

A black and white photograph of a sandy beach with several footprints. A large, clear footprint is in the lower left, and a trail of smaller footprints leads from the upper right towards the center. The text is overlaid on the left side of the image.

Debugging, Tracing
& Programming

*with
the*

Black Magic
Probe

Thiadmer Riemersma
CompuPhase
October 2019

Table of Contents

Introduction.....	3
Hardware and Software.....	3
About this Guide.....	4
License.....	4
The Debugging Pipeline.....	5
GDB Architecture.....	5
Embedded Debugging: Points for Attention.....	6
Requirements for Front-ends.....	8
Setting up the Black Magic Probe.....	9
Microsoft Windows.....	9
Linux.....	10
Connecting the Target.....	12
Checking the Setup.....	13
Running Commands on Start-up.....	14
Target Board Design for Debugging & Programming.....	15
Debugging Code.....	17
Prerequisite Steps.....	18
Loading a File and Downloading it to the Target.....	19
Starting to Run Code.....	21
Listing Source Code.....	21
Stepping and Running.....	22
Breakpoints and watchpoints.....	22
Examining Variables and Memory.....	24
The Call Stack.....	24
Debug Probe Commands.....	25
The BlackMagic Debugger Front-end.....	27
Run-Time Tracing.....	33
Secondary UART.....	33
Semihosting.....	34
SWO Tracing.....	37
Tracing with Command List on Breakpoints.....	42
The Common Trace Format.....	43
Binary Packet Format.....	44
A Synopsis of TSDL.....	45
Generating Trace Support Files.....	51
Integrating Tracing in your Source Code.....	52
Firmware Programming.....	53
Using GBD.....	53
Using the BlackMagic Flash Programmer.....	54
Updating Black Magic Probe Firmware.....	56
Further Information.....	58
Index.....	59

Introduction

The “Black Magic Probe” is a combined hardware & software project. At the hardware level, it implements JTAG and SWD interfaces for ARM Cortex A-series and M-series micro-controllers. At the software level, it provides a “gdbserver” implementation and Flash programmer support for ranges of micro-controllers of various brands. Both the hardware and software components of the Black Magic Probe are open source projects, designed by 1BitSquared in collaboration with Black Sphere Technologies.

The current (official) release of the Black Magic Probe is version 2.1 of the hardware and version 1.6.1 of the firmware. Derivatives of both hardware and firmware exist, with sometimes different capabilities or limitations. This guide focuses on the *native* hardware, and firmware version 1.6 or later.

Hardware and Software

The Black Magic Probe has a 10-pin JTAG connector with the signals of the ARM *Serial Wire Debug* protocol, plus a separate three-wire UART (TTL level, 3.3V). The debug interface gives you access to single-stepping, hardware breakpoints and watchpoints, dumping memory regions and programming Flash memory. This protocol is meant to be driven by a hardware interface, a *debug probe*.

The Black Magic Probe is such a debug probe. The “black magic” that it adds to alternative debug probes is that it embeds a software interface for GDB, the debugger for GNU GCC compiler suite — a widely used compiler for micro-controller projects. It is the closest that a debug probe can come to plug-&-play operation.

Next to the Black Magic Probe, you need GDB, and more specifically, the GDB from the toolchain that you use to build your embedded code. For the ARM Cortex-A and Cortex-M micro-controllers, this typically means the GDB from the *arm-none-eabi* toolchain.

While you do not *need* a debugger front-end, it is beneficial to get one. When you running on Linux, you may get by with GDB’s integrated *Text User Interface*. See [Requirements for Front-ends](#) (page 8) for tips to select a front-end.

About this Guide

This guide is not a book on GDB. That book is *The Art of Debugging with GDB, DDD and Eclipse* by Norman Matloff and Peter J. Salzman, and which is highly recommended. This guide does not delve into the hardware and software design of the Black Magic Probe either. Both the hardware and software of the Black Magic Probe are open source, and extensive information about its internals is available elsewhere on the internet (notably the GitHub project).

Instead, this guide aims at describing how to use the Black Magic Probe to debug embedded software running on an ARM Cortex micro-controller. It starts with an overview of the debugging pipeline, from the target micro-controller to the visualization of the embedded code on your workstation. Debugging embedded code usually implies remote debugging (with the code that is being debugged running on a different system than the debugger), but also cross-platform debugging. A broad understanding of these is helpful when making practical use of the Black Magic Probe.

The next chapters focus on setting up the hardware and software for the Black Magic Probe, and then a selection of GDB commands, with a special focus on those that are particularly useful for debugging embedded code.

Run-time tracing is an essential debugging technique for embedded systems, due to the real-time requirements that these systems often have. Coverage is split in two chapters: the first on the hardware and software support in the Black Magic Probe, and the second on generic techniques to perform tracing efficiently.

The Black Magic Probe can also be used for production programming of devices, through the same mechanism that GDB uses to download code to the target for purposes of debugging. This is the topic of another chapter, using both GDB and a separate utility.

License

This guide is written by Thiadmer Riemersma and copyright 2019 CompuPhase. It is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

The software associated with this guide is copyright 2019 CompuPhase and licensed under the Apache License version 2.

The Debugging Pipeline

Developing embedded software on small micro-controllers presents some additional challenges in comparison with desktop software. The software is typically developed on a workstation and then transferred to the target system. Accordingly, cross-compiling and remote debugging are the norm. Remote debugging implies the use of a hardware box to interface the workstation's USB, RS232 or Ethernet port to the micro-controller's debug protocol. On the ARM Cortex processors, the most common debug and Flash programming protocols are JTAG and SWD (Serial Wire Debug).

GDB Architecture

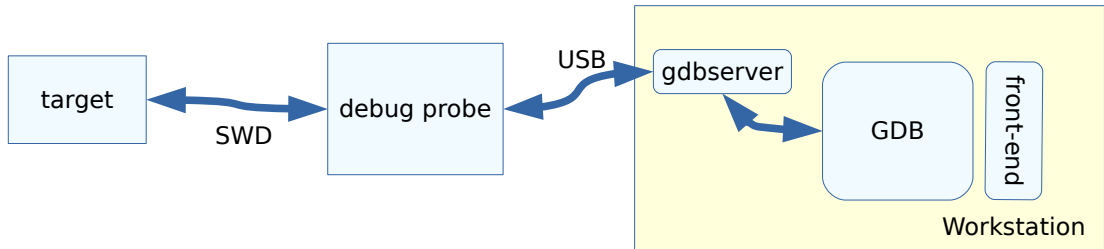
GDB is the GNU Debugger for programs built with GCC. It is also a debugger framework, with third-party front-ends and machine/protocol-specific back-ends.

GDB's user interface is, by today's standard, rather rudimentary, but GDB provides a "machine interface" to "front-ends", so that these front-ends can provide a (graphical) user interface with mouse support, source browser, variable watch windows, and so forth, while leaving symbol parsing and execution stepping to GDB. Most developers that use GDB actually run it hidden behind a front-end like Eclipse, KDbg, DDD, or the like. As a side note, a text-based front-end is built-in: TUI, and while it is an improvement over no front-end at all, TUI is not as stable as the alternatives.

To debug a different system than the one where the debugger runs on, GDB provides the *Remote Serial Protocol* (RSP). This is a simple text-based protocol with which GDB on the workstation communicates with a debugger "stub" on the target system. This stub acts as a server that GDB connects to, over an RS232 or Ethernet connection, and it is referred to as a *gdbserver*.

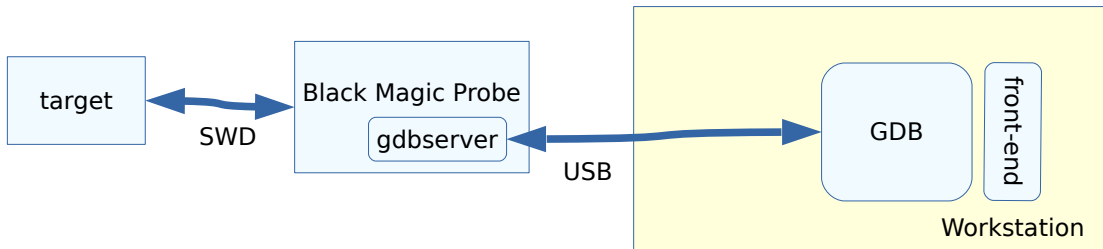
Directly implementing a *gdbserver* is impractical for micro-controllers such as the ARM Cortex M series, since the code developed for these small micro-controllers is typically monolithic and runs from Flash ROM. Micro-controllers typically provide hardware support for setting breakpoints and stepping through code, but make it available on a separate interface with dedicated pins for the task. On the ARM Cortex, this is *Serial Wire Debug* (SWD). To drive the serial wire protocol, a *debug probe* is needed: a hardware interface that drives the clock and data lines accord-

ing to the SWD protocol. Common debug probes are Segger J-Link and Keil ULINK-ME. The gdbserver functions as an interface to translate between GDB-RSP and the protocol of the hardware interface.



As is apparent, the debug data goes through a few hoops before the developer sees the code and data on the computer display in “GDB”.

The Black Magic Probe embeds gdbserver. One advantage of this design is that gdbserver has in-depth knowledge of the capabilities of the debug probe as well as what the debug probe has determined about the target. The only configuration that needs to be done in GDB is the (virtual) serial port of the Black Magic Probe (the USB interface of the Black Magic Probe is recognized as a serial port on the workstation).



Embedded Debugging: Points for Attention

On desktop computers and single-board computers, programs run in RAM. A debugger sets a breakpoint at a location by storing a special *software interrupt* instruction at that location (after first saving the instruction that was originally at that location). When the instruction pointer reaches the location, the software interrupt instruction causes the corresponding exception to be raised, which is intercepted by the debugger, which then halts the debuggee. The debugger also quickly puts the original instruction back into RAM, so that when you resume running the debuggee, it will execute the original instruction.

Contemporary micro-controllers often have limited SRAM, but a larger amount of Flash memory. The program for micro-controller projects therefore typically runs from Flash memory. For the purposes of running code, you may regard Flash memory as ROM; technically, it is rewritable, but rewriting is slow and needs to be done in full sectors. The upshot is: a debugger cannot set a breakpoint by swapping instructions in memory, because the memory (for practical purposes) is read-only.

The solution for the debugger is to team up with the micro-controller and tell the micro-controller to raise an exception if the instruction pointer reaches a particular address. This is called a hardware breakpoint (the former breakpoints are occasionally called *software* breakpoints). Unfortunately, micro-controllers provide only very few hardware breakpoints; rarely more than 8 and sometimes as few as 2.

A common architecture for an embedded application is one where the system responds to events (from sensors, switches or a databus) in a *timely* manner. The criterion “timely” regularly means: as quickly as possible, which then means that it is common to handle the event (and its response) in an interrupt. With crucial activity happening in various interrupt service routines, a puzzle that frequently pops up is that a global variable (or a shared memory buffer) takes on an unexpected value. A *watchpoint* can then tell you where in the code that variable got set. A watchpoint is a breakpoint that triggers on data changes. As with breakpoints, you will want hardware watchpoints, so that setting a watchpoint won’t interfere with the execution timing of the code.

Code that is stopped and stepped-through may not follow the same logic flow as code that executes in normal speed, because events or interrupts are missed or arrive in a different context (and those interrupts may set global variables or set semaphores). This change of behaviour may lead to bugs that “disappear” as soon as you try to debug them. The approach to tackle this situation is by tracing the execution path. Tracing can take multiple forms, from “printf-style” debugging to hardware support that records the entire execution flow of a session for post-mortem analysis.

A tracing technique that is unique to GDB is to add a command list to a (hardware) breakpoint, to immediately continue execution after recording that the breakpoint was passed. This way, you can evaluate which points in the code were visited and which were not, move the breakpoints to closer to the area where the bug is suspected and run another session — all without needing to edit and rebuild the code.

Requirements for Front-ends

GDB has powerful and flexible commands, but its console interface falls short of what is needed. Code is hard to follow if you only see a single line at a time. While you can routinely type the list command on the “(gdb)” prompt, it is clumsy and it distracts you from focusing on locating any flaws in your code. A front-end that provides a full-screen user interface is therefore highly desirable.

The front-end should not do away with the console, though. Some of the more advanced commands of GDB are not easily represented with icons and menu selections. This is especially true for remote debugging, and even more so for remotely debugging embedded systems. Without the ability to set or read the debug probe’s configuration, via the monitor command, your set-up depends on the defaults in the probe, which may not be appropriate for the target. Without the ability to set hardware breakpoints, you may not be able to debug code that runs from Flash memory; and as mentioned, running from Flash memory is the norm on small micro-controllers.

In a misguided attempt to increase “user friendliness”, KDbg, Nemiver and the Eclipse hide the GDB console (Eclipse has a console tab in its “debug mode”, but it is not the GDB console). Fortunately, this still leaves several front-ends to choose from in Linux: DDD, cgdb, gdbgui work well, and GDB’s internal TUI is adequate. The TUI is not available on Windows builds of GDB, and DDD and cgdb have not been ported to Windows. However, gdbgui works well and two (commercial) alternative front-ends for Microsoft Windows are WinGDB and VisualGDB (both function as plug-ins to Microsoft’s Visual Studio). Finally, a GDB front-end specifically designed for the Black Magic Probe exists (as a companion utility to this guide); it is covered extensively in section [The BlackMagic Debugger Front-end](#) on page 27.

