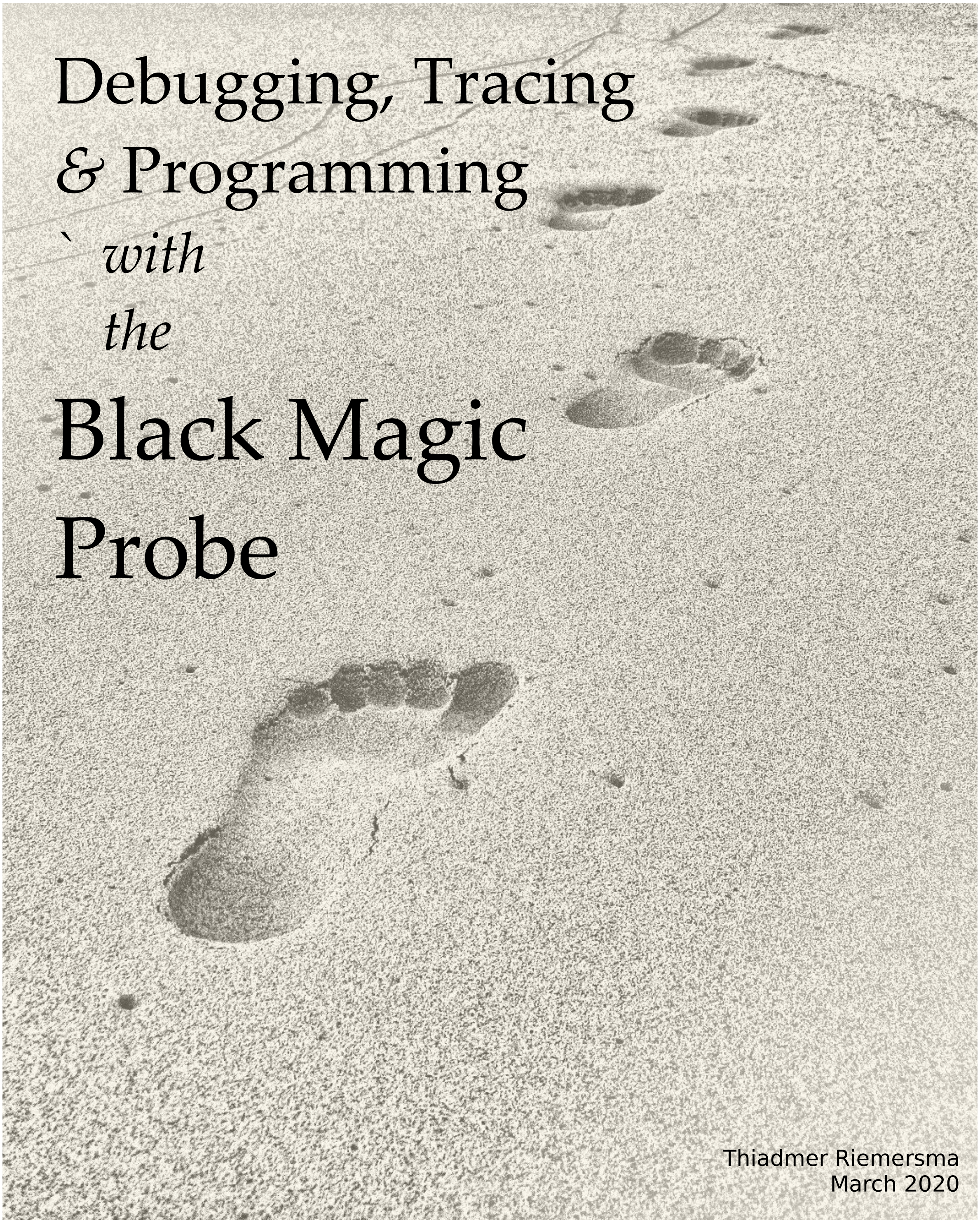


Debugging, Tracing & Programming ` *with the* Black Magic Probe

Thiadmer Riemersma
March 2020

Table of Contents

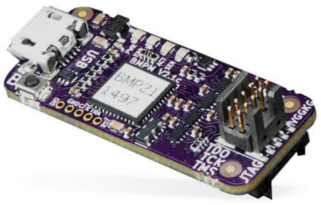
Introduction

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

Separate from the MCU core, the ARM Cortex series have a Debug Access Port (DAP) that gives you access to the debugging features of the micro-controller. On older architectures, the debugging interface used the JTAG port and protocol, but for the ARM Cortex series, a new protocol that required less physical pins was designed: the ARM *Serial Wire Debug* protocol (SWD). This protocol gives you access to features like single-stepping, hardware breakpoints and watchpoints, dumping memory regions and programming Flash memory. Like was the case with the JTAG interface, the SWD interface 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* — it's rudimentary, though. See Requirements for Front-ends (page 9) 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 Salzman,[1] 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 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 three chapters: the first on the hardware and software support in the Black Magic Probe, the second on generic techniques to perform tracing efficiently, and the third on particular applications of run-time tracing.

1 Matloff, Norman and Peter Jay Salzman; *The Art of Debugging with GDB, DDD, and Eclipse*; No Starch Press, 2008; ISBN 978-1593271749.

