If you did the optional step of creating the makefile you can run the **`make`** command from within GDB followed by **`load`** and **`run`** to see if your fix was successful.