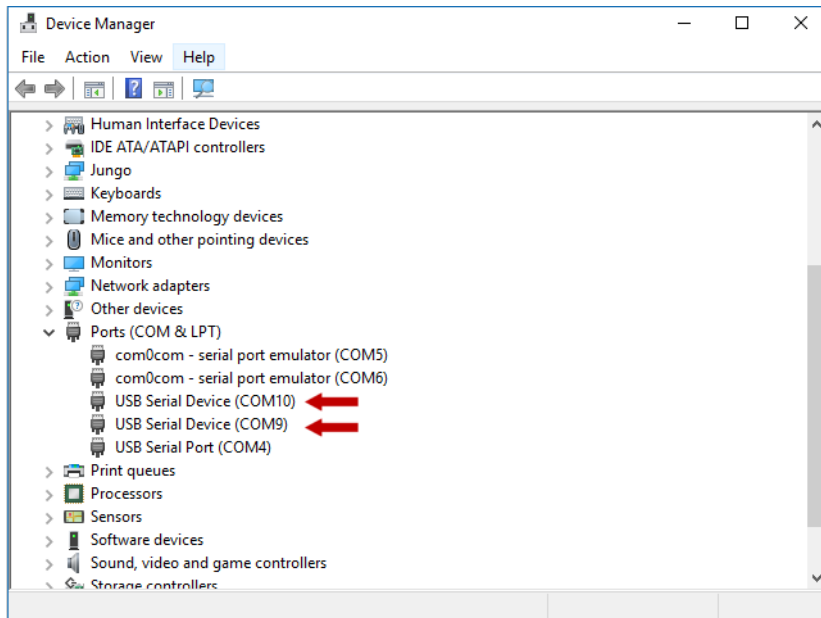
Setting up the Black Magic Probe

Microsoft Windows

On connecting the Black Magic Probe to a USB port on a workstation, four devices are added. The principal ones are two (virtual) serial ports (COM ports). One of these is for gdbserver and the other is the generic 3.3V TTL UART that the Black Magic Probe also provides. The other two are WinUSB devices for firmware update (via the DFU protocol) and trace capture.

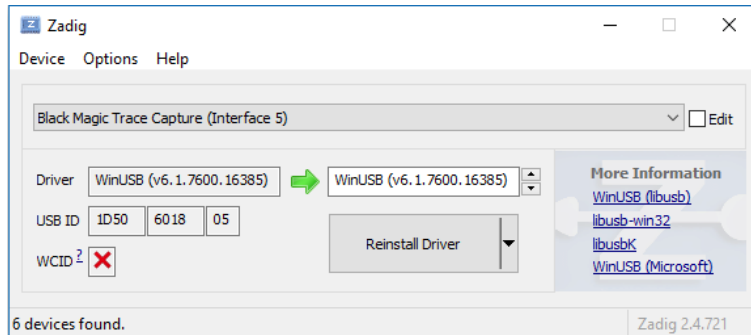
On Windows 10, no drivers are needed. Earlier versions of Microsoft Windows require that you install an “INF” file that references the CDC class driver that Microsoft Windows has already installed (“usbser.sys”). A suitable INF file can be found on the site of Black Sphere Technologies, as well as with this guide.

You can find out which COM ports were assigned to the Black Magic Probe in the Device Manager, where they are listed under the item “Ports (COM & LPT)”. Alternatively, you can run the bmscan utility on the command line (this is one of the utilities that comes with this guide).



Note that in Windows 10, as we are using the built-in CDC driver, the name for the Black Magic Probe interfaces is the generic “USB Serial Device” (see the red arrows in the picture above).

For trace capture and for firmware update, the two *generic* interfaces of the Black Magic Probe must be registered as a WinUSB device. The most convenient way to do so is by running the free utility “Zadig” by Akeo Consulting. The utility can currently be downloaded from <https://zadig.akeo.ie/>



You need to register both interfaces 4 (“Black Magic Firmware Upgrade”) and 5 (“Black Magic Trace Capture”) separately. Both are on USB ID 1D50/6018. You may need to first select List All Devices from the Options menu, to make the Black Magic Probe interfaces appear in the drop down list of the Zadig utility.

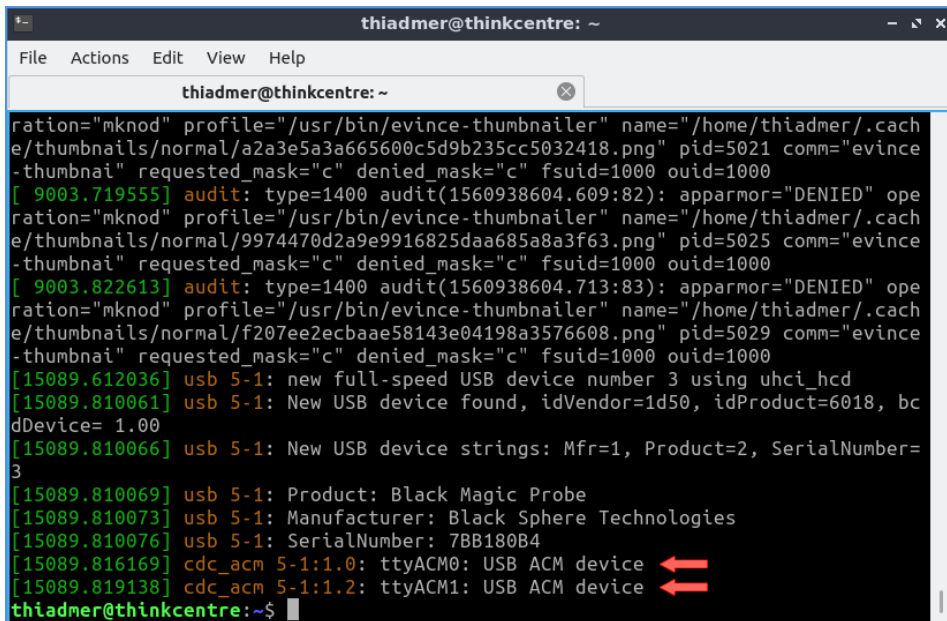
For firmware update, you should also register the DFU interface (in DFU mode, USB ID 1D50/6017) as a WinUSB device. This interface is hidden until the Black Magic Probe switches to DFU mode. To force the Black Magic Probe in DFU mode, keep the push-button (next to the USB connector) pressed while connecting it to the USB port of the workstation. The red, orange and yellow LEDs will blink in a pattern as a visual indication that the Black Magic Probe is in DFU mode. When you launch the Zadig utility at this point, the interface will be present. Note: in DFU mode, the Black Magic Probe has USB ID (VID:PID) 1D50:6017, in run mode it has USB ID 1D50:6018.

Linux

After connecting the Black Magic Probe to a USB port, two virtual serial ports appear. One of these is for gdbserver and the other is the generic 3.3V TTL UART that the Black Magic Probe also provides. Since the Black Magic Probe implements

the CDC class, and Linux has drivers for CDC class devices built-in, no drivers need to be set up.

The device paths for the serial ports are `/dev/ttyACM*` where the “*” stands for a sequence number. You can find out which `ttyACM` devices are assigned to the Black Magic Probe by giving the `dmesg` command (in a console terminal) shortly after connecting the Black Magic Probe. Alternatively, you can run the `bmscan` utility from inside a terminal (`bmscan` is a companion tool to this guide).

A screenshot of a terminal window titled 'thiadmer@thinkcentre: ~'. The terminal shows the output of the 'dmesg' command, which lists system events. Key events include: 'mknod' for '/home/thiadmer/.cache/thumbnails/normal/a2a3e5a3a665600c5d9b235cc5032418.png' (pid=5021), 'mknod' for '/home/thiadmer/.cache/thumbnails/normal/9974470d2a9e9916825daa685a8a3f63.png' (pid=5025), 'mknod' for '/home/thiadmer/.cache/thumbnails/normal/f207ee2ecbaae58143e04198a3576608.png' (pid=5029), and 'usb 5-1: New full-speed USB device number 3 using uhci_hcd'. The most relevant part of the output is the identification of the Black Magic Probe as a USB ACM device. The last two lines of the output are highlighted with red arrows pointing to them: '[15089.816169] cdc_acm 5-1:1.0: ttyACM0: USB ACM device' and '[15089.819138] cdc_acm 5-1:1.2: ttyACM1: USB ACM device'. The terminal prompt is 'thiadmer@thinkcentre:~\$'.

To be able to access the serial ports, the user must be included in the `dialout` group (unless the user is `root`). To add the current user to the group, use:

```
sudo usermod -a -G dialout $USER
```

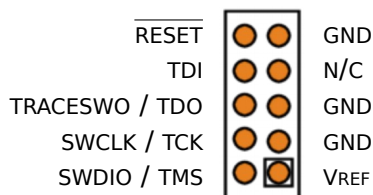
After this command, you need to log out and log back in, for the new group assignment to be picked up.

No driver needs to be installed for the firmware update and trace capture interfaces, but if you wish to use those features with needing `sudo`, a file with `udev` rules must be installed. For firmware update, it may not be a burden to use `sudo`, as you will update the Black Magic Probe’s firmware only occasionally, but trace capture is a valuable debugging tool for everyday use.

This guide comes with the file `50-blackmagicprobe.rules`, which you can copy into the `/etc/udev/rules.d` directory. It allows any user to access the trace capture interface of the Black Magic Probe.

Connecting the Target

The Black Magic Probe has a 2×5-pins 1.27mm pitch IDC header. This is the MIPI10 connector for JTAG and SWD. If your target board has the same connector, the two can be readily connected with the provided ribbon cable. Note that the MIPI10 connector is also called the Cortex Debug header.



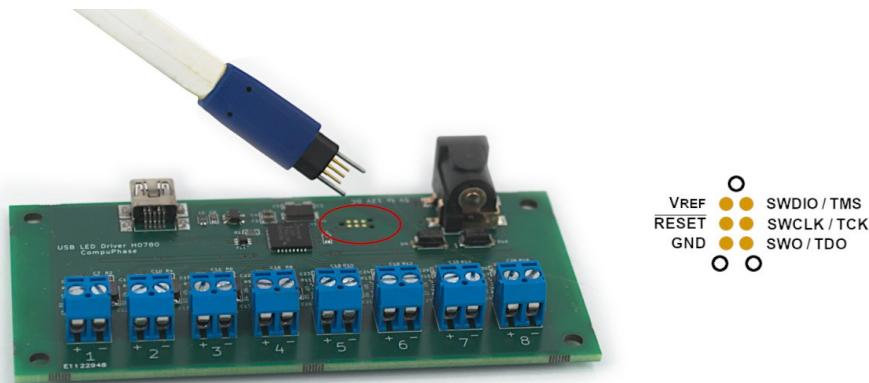
For target boards that do not have exactly this 2×5-pins header, you can use the break-out board that is also provided with the Black Magic Probe. This break-out board has the 2×5-pins 1.27mm pitch MIPI10 connector on one side and a 7-pins 2.54mm pitch IDC header (single row) on the other. Note that the MIPI10 connector on the break-out board lacks a polarity notch; the ribbon-cable should be plugged such that the red wire is toward the side with the text (on the board) “JTAG 7-pin adapter”.

Of the pins on the debug connector, SWCLK, SWDIO and GND are essential. These must always be connected to the target. The $\overline{\text{RESET}}$ pin is strongly recommended, especially for downloading firmware to Flash memory. The VREF pin should in most cases be connected as well, because the Black Magic Probe uses the target’s voltage level at this pin to shift the level on the signal lines to this same voltage. The alternative is to drive VREF to 3.3V from the Black Magic Probe (see the `monitor tpwr` command on page 25). Finally, the TRACESWO pin is for debug tracing, which requires support code in your firmware, and the TDI line is used for JTAG scan, not for debugging.

Our favourite debug connector is the decal for the tag-connect cable. This cable has a plug with six pogo-pins, plus three fixed pins that serve to align the plug.

The benefit of the tag-connect cable is that it requires less space on the target board than for most other connectors, and that the matching “connector” on the target board is simply a decal. For the target board, the added cost for the pro-

gramming/debugging connector is therefore zero. The tag-connect lacks the TDI pin, and hence the tag-connect cable is not suitable for JTAG scanning purposes.



See also section [Target Board Design for Debugging & Programming](#) on page 15 for additional tips when designing a new PCB.

Checking the Setup

When the Black Magic Probe is connected to a USB port, the green and orange LEDs (labeled “PWR” and “ACT” respectively) should be on. In Microsoft Windows, the ACT LED is dimly on, in Linux, it is bright at first but goes dim after some time.

If you have not checked which serial port the Black Magic Probe uses for its gdb-server, run `bmscan` on the command line.

```
d:\Tools>bmscan
```

```
Black Magic Probe found:  
gdbserver port: COM9  
TTL UART port:  COM10
```

To check whether the drivers were installed correctly, launch GDB from the command line. You should be using the GDB that was build for the architecture that matches the micro-controller (typically `arm-none-eabi`). On the “(gdb)” prompt, type (where you replace “*port*” with the COM port for gdbserver):

```
(gdb) target extended-remote port
```

In Microsoft Windows, when the port is above 9, the string “`\\.\\`” must be prefixed to the port name. So COM port 10 is specified as “`\\.\\com10`”. In Linux, the device path for the port must be used, like in “`/dev/ttyACM0`”.

There is no need to configure the baud rate or other connection parameters; what the operating system presents as a serial port is a USB connection running at 12 Mbits/s, irrelevant of what baud rate it is configured to.

After setting the remote port in GDB, the orange LED (“ACT”) will increase in brightness. In fact, this LED on the Black Magic Probe responds to the DTR signal set by the debugger; this was a physical line on the RS232 port, but now just a command on a virtual serial port.

The next step is to scan for the target micro-controller. There are two ways to do this: `swdp_scan` for micro-controllers supporting SWD and `jtag_scan` for devices supporting only JTAG.

```
(gdb) monitor swdp_scan
Target voltage: 3.3V
Available Targets:
No. Att Driver
1      LPC11xx
```

The output shows the driver name for the micro-controller. Note that multiple devices may be returned, for both the SWD scan (using the SW-DP protocol) and the JTAG scan (JTAG devices may be daisy-chained).