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.

The final (short) chapters are on updating the firmware of the Black Magic Probe itself and adding support for new micro-controllers to the GUI utilities that accompany this guide.

License

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 or interface to connect the workstation to the micro-controller's debug port & protocol. On the ARM Cortex processors, the most common debug and Flash programming protocols are JTAG and SWD (Serial Wire Debug).

In the idiom of remote debugging, the *target* is the device being debugged and the *host* is the workstation that the debugger runs on. The interface between host and target is the *probe*. A debug probe typically connects to the workstation's USB, RS232 or Ethernet port.

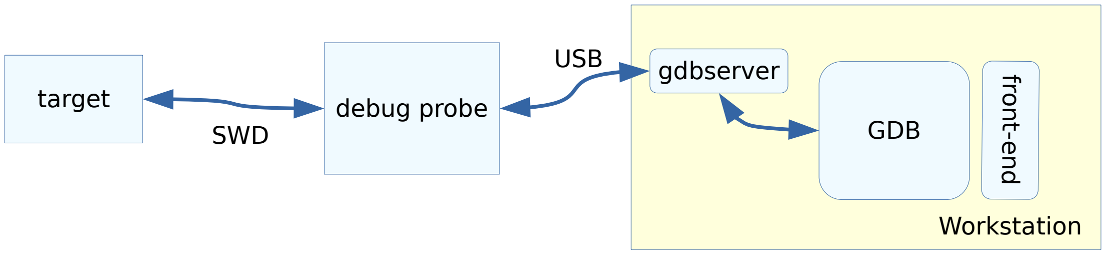
GDB Architecture

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.

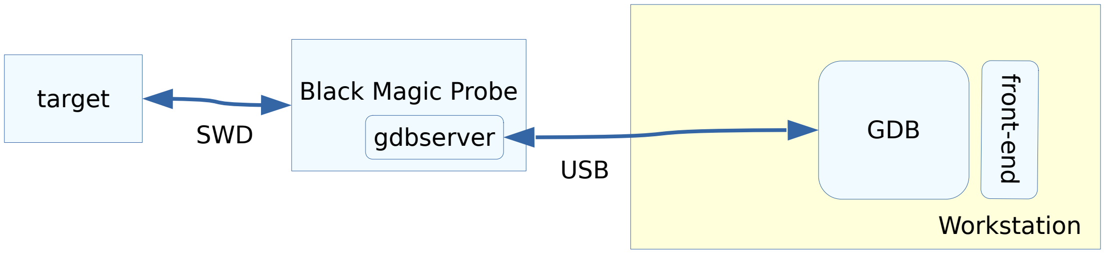
To debug a different system than the one where the debugger runs on, GDB provides the *Remote Serial Protocol* (RSP). This is a simple text-based protocol with which GDB on the workstation communicates with a debugger "stub" on the target system. This stub acts as a server that GDB connects to, over 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 according 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 OpenOCD project is an example of this set-up. The main openocd program opens a Telnet port for the communication link to GDB and a USB, RS232 or Ethernet connection to the debug probe.