The command also shows that the target is not yet “attached” to gdbserver (otherwise, there would be a “*” in the “Att” column of the target list). Attaching the target is done with the `attach` command.

```
(gdb) attach 1
Attaching to Remote target
```

At this point, the Black Magic Probe is attached to GDB and you can proceed to download firmware and/or to start debugging it, which is the topic of the next chapter starting at page 17.

Running Commands on Start-up

The above commands have to be repeated on each debugging session. On start-up, GDB reads a file called `.gdbinit` and executes all commands in it. This file is read from the “home” directory in Linux, and from the path set in the `HOME` environment variable in Microsoft Windows (this environment variable is not set by default, so you may need to create it).

Following the examples in this chapter, a suitable `.gdbinit` file could be:

```
target extended-remote COM9
monitor swdp_scan
attach 1
```

If the Black Magic Probe is not yet connected when starting GDB, or if the operating system decided to assign the Black Magic Probe to a different serial port, the above start-up code will fail. GDB quits parsing the `.gdbinit` file on the first error, so the remainder in the file is not executed either. Our recommendation is, therefore, to only add user-defined commands in `.gdbinit`, so that you have a shorthand for quickly connecting to the Black Magic Probe.

```
define bmconnect
  if $argc < 1 || $argc > 2
    help bmconnect
  else
    target extended-remote $arg0
    if $argc == 2
      monitor $arg1 enable
    end
    monitor swdp_scan
    attach 1
  end
end

document bmconnect
  Attach to the Black Magic Probe at the given serial port/device.
  bmconnect PORT [tpwr]
  Specify PORT as COMx in Microsoft Windows or as /dev/ttyACMx in Linux.
  If the second parameter is set as "tpwr", the power-sense pin is driven to
  3.3V.
end
```

Other settings can be added to the `.gdbinit` too. If you have per-project settings, these can be in a secondary `.gdbinit` file in the current directory. GDB will load the “current directory” `.gdbinit` file when adding the following command in the “home” `.gdbinit` file:

```
set auto-load local-gdbinit
```

Target Board Design for Debugging & Programming

Like almost any other debug probe, the Black Magic Probe can be used for Flash memory programming as well as for debugging the code that runs from Flash

memory. For the development cycle, this is very convenient: you build the code and then load it into the target and into the debugger in a single flow.

However, it is common for micro-controllers that several functions are shared on each single pin. If the code redefines one of the pins for SWD to some other function, by design or by accident, the debugging interface will stop functioning. If the code redefines the pins quickly after a reset, the Black Magic Probe does not have a chance to regain control of the SWD interface, even after a reset. The result is that not only the code cannot be debugged any more, but also that no new code can be flashed into the micro-controller.

The LPC series of micro-controllers from NXP have a pin that forces the micro-controller into *bootloader* mode when it is pulled low on reset (or on power cycle). The STM32 series from STMicroelectronics have two boot pins for the same purpose. Bootloader mode is designed for Flash programming over a serial port or USB, but the side effect is that it blocks the firmware from running. As a result, the pins for SWD have not been redefined and you can now start GDB and attach to the target (after which you can upload new firmware). The recommendation for PCBs with an LPC or STM32 micro-controller is therefore to branch out the “boot” pin(s) to a jumper or a tiny push-button, so that you can recover from an accidental pin redefinition.

As an aside, if the pin redefinition is by design, the advice is to wait several seconds before doing so. If new firmware needs to be downloaded into the micro-controller, this gives you a time slot (between reset and the pin redefinition) to attach the debugger to the micro-controller. This is advice for firmware design, however, not PCB design.

Debugging Code

Debugging code for embedded systems has its own challenges, in part due to the way that micro-controller projects differ from typical desktop applications. Some commands of GDB are skipped over in almost every book because they are not relevant for desktop debugging. This chapter focuses on the commands that are relevant for the Black Magic Probe and ARM Cortex targets. It is therefore more an addendum to books/manuals on debugging with GDB, than a replacement of them.

As mentioned in [The Debugging Pipeline](#) (page 5), you will probably prefer a front-end to do any non-trivial debugging. Below is a screen-capture of gdbgui connected to the Black Magic Probe, and ready to debug “blinky”.

The screenshot shows the gdbgui web interface in a browser. The address bar shows the URL `127.0.0.1:5000`. The interface includes a toolbar with buttons for loading a binary, fetching disassembly, reloading a file, and jumping to a line. The main area displays the source code of `blinky.c`, which is a simple LED blink program. The code is as follows:

```
32 int main(void)
33 {
34     uint32_t led_ioport, led_iobit;
35
36     if (SysTick_Config(SystemCoreClock / 1000)) { /* Setup SysTick Timer for 1000ms */
37         while (1); /* Capture error */
38     }
39
40     led_ioport = IOPORT(PIN_LED);
41     led_iobit = IOBIT(PIN_LED);
42     LPC_GPIO->DIR[led_ioport] |= led_iobit; /* LED = output */
43
44     for ( ;; ) {
45         LPC_GPIO->SET[led_ioport] = led_iobit; /* turn LED on */
46         mdelay(500);
47         LPC_GPIO->CLR[led_ioport] = led_iobit; /* turn LED off */
48         mdelay(500);
49     }
50 }
```

On the right side, the 'threads' panel shows the current thread as 'Remote target, id 1, stopped'. Below it, the 'local variables' panel shows the current state of variables: `curTicks` is 32000, `dlyTicks` is 500, and `dlyTicks@entry` is 500. The 'memory' panel shows the start and end addresses of the memory being debugged.

At the bottom, the console output shows the following commands and responses:

```
No. Att Driver
1 LPC11xx
attach 1
attach 1
Attaching to program: d:\products\usbkey\source\obj\blinky.elf, Remote target
0x0000033a in mdelay (dlyTicks=dlyTicks@entry=500) at blinky.c:27
27 while ((msTicks - curTicks) < dlyTicks)
(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

The `gdbgui` front-end is a fairly thin graphical layer over GDB: you have to type most commands in the console. However, the limited abstraction from GDB is actually an advantage. Front-ends typically aim at desktop debugging, and so the set of commands specific to embedded code are not wrapped in dialogs and popup menus.

Yet, while we recommend the use of a front-end with GDB, the commands and examples in this chapter use the GDB console. While a front-end may provide a more convenient way to perform some task, each will have its own interface for it. The GDB console is a common denominator for all GDB-based debuggers.

Prerequisite Steps

On every launch of GDB, it has to connect to the Black Magic Probe, scan for the attached target and attach to it. Unless you are using the `bmdebug` front-end that handles these steps automatically, they have to be given through the console.

```
(gdb) target extended-remote COM9
Remote debugging using COM9
(gdb) monitor swdp_scan
Target voltage: 3.3V
Available Targets:
No. Att Driver
1      LPC11xx
(gdb) attach 1
Attaching to Remote target
0x0000033a in ?? ()
```

These commands can be wrapped in a user-defined command in a `.gdbinit` file, see [Running Commands on Start-up](#) on page 14. In that case, you would type only a single command:

```
(gdb) bmconnect COM9
Target voltage: 3.3V
Available Targets:
No. Att Driver
1      LPC11xx
0x0000033a in ?? ()
```

Loading a File and Downloading it to the Target

The first step in running code in a debugger, is to generate debug symbols while building it. The GNU GCC compiler (and linker) use the command line option `-g` for that purpose.

You can specify the executable file to debug on the command line when launching GDB, but alternatively, you set it with the `file` command. The filename may be a relative or full path, with a `/` as the directory separator (this is of notice to users of Microsoft Windows, where directories are usually separated with a `"\"`).

```
(gdb) file blinky.elf
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from blinky.elf...done.
(gdb) load
Loading section .text, size 0x7da lma 0x0
Start address 0xd8, load size 2008
Transfer rate: 6 KB/sec, 669 bytes/write.
```

Note that the GDB `load` command downloads only the executable code to the target. The ELF file contains debug symbols, which makes the executable file much larger than when the code is compiled without debugging information. However, the size of the code that is downloaded to the target remains the same; the debug symbols are not transferred.

For the LPC micro-controller series, an additional step is recommended before the `load` command. NXP designed the micro-controllers such that the bootloader always runs on reset (or power-up). The bootloader then samples the boot pin, verifies whether there is valid code in the first Flash sector, and jumps to it if it checks out. The conflict is: the ARM Cortex starts running at the reset vector stored at address 0, which must initially point to ROM (where the bootloader resides) and then to Flash memory (where the user code sits). The LPC micro-controllers have the feature to remap address range 0...511 to either Flash, RAM or ROM via either the `SYSMEMREMAP` or the `MEMMAP` register. According to the documentation, after a reset the register is initialized such that address 0 maps to Flash memory. However, that is not what happens: the `SYSMEMREMAP` (or `MEMMAP`) register is initially 0 (remap to bootloader ROM) and the bootloader then modifies it to map to Flash before jumping to the user code in Flash. However, when the micro-controller is halted by the debug probe, `SYSMEMREMAP` is still 0. Then, if you download new code in the micro-controller, the bottom 512 bytes will be sent to ROM, and be lost.

The fix is to force mapping the SYMEMREMAP register to 2 from GDB (as is apparent, SYMEMREMAP is a memory-mapped register). The example below is for the LPC8xx, LPC11xx, LPC12xx and LPC13xx series.

```
set mem inaccessible-by-default off
set {int}0x40048000 = 2
```

For convenience, the above can be wrapped in a user-defined command in the .gdbinit file, see [Running Commands on Start-up](#) on page 14:

```
define mmap-flash
  set mem inaccessible-by-default off
  set {int}0x40048000 = 2
end

document mmap-flash
  Set the SYMEMREMAP register for NXP LPC devices to map address 0 to Flash.
end
```

You would then give the command `mmap-flash` before using the `load` command. The address of the SYMEMREMAP register (and the value to set it to) is different in other series in the LPC micro-controller range, and the above snippet therefore needs to be adapted for micro-controller other than the LPC8xx, LPC11xx, LPC12xx and LPC13xx series. A more complete version of the above user-defined command is in the .gdbinit file that comes with this guide.

To verify that the code in the micro-controller is the same as the code loaded in GDB, you can use the `compare-sections` command. This command also lets you verify that downloading code was successful.

```
(gdb) compare-sections
Section .text, range 0x0 -- 0x7d8: matched.
```

There is a caveat with the LPC series of micro-controllers from NXP: these micro-controllers require a checksum in the vector table at the start of the Flash code. The checksum can only be calculated at or after the link stage, but the GNU linker is oblivious of this requirement. Instead, firmware programmers calculate and set the checksum while downloading, and the Black Magic Probe is no exception. The upshot is that `compare-sections` will now always return a mismatch on the first section, since its contents were changed on the flight while downloading it.

To fix `compare-sections`, the checksum must be set in the vector table in the ELF file after the link phase. The Black Magic Probe will calculate it again, despite that it is already set, but that does no harm, since it comes to the same value. After

downloading, the code in the micro-controller will be identical to the code in the ELF file.

```
elf-postlink lpc11xxx blinky.elf
```

The program elf-postlink is a one of the utilities that come with this guide.

Starting to Run Code

The run command starts to run the loaded code from the beginning. If you have not set any breakpoints, the code runs until it is interrupted through Ctrl+C. The start command sets a temporary breakpoint at function main and then runs; the program will therefore stop at main.

```
(gdb) start
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Temporary breakpoint 1 at 0x348: file blinky.c, line 33.
Starting program: c:\Source\blinky\blinky.elf
Note: automatically using hardware breakpoints for read-only addresses.

Temporary breakpoint 1, main () at blinky.c:33
33      {
```

Note the mention of the automatic use of hardware breakpoints. With the help of the Black Magic Probe, GDB indeed inserts a hardware breakpoint on the break command.

Listing Source Code

The set of commands is a subset of the most common commands and parameters for listing the source code of a target.

list line	Show the source code around the given line number in the current source file.
list file:line	Show the source code in the file with the name in the first parameter, and around the line number in the second parameter.
list function	Show the source code starting at the given function.
list	Show the next lines (below the current position). You can optionally add a + as a parameter ("list +").

<code>list -</code>	Show the preceding lines (above the current position).
<code>info sources</code>	Show the names of the source files for the target executable.

Stepping and Running

These are the basic commands needed for debugging. Several of these commands were already informally introduced in earlier sections.

<code>start</code>	Start or re-start the program and break at function <code>main</code> . If no function “ <code>main</code> ” exists, it is the same as the <code>run</code> command.
<code>run</code>	Start or re-start the program (from the beginning).
<code>continue / c</code>	Continue running (from the current execution point). A count may follow the command, but it is only relevant if code stopped due to a breakpoint. If present, the breakpoint is ignored the next “count” times it is hit. This is particularly useful in when the breakpoint is inside a loop: the command <code>continue 10</code> will run 10 more iterations before stopping at the breakpoint again.
<code>step / s</code>	Step a single source line, step <i>into</i> functions if the current execution point is at line with a function call. A count may follow the command. If present, the command repeats the step “count” times.
<code>next / n</code>	Step a single source line, step <i>over</i> functions (if there is a function call at the current execution point). A count may follow the command. If present, the command repeats the step “count” times.
<code>until / u</code>	Run until a source line is reached that is below the current line (this command is intended for stepping out of loops). Alternatively, you can set a line number after the <code>until</code> command, and then it runs until that line is reached.
<code>finish / fin</code>	Step out of the current function and stop at the location from where it was called.

Breakpoints and watchpoints

<code>break line</code> <code>b line</code>	Set a breakpoint at the line number in the current source file.
<code>break file:line</code> <code>b file:line</code>	Set a breakpoint at the line number in the specified source file.

<code>break function</code> <code>b function</code>	Set a breakpoint at the start of the named function.
<code>tbreak</code>	Sets a one-time breakpoint, which auto-deletes itself as soon as it is reached. The <code>tbreak</code> command takes the same parameter options as the <code>break</code> command.
<code>watch expr</code>	Set a watchpoint, which causes a break as soon as the expression changes. In practice, the expression is typically the name of a variable, so that GDB halts execution of the program as soon as the variable changes.
<code>info break</code>	Show the list of breakpoints and watchpoints, together with the sequential index numbers (sometimes called the breakpoint IDs) that each breakpoint got assigned.
<code>delete</code>	When given without parameters, this command deletes all breakpoints. Otherwise, if one or more numbers follow the command (separated by spaces), the command deletes the breakpoints with those index numbers.
<code>clear</code>	Without parameters, this command deletes the breakpoint that is at the current code execution point. The primary use is to delete the breakpoint that was just reached.
<code>clear line</code> <code>clear file:line</code> <code>clear function</code>	Delete a breakpoint on the given line or function. It allows the same options as the <code>break</code> command.
<code>disable idx ...</code>	Disables the breakpoints with the given index numbers. There may be one or more numbers on the command list (separated by spaces).
<code>enable idx ...</code>	Enables the breakpoints with the given index numbers. There may be one or more numbers on the command list (separated by spaces). You may also use <code>enable</code> once to enable the breakpoints, but <code>disable</code> them when they are reached.
<code>cond idx expr</code>	Attaches a condition to the breakpoint with the given index number. The condition is what you would write between the parentheses of an “if” statement in the C language. For example: <code>cond 3 count == 5</code> causes breakpoint 3 to only halt execution when variable <code>count</code> equals 5 (assuming, of course, that variable <code>count</code> is in scope). When the expression is absent on this command, the condition is removed from the breakpoint (but the breakpoint stays valid).
<code>command idx</code>	Sets a command list on the given breakpoint. These commands are

...	executed when the breakpoint is reached. It can be used, for example, to automatically print out the stack trace on arriving at the breakpoint. See section Tracing with Command List on Breakpoints on page 42.
end	

For embedded development, enabling and disabling breakpoints (and watchpoints) is all the more useful, because hardware breakpoints and hardware watchpoints are a scarce resource. Most Cortex-M micro-controllers offer 6 hardware breakpoints and 2 hardware watchpoints. What counts, for the Black Magic Probe is not the number of breakpoints that have been set, but the number that is active. When you need more breakpoints than the micro-controller offers, you keep them defined, but disable the ones that are not immediately relevant for the next step in debugging the code.

Examining Variables and Memory

print var p var	Show the contents of the variable. GDB can parse C-language expressions to show array elements or dereferenced variables, like in: print var[6] show the value of an array element print *ptr dereference the pointer and show the value.
info args	Show the names and values of the function arguments.
info locals	Show the names and values of all local variables.
ptype var	Show the type information of the variable.
display var disp var	Watch the variable. Show the variable's value each time that the execution is halted.
undisplay num undisp num	Remove the watch with the given sequence number.
x address	Display the memory at the given address.
set var=value set addr=value	Set the variable to the value, or store the value at the address. You can use C-style type-casts on the address to specify the size of the memory field.

The Call Stack

backtrace bt	Show a list with the call-stack that lead to the current execution point.
up	Move to the frame one higher in the call-stack, which is the frame that contains the call to the current frame.

down	Move back to a lower frame.
frame <i>idx</i>	Move to the given frame index (the <code>backtrace</code> command prints these index numbers).

After changing to a different stack frame, commands like `info locals` will reference to the local variables of that frame. This may help you in determining what conditions caused the call to the function currently stopped in.

Debug Probe Commands

GDB has a pass-through command to configure or query a gdbserver implementation: `monitor`. Whatever follows the keyword `monitor` is passed to the gdbserver, in our case the embedded gdbserver in the Black Magic Probe.

The supported monitor-commands are listed below. Note that some of these commands are only available on particular micro-controller series; if this is the case, the applicable micro-controller series is noted.

help	Show a summary of the commands (essentially this list).
version	Show the current version of the firmware and the hardware.
jtag_scan	Scan the devices on the JTAG chain.
swdp_scan	Scan for <i>Serial Wire Debug</i> devices (using the SW-DP protocol). The command prints the I/O voltage and the list of targets. See also the <code>tpwr</code> command (below) for the I/O voltage and the <code>targets</code> command for the device list.
traceswo traceswo <i>rate</i>	Enable the SWO capture pin to for trace capture. The <code>rate</code> parameter is the bitrate of the SWO trace protocol in asynchronous mode. When this parameter is present, asynchronous encoding is set, if this parameter is absent, manchester encoding is set.
targets	Show the detected targets. This is the same list as the one returned by the <code>jtag_scan</code> and <code>swdp_scan</code> commands. For each detected micro-controller, it displays the driver (the driver is often specific to a micro-controller family).
tpwr enable tpwr disable	Enables or disables driving the VCC pin on the 2×5 pin header to 3.3V. See page 12 for the pin-out of the connector. When the Black Magic Probe drives the VCC pin, it can power the target (maximum current: 100mA). The VCC pin must always be driven, either by the target or by the Black Magic Probe, because the voltage at this pin is also used by level shifters on the logic pins on the connector. The default is that the VCC pin must

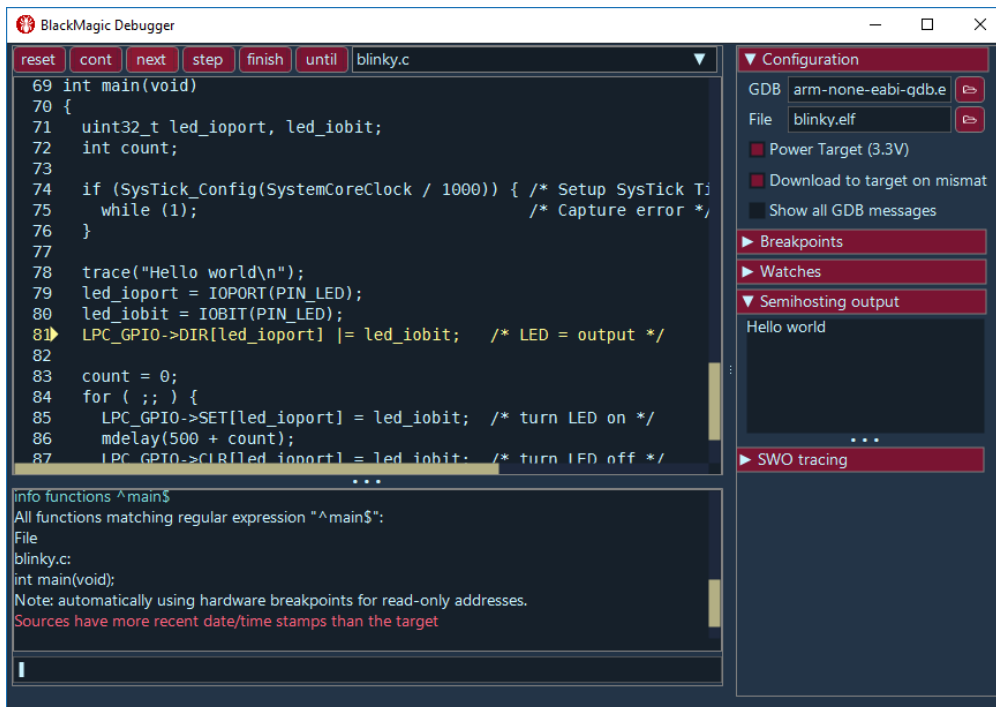
	be driven by driven by the target. A special case is to not wire the VCC pin between the Black Magic Probe and the target. The VCC pin must now also be driven by the Black Magic Probe, and the level shifters are therefore set to 3.3V TTL levels.	
connect_srst	Enables or disables a reset on connection.	
hard_srst	Resets the target by briefly pulling the $\overline{\text{RESET}}$ pin low on the 2×5 pin header (see page 12 for the connector).	
morse	When the Black Magic Probe encounters an error that it cannot handle otherwise, it will start to blink the red LED (labeled “ERR”) in a Morse code pattern. In case your Morse code mastery is a little rusty, the morse command returns the error message in plain text on the GDB console. But in fact, the only such error message is “TARGET LOST.”	
vector_catch	Break on specific exceptions. The first parameter must be enable or disable. The second parameter must be the exception for which the “catch” must be enabled or disabled. It is one of: hard Hard fault. int bus Bus fault. stat chk nocp mm Memory Manager fault. reset Reset vector.	ARM Cortex-M
erase_mass	Erase entire flash memory.	LPC17xx LPC4300 Cortex-M4 EFM32 Gecko nRF51xxx series SAMD STM32Fxx, STM32L4xx
erase_bank1	Erase entire flash memory in bank 1.	STM32L4xx
erase_bank2	Erase entire flash memory in bank 2.	STM32L4xx
reset	Reset target.	LPC4300 Cortex-M4
mkboot	Make flash bank bootable. The parameter is the bank number, 0 or 1.	LPC4300 Cortex-M4
serial	Print the micro-controller serial number.	EFM32 Gecko SAMD

unsafe	Allow programming the security byte. The parameter must be enable or disable.	<i>Kinetis</i>
read	Read target device parameters. The parameter is one of: help Show brief help on the command. hwid The hardware identification number. fwid The pre-loaded firmware ID. deviceid The unique device ID. deviceaddr The device address.	<i>nRF51xxx series</i>
gpnvm_get	Get value of the GPNVM register.	<i>SAM3N, SAM3S, SAM3U, SAM3X SAM4S</i>
gpnvm_set	Set bit in the GPNVM register. The first parameter is the bit number. The second parameter is the value for the bit (0 or 1).	<i>SAM3N, SAM3S, SAM3U, SAM3X SAM4S</i>
lock_flash	Lock Flash memory against accidental change.	<i>SAMD</i>
unlock_flash	Unlock Flash memory.	<i>SAMD</i>
user_row	Print the user row from Flash.	<i>SAMD</i>
mbist	Run the “Memory Built-In Self Test” (MBIST).	<i>SAMD</i>
option	Set option bytes. The first syntax is option erase to erase the entire Flash memory. The second syntax is option address value which stores a value at the given address.	<i>STM32Fxx, STM32L0x, STM32L1x, STM32L4xx</i>
eeeprom	Set values in EEPROM (non-volatile memory). The first parameter is one of: byte 8-bit value. halfword 16-bit value. word 32-bit value. The second parameter is the address in the EEPROM. The third parameter is the value (with the size as specified in the first parameter).	<i>STM32L0x, STM32L1x</i>

The BlackMagic Debugger Front-end

The bmdebug utility is a front-end for GDB that is designed for the Black Magic Probe. On start-up, it locates the Black Magic Probe and attaches to it, optionally provides power to the target, and verifies whether the code in the micro-controller

matches the file loaded in GDB and downloads it into the micro-controller on a mismatch. The [Prerequisite Steps](#) described on page 18 are handled automatically. Another distinguishing feature of bmdebug is that it combines traditional debugging with run-time tracing.



Starting up

After loading an ELF file, bmdebug stops at function main in that code. If the ELF file lacks a main function, you will want to set a breakpoint at some code of interest before giving the run command (or pressing the “Go” button).

Unlike the bmflash utility (see page 54), the bmdebug front-end is not able to calculate the header checksum for the LPC micro-controller family *before* uploading it. This is because bmdebug is based on GDB (it is a “front-end”), whereas bmflash is independent of GDB. As a consequence, GDB (and thereby bmdebug) will *always* see a CRC mismatch between the ELF file loaded in the debugger and the one downloaded in the target, and re-download it at every run. To avoid this, one option is to run elf-postlink on the ELF file as part of the build process; the other option is to disable automatic download of the ELF file in bmdebug (option “Download to target on mismatch” in the “Configuration” section in the sidebar). See the

discussion of the `elf-postlink` utility on page 20 for more information on the checksum for LPC micro-controllers.

Source View

The source view shows the execution point with a rightwards pointing triangle in the left margin. The execution point is the line that will be executed next when continuing execution.

The “cursor line” in the source view is highlighted. You can freely move the cursor line. Every time the target micro-controller stops, `bmdebug` sets the cursor line to the execution point. Alternatively, you can also run to the cursor line with the button `Until (F7)`.

When stepping through code, the source view automatically switches to the source file that the execution point is in. You can select any source file from the drop-down list in the button bar above the source view. Alternatively, you can use the `list` command in the console line (see section [Listing Source Code](#) on page 21). For switching to another source file, the file extension may be omitted. For example, the following command will load the file `blinky.c` or `blinky.cpp` (whichever is available).