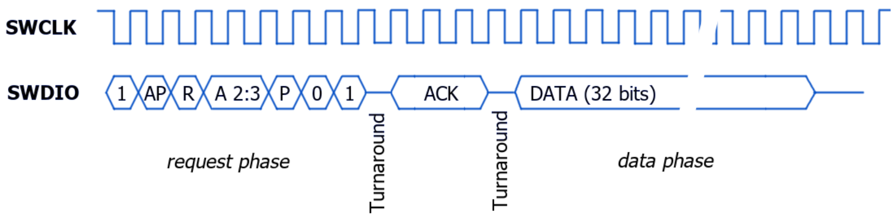
The Black Magic Probe embeds gdbserver. One advantage if this design is that gdbserver has in-depth knowledge of the capabilities of the debug probe as well as what the debug probe has determined about the target. The only configuration that needs to be done in GDB is the (virtual) serial port of the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



The Serial Wire Debug Protocol in a Nutshell

The Serial Wire Debug protocol (SWD) is designed as an alternative to the JTAG protocol, for micro-controllers with a low pin count. At the physical layer, it needs two lines at the minimum (plus ground), as opposed to five for JTAG. These are the clock (SWCLK, driven by the debug probe) and a bi-directional data line (SWDIO). Tracing output goes over a third (optional) line: TRACESWO, but using an unrelated protocol (independent from SWCLK).

The SWCLK signal is driven by the debug probe, regardless of the direction of the transfer. Each transfer starts with a request, that the probe sends to the target. The target replies by sending an acknowledgement back. After that, a *data phase* follows, which may be in either direction, depending on the request. As is apparent, the direction of the SWDIO line switches between input and output at least once during a transfer, on both sides. The SWD protocol calls this the *turnaround*, and there is an extra clock cycle for each turnaround in the transfer.



The above example is for a write transfer; in a read transfer there is no turnaround after the ACK. The request starts with a start bit (always 1). The AP bit is 1 if this transfer is for the AHB bus (or another access port) beyond the debug port. The R bit is 1 for a read request and a 0 for a write request. There are two address bits, to access the debug registers. The P bit is a parity bit, it is set such that the sum of the bits in the request byte is even. Following the P bit are a stop bit and a park bit, which are 0 and 1 respectively.

The ACK is a three bit sequence with the value 1 (on success), sent with the low bit first. The data is likewise transmitted low bit first. After the 32-bits of data are transmitted, a parity bit follows (calculated in the same way as the parity in the request byte).

With two address bits in a transfer request, you can only address four registers. To access code or data memory, the access port of the AHB provides the TAR register. In this register you set a memory address so that you can read from or write to that memory location on a subsequent transfer. A peculiarity of the SWD protocol is that a read transfer returns the value from the previous transaction. Hence, to read the current value of a register or memory location, you need to perform the read operation twice, and discard the first result.

Before a micro-controller's SWD port is serviceable, an initialization sequence must be performed, part of which is to switch the protocol from JTAG to SWD. Some ARM Cortex micro-controllers do not support JTAG, but the protocol requires that the JTAG-to-SWD switch is still performed.

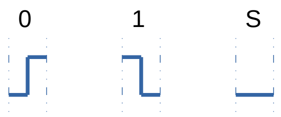
TRACESWO Protocol

The TRACESWO protocol is independent from the SWD protocol. You can trace without debugging as well as debug without tracing. The data is transmitted over a single line using one of two serial format: 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) respectively.

The asynchronous encoding is in essence TTL-level UART, with a start bit, eight data bits, one stop bit and no parity. As is common with UART protocols, the target and the debug probe must use the same bit rate within a narrow margin. Since the UART clock is typically derived from the micro-controller clock, at high bit rates it becomes harder to find a bit rate shared by both the target and the debug probe within the required margin.

Manchester encoding, on the other hand, has the property that the clock frequency can be established from the data stream. This makes it a self-adapting protocol, tolerant to jitter and unsusceptible to clock drift. These properties make Manchester encoding the option of choice for micro-controllers that lack hardware support for SWO tracing (such as the ARM Cortex-M0 and Cortex M0+ architectures), because it is easier to implement it with bit-banging. A drawback of Manchester encoding is that the encoding takes two clocks per bit, which means that the maximum bit rate is typically half as high as for asynchronous encoding.

The physical Manchester protocol on the TRACESWO pin transmits sequences of 1 to 8 bytes, where each sequence is prefixed with a start bit (a 1-bit) and suffixed with a "space".