```
list blinky
```

The standard keys for scrolling through the source are available (`Arrow Up/Down`, `Page Up/Down`, `Ctrl+Home` and `Ctrl+End`). The `Ctrl+R` key combination scrolls though earlier commands on the command line.

An additional command is provided to search for text in the source file that is displayed.

<code>find text</code>	Finds the first occurrence of the text starting from the cursor line. The search wraps from the bottom of the text to the top. The key combination <code>Ctrl+F</code> inserts the <code>find</code> command on the edit line.
<code>find</code>	Repeats the last search. Function key <code>F3</code> is a shorthand for this action.

Stepping and Running

The button bar above the source code view has the essential functions for running and stepping through code. The names of most buttons reflect the GDB command that it executes: the `Step` button executes a `step` command and the `Finish` button lets GDB execute a `finish` command.

The exception is the Reset button, which reloads and restarts the target firmware, and then runs up to main.

All buttons have a function key associated with them. For example, F10 does a next command (step over) and F11 does a step command (step into). A tooltip on each button shows the equivalent function key.

Breakpoints


You can set a breakpoint by clicking on the left of a line in the *source view*, or with a break command on the console. When clicking in the source view, clicking a second time on an existing breakpoint disables the breakpoint (rather than removing it). To remove the breakpoint, you need to click on it a third time (while staying on the line with the mouse cursor). The breakpoints can also be toggled between enabled and disabled in the *breakpoints view*. When debugging code in Flash ROM, you can set as many breakpoints as you like, but only a limited number can be enabled at any time (most Cortex-M micro-controllers provide 6 hardware breakpoints).

The break commands (see [Breakpoints and watchpoints](#) on page 22) can also be used on the console line.

Viewing Variables

Hovering over a variable name in the source view shows the current value of that variable in a tooltip. Note that the tooltip only appears when the target is in a stopped state.

The “Watches” view in the right sidebar shows the current value of all expressions that have been added to it. The expression can be as simple as the name of a variable, but it may include redirections or arithmetic operations. When adding a watch, all variables that are mentioned in the expression are evaluated in the active scope. The expression of the watch retains this scope. When stepping into a sub-routine or function, the Watches view keeps showing the watches in the scope that the watch was declared in.

A watch can be added by typing the expression in the edit field in the Watches view and clicking on the  button. You can also use the `display` command in the console line (see section [Examining Variables and Memory](#) on page 24). The `bmdebug` front-end handles the `display` and `undisplay` commands internally.

Trace Views

Two trace views are provided: one for semihosting and one for SWO tracing. See chapter [Run-Time Tracing](#) on page 33 for more information on tracing.

The view for semihosting is always active and it requires no configuration, except that the target firmware must be built to send output via the semihosting interface.

The SWO tracing view must be configured through commands on the console line. These commands are specific to the Black Magic Probe and the bmdebug front-end; they are not passed on the GDB.

<code>trace clock bitrate</code>	Enable tracing in Manchester encoding with the given clock of the target micro-controller and bit rate. Note that while the Black Magic Probe does not need the clock and bit rates, the bmdebug front-end requires these to configure the target. The clock and bitrate parameter may have a MHz or kHz suffix. For example, the clock may be specified as either 12mhz or 12000000.
<code>trace async clock bitrate</code>	Enable tracing in Asynchronous encoding with the given clock of the target micro-controller and bit rate. The clock and bitrate parameters are the same as with the preceding command.
<code>trace passive</code> <code>trace pasv</code>	Enable tracing in Manchester encoding, but without configuring the clock and bitrate in the target micro-controller. The firmware of the target must itself configure SWO tracing.
<code>trace disable</code>	Disables SWO tracing.
<code>trace channel index enable</code> <code>trace chan index enable</code> <code>trace ch index enable</code>	Enables display of the given channel (range 0..31).
<code>trace channel index disable</code> <code>trace chan index disable</code> <code>trace ch index disable</code>	Disables display of the given channel.
<code>trace channel index name</code> <code>trace chan index name</code> <code>trace ch index name</code>	Set a name for the channel marker in the view (the default name is the channel number). Note that when using the Common Trace Format, the channel names are initially set to the “stream” names in the trace metadata.

trace channel <i>index</i> # <i>colour</i>	Set the background colour of the channel marker. The colour must be in "HTML format" with three pairs of hexadecimal digits following the "#", in the order R/G/B.
trace chan <i>index</i> # <i>colour</i>	
trace ch <i>index</i> # <i>colour</i>	
trace info	Show the current configuration and all active channels.

On start-up, the bmdebug front-end searches for a metadata file for the [Common Trace Format](#) (see page 43). When this file is available, it switches to this format for decoding the trace messages. The bmdebug front-end looks for a file with the same base name as the ELF file and a ".tsdl" extension, and it searches in the same directory as the ELF file, as well as in the directories where the source files are.

Run-Time Tracing

The standard “stop & stare” style of debugging, where you step through code one line at a time after hitting breakpoint, may not be suitable for an embedded system. When the code hits a breakpoint, the micro-controller stops, and this may be *too little* or *too much* (and even both at the same time). The micro-controller may not run in isolation: if it drives a linear actuator, that actuator will continue to run while the MCU is in stopped state, until it reaches a safety end stop — unless that end stop is handled by an interrupt routine on the same MCU, in which case the actuator will run until it damages itself. Stopping the micro-controller does too little in this case: it does not stop the linear actuator, but it also does too much: it no longer responds to the signal of the safety end stop.

The alternative debugging technique for such circumstances is run-time tracing. The goal of tracing is to be non-intrusive: it gives you insight in what the code does *without* interfering with it. Run-time tracing is similar to logging, the differences between the two are mostly due to their distinctive purposes (logging is used by system administrators to review activity of the system; tracing is used by developers to spot software faults). Run-time tracing is also akin to post-mortem analysis in the sense that you are analysing the code flow (and the logic behind that code flow) after the fact.

This chapter starts with an overview of the various methods for tracing that the Black Magic Probe offers. Each of these has its own advantages and disadvantages. In the second part, it delves into an efficient binary format and protocol for run-time tracing.

Secondary UART

The Black Magic Probe combines the gdbserver interface with a TTL UART interface (on the same USB connection). If the target board has the TxD and RxD lines of a UART branched out of the micro-controller, and the target does not need the UART for other purposes, you can use that port to output trace messages and capture those on a general purpose serial terminal.

Sending trace messages over a UART is a boiler plate technique, because it works everywhere: all micro-controllers offer one or more UART peripherals and (virtual) serial ports on workstations are commonplace too. Other than its ubiquity, a bene-

fit of the UART is that it only a *single* pin —configuring Rx/D is superfluous for tracing purposes. Of course, this is only valid in the case that you use tracing as your *only* means of debugging; otherwise, the UART pins are *in addition to* the pins reserved for the JTAG or SWD interface.

The RS232 transmission rates are, for today's standards, rather slow. Therefore, there is the risk that tracing slows down the code flow too much, defeating the entire purpose of run-time tracing.

The calls to a transmit (or “print”) function must also be added to the source code, a process called code instrumentation. These instrumented calls make the code bigger and run slower. Unless you also build a method to disable tracing dynamically in the production code (the code that you distribute), you will want to remove the trace instrumentation from the production build. It is therefore common that the code instrumentation is implemented with conditionally compiled macros.

Semihosting

Semihosting uses the debug protocol and interface, so that it does not require extra pins if you already have the JTAG or SWD pins branched out. This is especially convenient if you are using an ST-Link clone instead of the original Black Magic Probe hardware, because the ST-Link clones have neither a secondary UART for tracing, nor the TRACESW0 pin branched out (see page 37 for SWO tracing).

On the other hand, due to additional overhead by the debug probe, semihosting has lower performance than using a UART. Semihosting also requires support from the debug probe and the debugger running on the remote host, but both the Black Magic Probe and GDB provide the necessary support. The source code must furthermore be instrumented with calls to trace, printf or similar.

At a low level, semihosting works by inserting a software breakpoint (or sometimes a software exception) in the code, followed by a special token value. When the micro-controller reaches that instruction, it halts and signals the debug probe. The debug probe first looks at the address of the break instruction, sees the token, and enters semihosting state. It then analyses two registers, r0 and r1, which carry a command code and a pointer to a parameter block. The debug probe forwards the commands to the debugger (GDB in our case), which runs it and may transmit results back.

The ARM semihosting protocol is extensive and flexible. In principle, it allows the embedded target to relegate console and file I/O to the host. For tracing, only a single command code is relevant (SYS_WRITE). The snippet below is a function for transmitting a trace message using semihosting, implemented in GCC.

```
void trace(const char *message)
{
    uint32_t command = 5;    /*SYS_WRITE*/
    uint32_t packet[3] = { 2 /*stderr*/, (uint32_t)message, strlen(message) };
    __asm__ (
        "mov r0, %0\n"
        "mov r1, %1\n"
        "bkpt #0xAB\n"
        :
        : "r" (command), "r" (packet)
        : "r0", "r1", "memory"
    );
}
```

The command code 5 is defined for writing to a file, and file handle 2 (the first word in the packet array) is the predefined handle for “standard error” console output. When calling `trace(“Hello world”)` from your code (and running it from GDB), this text will be printed on the GDB console.

The reason for writing to file handle 2 (stderr) instead of handle 1 (stdout) is that when you use GDB without a front-end, `stderr` can be redirected to a file or separate terminal (instead of being mixed with GDB console output). Note however, that GDB prints error messages to `stderr` as well, so GDB output and trace messages can still wind up interwoven. A front-end may write semihosting output to a separate view or window (regardless of whether it is sent to `stderr` or `stdout`), however in this case, output from the Black Magic Probe itself may also wind up in that view. The `bmdebug` front-end shows semihosting output in the “Target output” view, see page 27).

The above snippet is for the ARMv6-M and the ARMv7-M architectures (ARM Cortex M0, M0+ M1, M3, M4 and M7 series). On other architectures, you may need the `SVC` instruction rather than `BKPT`.

Depending on the standard libraries that you use, you may not need to implement a trace function yourself, but simply use `printf()` via semihosting. In particular, the library `librdimon` (part of `newlib`) implements semihosting calls. If you use `newlib`, it is sufficient to add the following option to the linker command line:

```
--specs=rdimon.specs
```

A drawback of semihosting is that the target requires to see a debugger attached in order to run. If the debugger is not present, and the code sends a trace message, it drops into a software breakpoint —and triggers a *HardFault* exception. Trace calls via semihosting are therefore typically wrapped inside macros whose definition is conditional on the build: debug versus release.

An alternative is to determine at run-time whether a debugger is attached and adjust the `trace()` function to return straight away if otherwise. On a Cortex M3/M4/M7 micro-controller, this is as easy as testing the lowest bit of the *Debug Halting Control & Status Register* (DHCSR):

```
if (CoreDebug->DHCSR & 1) {  
    /* debugger attached */  
} else {  
    /* not running under a debugger */  
}
```

On the Cortex M0 micro-controller architecture, however, the CoreDebug registers are only accessible from the JTAG/SWD interface, not from the code that runs on the micro-controller. Instead, you can implement a *HardFault* handler to check the cause of the exception and return to the caller if it turns out to be a semihosting call. This way, the `trace()` function still drops on the BKPT instruction and still causes a *HardFault* exception (in absence of a debugger), but the *HardFault* handler ignores it and moves the program counter to the instruction behind it.

```
__attribute__((naked))  
void HardFault_Handler(void)  
{  
    __asm__ (  
        "mov    r0, #4\n"           /* check bit 2 in LR */  
        "mov    r1, lr\n"  
        "tst    r0, r1\n"  
        "beq    msp_stack\n"       /* load either MSP or PSP in r0 */  
        "mrs    r0, PSP\n"  
        "b      get_fault\n"  
        "msp_stack:\n"  
        "mrs    r0, MSP\n"  
        "get_fault:\n"  
        "ldr    r1, [r0,#24]\n"   /* read program counter from the stack */  
        "ldrh   r2, [r1]\n"       /* read the instruction that caused the fault */  
        "ldr    r3, =0xbeab\n"     /* test for BKPT 0xAB (or 0xBEAB) */  
        "cmp    r2, r3\n"  
        "beq    ignore\n"       /* BKPT 0xAB found, ignore */  
    )  
}
```

```

    "b    .\n"                /* other reason for HardFault, infinite loop */
    "ignore:\n"
    "add  r1, #2\n"           /* skip behind BKPT 0xAB */
    "str  r1, [r0,#24]\n"    /* store this value on the stack */
    "bx   lr"
);
}

```

The way the HardFault handler works is slightly convoluted, because the ARM Cortex micro-controller has two stack pointers, for the “main stack” and the “process stack”. When the exception occurred, the micro-controller has pushed a set of registers on the stack, including the program counter, but the first thing the HardFault handler must do is to check *which* stack. Once it has the appropriate stack pointer, by testing bit 2 in the LR register, it gets the value of the program counter. The program counter is the address of the instruction that caused the exception, so the handler reads from that address and tests for opcode 0xBE with parameter 0xAB. On a match, it is a semihosting breakpoint and it increments the program counter value on the stack before returning; effectively returning to the instruction that follows the breakpoint. Otherwise, it drops into an infinite loop, just like the default implementation for the HardFault handler.

The HardFault handler approach for run-time debugger detection works on all Cortex architectures, by the way.

SWO Tracing

The ARM Cortex M3, M4, M7 and A architectures provide a separate pin for tracing system and application events at a high data rate. This is the TRACESWO pin on the Cortex Debug header (see page 12). The ARM Cortex M0 lacks support for SWO tracing.

The SWO Trace protocol allows messages to be transmitted on 32 channels (or *stimulus ports*, per the ARM documentation). This allows you to separate output for different modules in the firmware or to implement different levels of trace detail, because each channel can be individually enabled or disabled. Sending a trace message on a channel that is disabled takes negligible time, and therefore it may be an option to leave the trace calls in the production code.

With CMSIS, a typical implementation of a `trace()` function is as below. Note, however, that the CMSIS function `ITM_SendChar()` is hard-coded to use channel 0.

```
void trace(const char *msg)
```

```

{
    while (*msg != '\0')
        ITM_SendChar(*msg++);
}

```

A more flexible function is below. It starts by checking whether tracing is enabled, both globally and on the chosen channel, so that it doesn't even run through the message string if nothing would be output anyway. Before storing every next character in the queue for the trace subsystem, it checks whether the queue has space left to hold the character.

```

void trace(int channel, const char *msg)
{
    if ((ITM->TCR & ITM_TCR_ITMENA) != 0UL && /* ITM tracing enabled */
        (ITM->TER & (1 << channel)) != 0UL) /* ITM channel enabled */
    {
        while (*msg != '\0') {
            while (ITM->PORT[channel].u32 == 0UL)
                __NOP();
            ITM->PORT[channel].u8 = (uint8_t)*msg++;
        }
    }
}

```

SWO Tracing must first be configured in the micro-controller. This configuration can be done either in the firmware code on the micro-controller, or by the debugger or trace viewer. Joseph Yiu, author of *The Definitive Guide to ARM Cortex-M3 Processors*, considers that configuration should be done by the debugging tool, as to avoid that the firmware and the debugging tool overwrite each-other's settings. On the other hand, some micro-controllers require additional configuration that is specific to that device and not standardized by ARM. Configuring the tracing in code (at least partially) is a viable option.

The Orbuculum project allows both approaches. The trace capture tools of this project do not perform any configuration, but the project comes with .gdbinit files with settings and definitions to perform the configuration from within GDB. The Orbuculum trace tools do not require GDB in itself, but even if you perform the trace configuration in code, you still need GDB to enable the trace option on the Black Magic Probe.

The command to enable tracing in the Black Magic Probe is:

```
monitor traceswo
```

The SWO Trace protocol uses one of two serial formats: asynchronous encoding and Manchester encoding. The ARM documentation occasionally refers to these encodings as NRZ and RZ (Non-Return-to-Zero and Return-to-Zero). The official firmware of the Black Magic Probe supports only Manchester encoding. A property of Manchester encoding is that the clock speed can be determined from the data stream, so the bit rate does not need to be specified on the `traceswo` command. However, the Black Magic Probe lacks a hardware decoder for the Manchester bit stream, and therefore (since it handles the decoding in software) the supported bit rates are limited to roughly 100 kb/s. Recent development builds also support asynchronous encoding, which allows bit rates of up to 2.25 Mb/s, but in this case, the bit rate must be set on the `traceswo` command.

```
monitor traceswo 2250000
```

The target must be able to configure the same bit rate, within an error margin of 3%. The bit rate must furthermore be 4.5MHz divided by an integer value and therefore there is a limited choice at the high end of the bit rates: 2.25 Mb/s, 1.5 Mb/s, 1.125 Mb/s, 900 kb/s, 750 kb/s, 642.9 kbps, 562.5 kb/s, etc.

The initialization that is generic for all ARM Cortex micro-controllers is below. It involves a number of sub-components of the CoreSight architecture, notably the *Instrumentation Trace Macrocell* (ITM) and the *Trace Port Interface Unit* (TPIU, also called TPI), but registers in the Core Debug and *Data Watchpoint & Trace* (DWT) modules may come into play as well.

```
void trace_init(uint32_t bitrate, uint32_t channelmask)
{
    CoreDebug->DEMCR = CoreDebug_DEMCR_TRCENA_Msk;

    TPI->CSPSR = 1;          /* protocol width = 1 bit */
    TPI->SPPR = 1;           /* 1 = Manchester, 2 = Asynchronous */
    TPI->ACPR = (CPU_CLOCK_FREQ / (2 * bitrate)) - 1;
    TPI->FFCR = 0;           /* turn off formatter, discard ETM output */

    ITM->LAR = 0xC5ACCE55; /* unlock access to ITM registers */
    ITM->TCR = ITM_TCR_SWOENA_Msk | ITM_TCR_ITMENA_Msk;
    ITM->TPR = 0;           /* privileged access is off */
    ITM->TER = channelmask; /* enable stimulus channel(s) */
}
```

For Manchester encoding, the clock frequency must be set to twice the bit rate, because there are transitions halfway the bit period for “1” bits in the signal.

An extra device-specific initialization step often needs to precede the generic initialization. A few sample snippets are below. Note that some micro-controller series do not need any device-specific initialization (for example, the LPC175x and LPC176x series).

STM32Fxx series

```
void trace_init_STM32Fxx(void)
{
    RCC->APB2ENR |= RCC_APB2ENR_AFIOEN; /* enable AFIO access */
    AFIO->MAPR |= AFIO_MAPR_SWJ_CFG_1; /* disable JTAG to release TRACESWO */
    DBGMCU->CR |= DBGMCU_CR_TRACE_IOEN; /* enable IO trace pins */
}
```

LPC13xx series

```
void trace_init_LPC13xx(void)
{
    LPC_SYSCTL->TRACECLKDIV = 1;
    LPC_IOCON->PIO0_9 = 0x93;
}
```

LPC15xx series

```
void trace_init_LPC15xx(int pin)
{
    LPC_SYSCTL->TRACECLKDIV = 1;
    LPC_SWM->PINASSIGN15 = (LPC_SWM->PINASSIGN15 & ~(0xff << 8)) | (pin << 8);
}
```

LPC5410x series

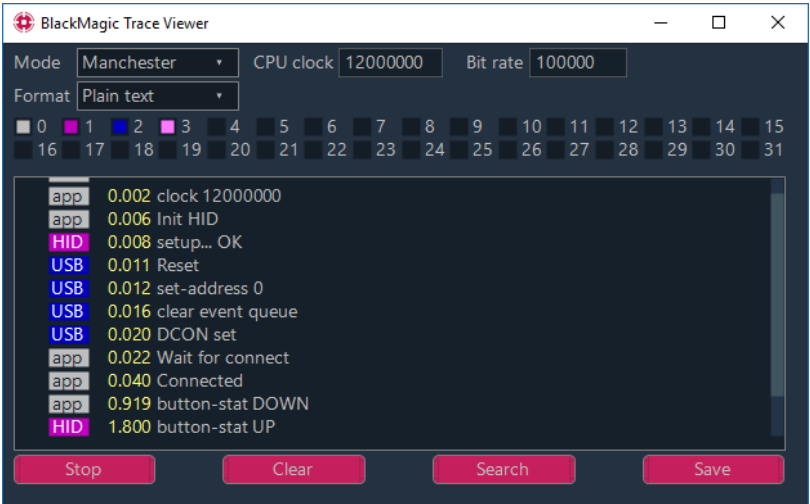
```
void trace_init_LPC15xx(void)
{
    LPC_SYSCTL->TRACECLKDIV = 1;
    LPC_SYSCTL->SYSAHBCLKCTRLSET = 1 << 13;
    LPC_IOCON->PIO0_15 = 0x82;
}
```

For capturing the trace data, the Orbuculum project was already mentioned. This is a set of console utilities for Linux and macOS, of which the main component creates output files in CSV format, for further analysis (or plotting) in a spreadsheet or other application.

A stand-alone graphical trace viewer for SWO tracing using the Black Magic Probe is *bmttrace*: the *BlackMagic Trace Viewer* (a companion tool to this guide). It runs under Microsoft Windows and Linux. The *bmttrace* utility does not require GDB, because it uses the *Remote Serial Protocol* (RSP) to configure the target and the Black Magic Probe. The *bmttrace* utility performs the generic configuration for SWO

tracing as well as the device-specific configuration for the micro-controllers that it supports. Another distinctive feature of `bmtrace` is that it supports the [Common Trace Format](#), see page 43.

To perform trace capture alongside a debugging session, `bmtrace` can be set to run as a “passive listener”. In passive mode, it only captures trace messages, but does not interact with the target or the Black Magic Probe. The other two available modes are “Manchester” and “Async.”: the two serial formats for SWO tracing.



Any of the 32 channels can be enabled or disabled. A right-click on the channel selector pops up a window to set a colour and a name for the channel. Note that when running in passive mode, any disabled channels are simply hidden in the trace viewer; they are *not* disabled in the target (because `bmtrace` does not communicate with the Black Magic Probe in passive mode). When running in CTF mode ([Common Trace Format](#), see page 43), the names of the channels are overruled by the “stream” names that are defined in the metadata file for the traces.

The time stamps in the `bmtrace` utility are relative to the first message that was received. With one exception, these time stamps are of the moment of *reception* of the trace data. Due to latencies of the USB stack and jitter in the scheduling of the operating system, these time stamps are indicative, but not conclusive. The exception is that `bmtrace` shows the timestamps in the Common Trace Format stream, if these are present. These timestamps are generated on the target, and they are generally more accurate.

Tracing with Command List on Breakpoints

Breakpoints were briefly covered in section [Breakpoints and watchpoints](#) (page 22). A feature of GDB is that a list of commands may be attached to a breakpoint, and this list is executed whenever the breakpoint is hit. The trick is: when the final command in this list is “continue”, you have created a breakpoint that breaks only very briefly. However, information that the breakpoint was hit is output on the GDB console, and you may add commands to print the values of variables in the command list for the breakpoint.

As a simple example, consider a command list that only contains the continue command:

```
(gdb) break 121
Breakpoint 3 at 0x3ce: file blinky.c, line 121.
(gdb) command 3
Type commands for breakpoint(s) 3, one per line.
End with a line saying just "end".
>continue
>end
```

When running the code, GDB will print lines similar to the following, each time that the breakpoint is hit:

```
Breakpoint 3, main () at blinky.c:121
121          LPC_GPIO->SET[led_ioport] = led_iobit; /* turn LED on */
```

While this only shows that the line was reached, the important difference with the alternative trace methods is that the code does not need to be instrumented with trace calls. This implies that no recompilation is necessary if you want to move or add a trace-point. This method of tracing is therefore convenient if you want to check whether a particular line is reached. A limitation of this technique is that there is only a small pool of hardware breakpoints (which are needed when running from Flash).

Any GDB command can be inserted before the continue command. For example print to show the variable values.

The Common Trace Format

As explained in the chapter on [Run-Time Tracing](#) (page 33), the intention of run-time tracing is to be a non-intrusive method of debugging. This implies that the trace messages should have negligible overhead, in time and other resources. If the overhead is non-negligible, the software may behave differently when being traced, than when running without tracing: a symptom that is called the *probe effect*.¹

When we focus on the time, the factors that contribute to “overhead” (delays) are:

- The need to format the data into a trace message on the micro-controller prior to transmitting it.
- The amount of data to transfer, either to a remote “trace viewer” or internally to a display system.
- The speed of the data transfer channel and any I/O overhead in accessing it.

When it comes to avoiding the probe effect, there is a general fixation on the last point, the speed of the transfer interface. Perhaps as a corollary to the *Law of the Hammer*,² the reflex is to search for a bigger hammer if issues resurface. Yet, it is obvious that no matter how well you’ve optimized `sprintf`, skipping it will always be quicker; like it is obvious that transmitting few bytes is quicker than transmitting many (under equal conditions).

This brings us to the Common Trace Format (CTF), by the Diagnostic and Monitoring workgroup (DiaMon) of the Linux Foundation. The Common Trace Format is a specification for a binary data format plus a human-readable “metadata file” to map the binary data to readable text. It thus does away with the formatting and conversion on the micro-controller and it also skips transferring text strings if it can instead reference these strings in the metadata file.

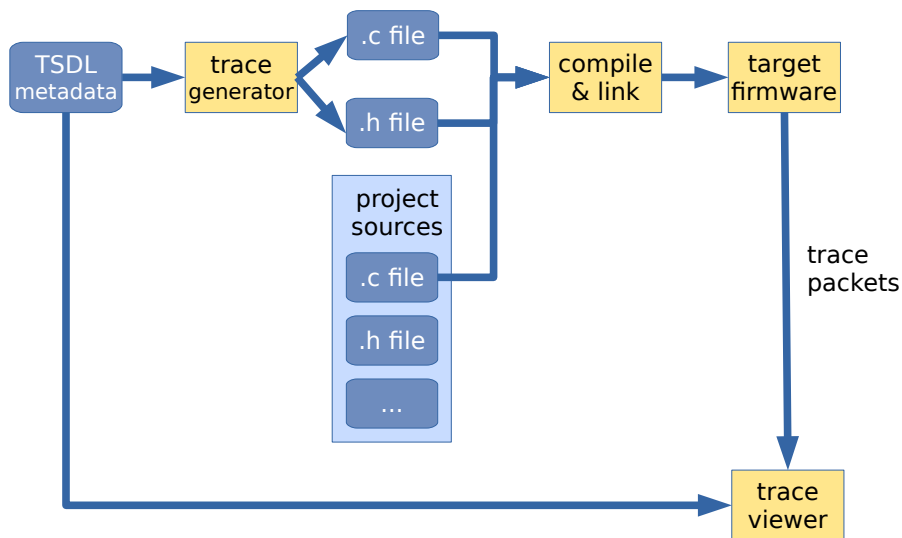
The metadata file defines the names of trace “events”, the streams that these events belong to and the names and types of any parameters of each event. This is all recorded in a declarative language with a C-like syntax: the *Trace Stream Description Language* (TSDL). The metadata is shared (directly or indirectly) between the target that produces the trace messages and the trace viewer. The Common Trace Format achieves its compactness because the data in this metadata file is never transmitted.

1 J. Gait; *A probe effect in concurrent programs*; Software: Practice and Experience; March 1986.

2 "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." [Abraham Maslow; *The Psychology of Science*; 1966]

The Common Trace Format is the cornerstone of LTTng (Linux Trace Toolkit next generation); however, a call into LTTng is not exactly low-overhead in execution time (the rationale for LTTng’s use of CTF is to minimize storage requirements). Besides, it is not an option for embedded systems that run on something other than the full Linux kernel.

Two tools exist that generate OS-independent C code for CTF support: `barectf` by the same authors as CTF, and `tracegen` (which is a companion tool to this guide). Both tools use the metadata to generate individual C functions to build a binary CTF “packet” for each particular trace event. The generated file is then included in the build for the target firmware, and the source code can call the generated functions to transmit a trace packet in the compact CTF format. The `barectf` tool replaced TSDL with YAML as the metadata language (and it generates a TSDL file for the trace viewer), while the `tracegen` tool sticks with TSDL, but adds some extensions to make it more convenient.

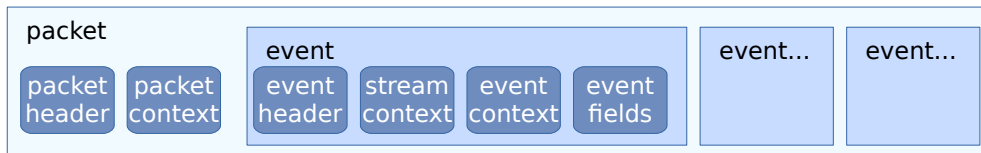


In the above flow chart, the “trace generator” would be `barectf` or `tracegen`, and the “trace viewer” is either `bmtrace` (the BlackMagic Trace Viewer, see page 40) or another CTF compatible viewer, like Trace Compass. In fact, when using `barectf`, the flow is slightly different: the input to `barectf` is a YAML file and it creates a TSDL file (along C source and header files) for the trace viewer.

Binary Packet Format

The Common Trace Format sends trace messages in packets. A packet holds one or more events. An event is basically a single trace message. In practice, packing multiple events in a packet is only useful if the transport protocol imposes a fixed or

minimum size on packets. For stream-based protocols like RS232 or SWO (which this guide focusses on), a packet holds a single event.



The packet header is optional; it contains a magic value to flag the binary data as the start of a CTF packet and the stream identifier. More information about the packet, such as its size and encoding, may follow in the (equally optional) packet context block.

For each event, an event header is required, because it contains the event identifier (plus possibly a timestamp for the event). The “event fields” block, at the tail of the event, holds any additional parameters that the event has. For example, if you trace a temperature sensor, the event name could be “temperature” and the single field the value in degrees Celsius or Fahrenheit (or Kelvin, for that matter). The “stream context” and “event context” blocks, are usually not relevant for embedded systems. The stream context holds data that applies to all events in the stream, whereas the event context has data that is specific to the event (but cannot be represented in the event fields).

Which of the optional headers you should include in the packet depends in part on the transfer protocol. If it is packet-based, like USB or Ethernet, you may choose to omit the packet header, but instead include a packet context with the size of that packet. If, on the other hand, it is a byte stream, like RS232 or SWO, the packet header is as good as mandatory, while the package context is of little use.

A Synopsis of TSDL

The *Trace Stream Description Language* uses a syntax inspired by the C typing system. It will therefore be familiar to most embedded systems developers.

A minimal example for a specification of a single event is below. It defines an event called “peltier-plate”, with a single field called “voltage” of type “unsigned char”.

```
event {
    name = "peltier-plate";
    fields := struct {
        unsigned char voltage;
    };
};
```

Neither a packet header nor an event header are defined; therefore these will not be present in the byte stream. Since the size of the single field is a byte, when the byte stream is:

```
18 1A 1B
```

it will be translated by the trace viewer to the following three events:

```
peltier-plate: voltage = 24  
peltier-plate: voltage = 26  
peltier-plate: voltage = 27
```

Merely a single byte needs to be transmitted for a descriptive parametrized event, it does not get much more compact than that. However, this is an exceptional case. When there is more than one event, an event header is needed so that the various events can be distinguished. This leads to the need for a packet header as well: to determine the function of each byte in a byte stream, one must know its position in the packet definition, and therefore one must know where the packet starts in the byte stream.

The following snippet addresses those issues. It defines a packet header in the trace section and an event header in the stream section. A second event is added too.

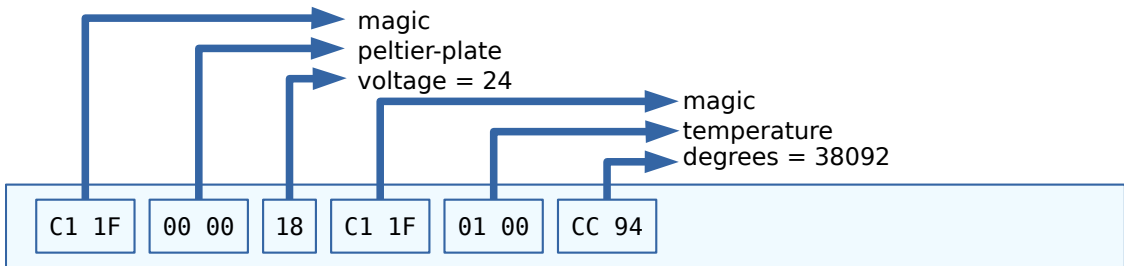
```
trace {  
    major = 1;  
    minor = 8;  
    packet.header := struct {  
        uint16_t magic;  
    };  
};  
  
stream {  
    event.header := struct {  
        uint16_t id;  
    };  
};  
  
event {  
    id = 0;  
    name = "peltier-plate";  
    fields := struct {  
        unsigned char voltage;  
    };  
};
```

```

event {
    id = 1;
    name = "temperature";
    fields := struct {
        uint16_t degrees;
    };
};

```

An example of a byte stream that matches the above trace description is:



The trace viewer would display the two trace messages:

```

peltier-plate: voltage = 24
temperature: degrees = 38092

```

In `tracegen`, types like `uint16_t` (as used in the above example) are predefined. When using Babeltrace or another system, you may need to define these types yourself. This can be done with `typedef`, in the same way as in C, or with the more comprehensive `typealias`. The `typealias` construct allows you to set the size of the variable unambiguously, as well as any scaling (“fixed-point” representation), and in which base the number must be displayed (decimal, hexadecimal, binary). The snippet below shows the changes to the “temperature” event.

```

typealias integer {
    size = 16;
    scale = 1024;
    signed = false;
} := fixed_point;

event {
    id = 1;
    name = "temperature";
    fields := struct {
        fixed_point degrees;
    };
};

```


When the event for the temperature sensor is changed to a scaled integer (6 bits integer part, 10 bits fractional part, or a scaling factor of 2^{10}), the trace viewer would display the following on the byte stream C1 1F 01 00 CC 94:

```
temperature: degrees = 37.199
```

Packet header

The packet header may contain the following fields (in any order):

magic	A 1-, 2-, or 4-byte integer, whose purpose is to mark the start of a packet in a stream of bytes. A longer magic value gives a more reliable detection of the start of a packet, at the cost of more bytes being transmitted. A 2-byte integer is a common compromise.
uuid	A user-supplied identifier, used to make sure that the byte stream of the traces matches the definitions in the metadata (the TSDL file). Due to its heavy cost in overhead (16 bytes added to every packet), its use is not recommended for embedded systems.
stream.id	A 1-, 2-, or 4-byte integer with the stream number. Redundant if the trace information uses only a single stream; also redundant for SWO tracing when less than 32 streams are used (because the stream ID is mapped to the SWO channel). This field may also be called “stream_id” for compatibility with other CTF implementations.

Event header

event.id	A 1-, 2-, or 4-byte integer with the event ID. This field may also be called “id” for compatibility with other CTF implementations.
timestamp	A 4-byte or 8-byte timestamp for the event. The timestamp is linked to the definition of a <i>clock</i> in the TSDL file (see notes below).

Timestamps must be linked to a clock. This takes two parts: the definition of a clock and the definition of a type that references this clock. The timestamp is then defined as that type.

```
clock {
    name = cycle_counter;
    freq = 1000000000;          /* frequency, in Hz */
};

typealias integer {
    size = 64;
    signed = false;
    map = clock.cycle_counter;
} := tickcount_t;
```

```

stream {
    event.header := struct {
        uint16_t event.id;
        tickcount_t timestamp;
    };
};

```

There are more fields in the clock specification, specifically for synchronizing various clocks in a heterogeneous tracing environment, but these are skipped here. The new type `tickcount_t` maps to this clock, and the `timestamp` field in the event header is defined as a `tickcount_t` type. Following the chain backward, the `timestamp` field is now linked to the clock “`cycle_counter`”.

Instead of having the target transmit the timestamps of every event, we recommend that a trace viewer displays the timestamp of when the trace packets are received (and that the timestamp is omitted from the event header). The timestamp of the reception is less accurate (due to latencies and jitter in the transmission protocol), but accuracy in the timestamps is usually only required for specific events: in those events, the timestamp can be transmitted as a parameter (an “event field”).

Scaling up: multiple streams, many events

When there are many trace events or multiple streams involved, a few shorthand notations exist to make maintenance of the metadata easier. When there are multiple streams, each stream should have a unique ID and each event (which should also have a unique ID) must indicate which stream it belongs to.

The `tracegen` utility extends TSDL by allowing a stream to have a name, so that an event can identify its stream by its name rather than a numeric constant. It also supports automatic numbering of streams and events (`barectf` also supports auto-numbering). For brevity in the TSDL file, the names of a stream and of an event can be placed immediately following the `stream` or `event` keywords. In the case of an event, it specifies the stream name and its own name in combination.

Below is the example from page 46 with the shorthand notations.

```

trace {
    version = 1.8;
    packet.header := struct {
        uint16_t magic;
        uint8_t stream.id;      /* redundant with SW0 */
    };
};

```

```

    };
};

typedef integer {
    size = 16;
    scale = 1024;
    signed = false;
} := fixed_point;

stream cooler {
    event.header := struct {
        uint16_t id;
    };
};

event cooler::"peltier-plate" {
    fields := struct {
        unsigned char voltage;
    };
};

event cooler::temperature {
    fields := struct {
        fixed_point degrees;
    };
};
};

```

This snippet defines a stream “cooler” and the events “peltier-plate” and “temperature”, both linked to stream “cooler”. The name “peltier-plate” is between quotation marks, because it contains a “-” character. You may enclose all identifiers in quotation marks, but it is not needed if a name only contains letters, digits and “_” characters (like C identifiers).

Since there is only a single stream in this example, giving the stream a name and referencing its name explicitly in the events is actually redundant. The stream could equally well be anonymous and the “cooler::” prefix could then be omitted from the event specifications.

When there is a single stream, the `stream.id` in the `packet.header` is usually redundant. With SWO tracing, it is also redundant in the case of multiple streams, because the stream ID is mapped to the SWO channel. The ID therefore does not have to be repeated in the packet header. Note that you are limited to 32 streams in this case.

Note that these shorthand notations are specific to the `tracegen` and `bmtrace` utilities. When using a different trace viewer, the basic TSDL syntax (as specified on the site of the DiaMon workgroup) should be used.

Generating Trace Support Files

When running the `tracegen` utility on the metadata file, it generates a C source and a C header file. These files contain the definitions and the implementations of functions, and each of these functions creates and transmits a packet for an event. For example, when the snippet on page 49 is a file with the name “`peltier.tsd`”, you can run the following command:

```
tracegen -s peltier.tsd
```

The output is two files, with the names `trace_peltier.c` and `trace_peltier.h`. These contain the functions:

```
void trace_cooler_peltier_plate(unsigned char voltage);  
void trace_cooler_temperature(fixed_point degrees);
```

The function names contain both the name of the stream and the names of the events. If the stream were anonymous, that part would not be present in the function names either. Any characters that are not valid for use in C identifiers are replaced by an underscore. This happened with the event name “`peltier-plate`” for example: the C identifier replaces the “`-`” by a “`_`”.

The “`-s`” option to `tracegen` makes it generate code for SWO tracing. When you would use the Common Trace Format for tracing over an RS232 line, this option is not needed.

Also note how the types of the function arguments are copied from the metadata file into the C functions. Your source code should define a `fixed_point` type that matches the definition in the metadata. The alternative is to use the “`-t`” option on `tracegen`, in which case it will always attempt to translate the type in the metadata file to a basic C type.

```
tracegen -s -t peltier.tsd
```

The above call would generate the following function prototype for the `temperature` event:

```
void trace_cooler_temperature(unsigned short degrees);
```

The function prototypes and implementations in the source and header files are wrapped in conditional compiled sections that test for the `NTRACE` macro. If the

NTRACE macro is defined, the functions are disabled. Thus, if you need to build a release version of the firmware without any tracing functions, rebuild all code with a definition of NTRACE on the compiler command line.

Integrating Tracing in your Source Code

The tracegen utility generates prototypes and implementations for transmitting trace events, as was shown in the previous section. When integrating this code in your project, one or two additional functions need to be provided by your code.

```
void trace_xmit(int stream_id, const unsigned char *data, unsigned size);
unsigned long long trace_timestamp(void);
```

The task of the trace_xmit function is to truly transmit the data over a kind of port or interface. For SWO tracing, this would be an adaption of the trace function on page 38:

```
void trace_xmit(int stream_id, const unsigned char *data, unsigned size)
{
    if ((ITM->TCR & ITM_TCR_ITMENA) != 0UL && /* ITM tracing enabled */
        (ITM->TER & (1 << stream_id)) != 0UL) /* ITM channel enabled */
    {
        while (size-- > 0) {
            while (ITM->PORT[stream_id].u32 == 0UL)
                __NOP();
            ITM->PORT[stream_id].u8 = (uint8_t)*data++;
        }
    }
}
```

The above examples assume that you have run tracegen with the “-s” option on the TSDL file. Without the “-s” option, the definition of trace_xmit lacks the stream_id parameter (the stream ID would instead be present in the packet header).

The trace_timestamp function returns a timestamp, which is then transmitted as part of the event header. The return type of this function depends on the declaration of the clock in the TSDL file, see page 48. If the event header does not include a timestamp, there is no need to implement this function (as it will not be called).

Firmware Programming

As show in chapter [Debugging Code](#) on page 17, GDB downloads the code in the micro-controller as part of the debugging process. This opens the way for using the Black Magic Probe for small-scale production programming as well.

Using GBD

You can use GDB for uploading code to Flash memory by setting commands on the command line. The following snippet is a single command broken over multiple lines, for the Microsoft Windows command prompt (in Linux, replace the “^” symbol at the end of each line by a “\”). In practice, you would put it in a batch file or a bash script.

```
arm-none-eabi-gdb -nx --batch ^
-ex 'target extended-remote COM9' ^
-ex 'monitor swdp_scan' ^
-ex 'attach 1' ^
-ex 'load' ^
-ex 'compare-sections' ^
-ex 'kill' ^
blinky.elf
```

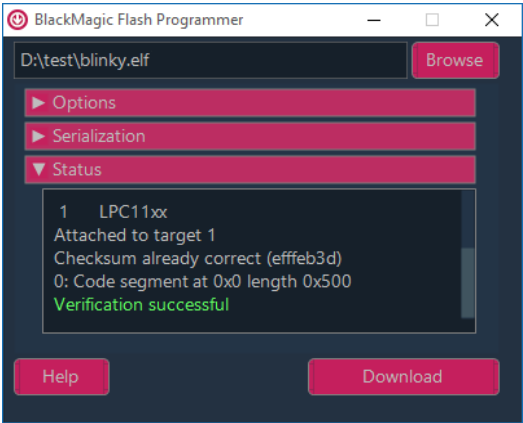
You need to change COM9 to the serial device that is appropriate for your system, and blinky.elf to the appropriate filename. In Linux, you may use the bmscan utility to automatically fill in the device name for the gdbserver virtual serial port:

```
arm-none-eabi-gdb -nx --batch \
-ex 'target extended-remote `bmscan gdbserver`' \
-ex 'monitor swdp_scan' \
-ex 'attach 1' \
-ex 'load' \
-ex 'compare-sections' \
-ex 'kill' \
blinky.elf
```

Also see the note on the LPC micro-controller series from NXP regarding the compare-sections command on page 20.

Using the BlackMagic Flash Programmer

The `bmflash` utility is a GUI utility that offers a few additional features over GDB for firmware programming. The `bmflash` utility uses the *Remote Serial Protocol* (RSP) of GDB to directly communicate with the Black Magic Probe. GDB is therefore not required to be installed on the workstation on which you perform production programming.



The `bmflash` utility automatically scans for the Black Magic Probe on start-up, and connects to it. It also has built-in handling of the idiosyncrasies of the LPC microcontroller series from NXP (see page 20).

Furthermore, `bmflash` supports serialization, in which the utility stores a serial number in the Flash memory of the target, and increments that serial number for each successful download.

The modes that are available for serialization are:

No serialization	No serialization is performed.
Address	The options for this mode are the name of a section in the ELF file, and the offset in bytes from that section. The offset is a hexadecimal value. The section name is typically <code>".text"</code> or <code>".rodata"</code> . If the section name is empty, the offset is from the beginning of the ELF file.
Match	<p>In this mode, the <code>bmflash</code> utility searches for a signature or byte pattern in the original ELF file, and stores the serial number at a fixed offset from the position where a match is found. The offset is a hexadecimal value.</p> <p>The <code>"match"</code> string can be an ASCII string, like <code>"\$serial\$"</code>. It can also contain binary values, which you specify with <code>\ddd</code> or <code>\xhh</code> where <code>ddd</code> is</p>

a decimal number of up to three digits and *hh* is a hexadecimal number of up to two digits (thus, the codes `\27` and `\x1b` are the same).

When the code `\U*` appears in the string, a zero byte is added to the match pattern after each byte. The purpose is to make matching Unicode strings easier. The code `\A*` reverts back to single-byte characters.

If a backslash must be matched, it must be doubled in the match field.

The starting serial number itself and its width in characters or bytes are decimal values. The serial number can be stored in one of three formats:

Binary	The serial number is stored as an integer, in Little Endian byte order. The width of the serial number will typically be 1, 2, or 4, for 8-bit, 16-bit and 32-bit integers respectively, but other field sizes are valid.
ASCII	The serial number is stored as text, using ASCII characters. The number is stored right-aligned in the field size of the serial number, and padded with zero digits on the left. For example, if the serial number is 321 and the width is 6, the serial number is stored as the ASCII string "000321".
Unicode	The serial number is stored as text, using 16-bit wide Unicode characters. The width for the serial number should be an even number.

Settings for serialization and other configurations are stored in a file that has the same name as the target (ELF) file, but with the extension `".prog"` added to it.

Updating Black Magic Probe Firmware

At the time of this writing, the latest “stable” firmware is version 1.6.1 from May 2017. Since then, support for more micro-controllers has been added and quite a few minor improvements were committed to the GitHub project. There is therefore good reason to update the firmware of the Black Magic Probe to a recent “development version”.

You can build the latest firmware yourself, but you do not need to. A pre-compiled “daily” build of the development release is available on what remains of the Black Sphere Technologies’ web site. See chapter [Further Information](#) on page 58.

An essential step for Microsoft Windows is to complete the set-up for DFU. See the instructions in [Setting up the Black Magic Probe](#) on page 10. As noted in that section, both the DFU interfaces for normal mode and DFU mode must be installed.

The next step is to install `dfu-util` for your operating system. For Microsoft Windows, download a “binaries” release (see [Further Information](#) on page 58 for the download location) and unpack it in a directory of your choice. For Linux, it is more convenient to use the package manager of your distribution to get the latest version; for example:

```
$ sudo apt-get install dfu-util
```

The options on `dfu-util` for updating the firmware are:

```
dfu-util -d 1d50:6018,:6017 -s 0x08002000:leave -D blackmagic-native.bin
```

On Linux, you may need to run the command with `sudo` (this depends on whether a `udev` rules file has been installed for the Black Magic Probe, see [Setting up the Black Magic Probe](#)).

You can check which firmware version you have with the GDB monitor command (after connecting it as an extended-remote target).

```
(gdb) monitor version
Black Magic Probe (Firmware fbf1963) (Hardware Version 3)
Copyright (C) 2015 Black Sphere Technologies Ltd.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

The development release of the firmware uses a GitHub hash instead of a version number. In the above snippet, it is indicated as `Firmware fbf1963`. A drawback of a hash is that they are not monotonically incrementing: a more recent firmware

may have a hash value that is lower than the previous version. To find out at what position on the commit timeline a particular hash sits, you have to go to the GitHub project for the Black Magic Probe, and search that repository for the hash number. The main page for the search results will then tell you that it couldn't find any *code* matching the hash, but to the left of that message is a selection list for Code, Commits, Issues, and a few others. If you click on "Commits" you will get a summary of the relevant commit, plus the date of that commit.

Further Information

Black Magic Probe: The GitHub project for the Black Magic Probe holds the firmware, documentation and schematics.

<https://github.com/blackspere/blackmagic>

Automated builds of the development version of the firmware (which is ahead of the released version, but may not be fully tested) can be found at:

<http://builds.blackspere.co.nz/blackmagic/>

Zadig: A utility for installing the drivers for SWO tracing and firmware update, see chapter [Setting up Black Magic Probe](#) on page 9.

<https://zadig.akeo.ie/>

gdbgui: Various GDB front-ends were mentioned in chapter [Requirements for Front-ends](#) (page 8), but we have singled out gdbgui because it is cross-platform and open source, and it offers the required features in a simple interface.

<https://www.gdbgui.com/>

Orbuculum: A set of utilities to process the output ARM Cortex Debug interface (notably SWO tracing), see section [SWO Tracing](#) on page 37.

<https://github.com/orbcode/orbuculum>

Common Trace Format: The specification of the binary format as well as the Trace Stream Description Language (TSDL), see chapter [The Common Trace Format](#) on page 43.

<https://diamon.org/ctf/>

dfu-util: A utility to update the firmware of USB devices that support the DFU protocol.

<http://dfu-util.sourceforge.net/>

Index

- !**
 - 1BitSquared, 3
 - .gdbinit file, 14, 18, 20, 38
 - .prog file, 55
- A**
 - Akeo Consulting, 10
 - ARM Cortex, 3, 35
 - Asynchronous encoding, 25, 31, 39, 41
 - Attach target, 14
- B**
 - Babeltrace, 47
 - barectf utility, 44, 49
 - Base (number), 47
 - Bit rate, 39
 - BKPT (instruction), 35
 - Black Sphere Technologies, 3, 9
 - bmdebug front-end, 18, 27, 35
 - bmflash utility, 28, 54
 - bmscan utility, 9, 11, 13, 53
 - bmtrace utility, 40, 41, 44
 - Bootloader (MCU), 16, 19
 - Break,
 - on exceptions, 26
 - Breakpoint, 6, 22, 24, 30, 33, 42
 - command list, 42
 - disable, 30
 - enable, 30
 - enable / disable, 24
 - hardware, 7, 21, 30, 42
 - software, 34
- C**
 - Call stack, 24
 - CDC class driver, 9-11
 - cgdb, 8
 - Channel (tracing), 50
 - Channels (tracing), 37
 - Checksum (vector table), 20, 29
 - Clone (debug probe), 34
 - CMSIS, 37
 - Code instrumentation, 34, 42
 - Command list, 42
 - Commands (GDB), 15
 - attach, 14
 - compare-sections, 20, 53
 - continue, 42
 - define, 15
 - file, 19
 - load, 19
 - monitor, 8, 14, 25, 56
 - run, 21
 - start, 21
 - trace, 31
 - user-defined, 15, 20
 - Common Trace Format, 32, 41, 43, 58
 - compare-sections command, 20, 53
 - Conditional compilation, 34
 - Connector,
 - MIPI10, 12
 - Console (GDB), 8, 18
 - continue (command), 42
 - CoreDebug, 36
 - CoreSight architecture, 39
 - Cortex Debug header, 12, 37
 - CTF packet, 44
- D**
 - DDD, 5, 8
 - Debug probe, 5, 6
 - Debug symbols, 19
 - Debugger attached check, 36
 - Development release (firmware), 56
 - Device Manager (Microsoft Windows), 9
 - DFU protocol, 9, 10, 56
 - dfu-util, 56, 58
 - DHCSR, 36
 - dialout group, 11

DiaMon, 43
Disable breakpoint, 24, 30
DTR (serial port), 14

E

Eclipse, 5, 8
ELF file, 20, 28, 54
elf-postlink utility, 21
Enable breakpoint, 24, 30
Ethernet, 45
Event,
 header, 45
Exceptions, 26
Execution point, 29

F

file command, 19
Firmware update, 9, 10, 56
Fixed-point numbers, 47
Flash memory, 7, 8, 15
 programming, 15, 53
Flash Programmer, see also bmflash, 54
Frame (call stack), 25
Front-end, 5, 8
 bmdebug, 18, 27, 35
Function key, 30

G

Gait, J., 43
GDB commands, see Commands, 15
GDB console, 8, 18
gdbgui, 8, 17, 18, 58
gdbserver, 3, 5, 6, 9, 10, 13, 25
GitHub, 4, 56, 58

H

HardFault handler, 36, 37
Hardware breakpoint, 7, 21, 30, 42
HOME environment variable, 14

I

IDC header, 12
Installing Black Magic Probe, 9

Instrumenting code, 34, 42
Interrupt Service Routine, 7
ITM, 39

J

Jitter, 49
JTAG, 3, 5, 12, 14

K

KDdbg, 5, 8
Keil ULINK-ME, 6

L

Law of the Hammer, 43
LED, 10, 13, 14, 26
Level shifters, 12, 25
License, 4
Linux Foundation, 43
Little Endian, 55
load (command), 19
LPC micro-controllers, 16, 20, 28, 40, 54
LTTng, 44

M

Manchester encoding, 25, 31, 39, 41
Maslow, Abraham, 43
Matloff, Norman, 4
MEMMAP register, 19
Memory (display/set), 24
Metadata file (CTF), 32, 43
MIPI10 connector, 12
monitor, 56
monitor (command), 8, 14, 25, 56
Morse code, 26

N

Nemiver, 8
newlib, 35
Non-intrusive debugging, 33
Non-intrusive debugging., 43
NRZ, 39
NTRACE macro, 51

Number base, 47

O

Orbuculum, 38, 40, 58

P

Packet,

- header, 45, 50, 52

- layout, 44

Packet-based protocol, 45

Pogo-pins, 12

Post-mortem analysis, 33

printf, 35

Probe effect, 43

Production code, 34

Protocol,

- packet-based, 45

- stream-based, 45

Push-button (on board), 10

R

RS232, 45

RS232, see UART, 33

RSP, 5, 6, 40, 54

run (command), 21

Run-time tracing, 7, 33

S

Salzman, Peter J., 4

Scan targets, 25

Section (ELF file), 54

Segger J-Link, 6

Semihosting, 31, 34, 36

Serial number, 54

Serial port, 9

Serial terminal, 33

Serialization, 54

Software breakpoint, 34

sprintf, 43

ST-Link clone, 34

Stable release (firmware), 56

Stack pointer, 37

start (command), 21

stderr, 35

Stimulus ports, 37

STM32 micro-controllers, 16, 40

Stop & Stare, 33

Stream-based protocol, 45

sudo, 11, 56

SVC (instruction), 35

SW-DP protocol, 14

SWCLK, 12

SWD, 3, 5, 12, 14, 16

SWDIO, 12

SWO tracing, 31, 37, 45, 48, 50, 51

SYSMEMREMAP register, 20

T

Tag-connect, 12, 13

Target,

- attach, 14

- list, 25

- scan, 25

Time stamp, 41

TPIU, 39

Trace capture, 9-11

trace command, 31

Trace Viewer, see also bmtrace, 40

tracegen utility, 44, 47, 49, 51, 52

traceswo command, 39

TRACESWO pin, 12, 34, 37

Tracing, 7, 33

TSDL, 43, 44, 45, 58

TUI, 5, 8

typealias, 47

typedef, 47

U

UART, 9, 10, 33

udev rules, 11, 56

Unicode, 55

USB, 45

USB ID, 10

User-defined command, 15, 20

V

Variable watch, 24, 30

VisualGDB, 8
Voltage level, 12

W

Watch variable, 24, 30
Watchpoint, 22, 24
WinGDB, 8
WinUSB device, 9, 10

Y

YAML, 44
Yiu, Joseph, 38

Z

Zadig, 10, 58