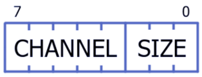


The pin is low on idle; a 0-bit has a rising edge halfway the bit period, a 1-bit has a falling edge halfway the bit period, and a space is a low level for the full bit period.

Obviously, since a 1-bit starts high, if the pin is low at the start of the bit period, there is also a rising edge at the start of the 1-bit. This occurs when the previous bit is also a 1-bit, or when the previous state was idle or space. Similarly, there is a falling edge at the start of a 0-bit if the pin is high at the start of the 0-bit, which occurs when the previous bit was also a 0-bit. A space resets the decoder state back to idle.

Although Manchester is a bit transmission protocol, the ITM always transmits a multiple of 8 bits. After a start bit and up to 64-bits (8-bytes) have been transmitted, a space follows and after that (if there is more data to transmit) a new start bit plus another sequence of data. This short interruption after every 64-bits is to resynchronize the bit stream. The start bit is needed to determine the clock frequency of the protocol (the start bit is transmitted from idle state, so there is a rising edge at the start of the bit and a falling edge half way), and the space at the end of a sequence is needed to properly decode the *next* start bit (it needs to come after a known state).

At a higher level the TRACESWO protocol transmits *packets* consisting of an 8-bit packet header followed by a 32-bit payload. The protocol uses leading-zero compression on the payload, where *leading* means *most significant* (because the data is actually transmitted with the least-significant bit first).



The header byte contains the channel number in the highest five bits. The low three bits indicate the number of payload bytes that follow; the value can be 1, 2 or 3, where 3 means that *four* payload bytes follow.

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

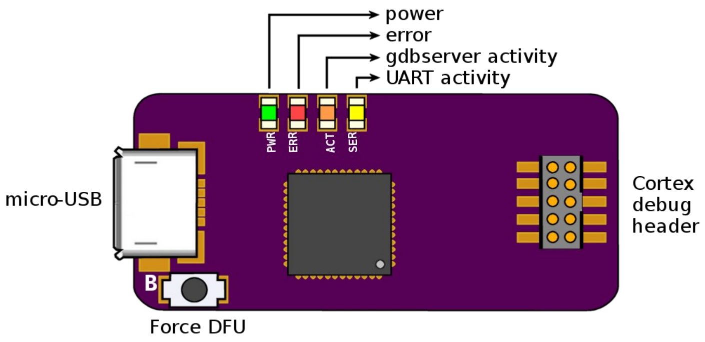
GDB has powerful and flexible commands, but its console interface falls short of what is needed. Code is hard to follow if you only see a single line at a time. While you can routinely type the list command on the “`(gdb)`” prompt, it is clumsy and it distracts you from focusing on locating any flaws 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 35.

Setting up the Black Magic Probe

The Black Magic Probe has a micro-USB connector for connection to a workstation and a 2×5-pins 1.27mm pitch “debug” header for connection to the target micro-controller. See section Connecting the Target on page 15 for details on the Cortex Debug header.



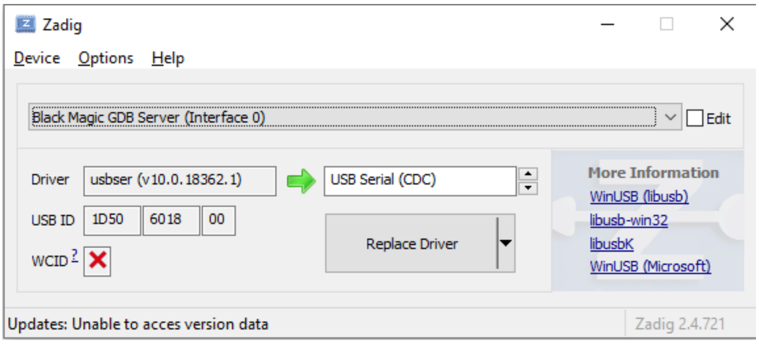
One the reverse site, the Black Magic Probe has a 4-pins 1.25mm pitch “PicoBlade” connector for a secondary 3.3V TTL-level UART.

Next to the three connectors, the Black Magic Probe has an on-board switch (that you will only use to upgrade the firmware to the Black Magic Probe) and four LEDs that signal power and activity status.

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 vendor-specific interfaces for firm-ware update (via the DFU protocol) and trace capture.

On Windows 10, no drivers are needed (a class driver is built-in and automatically set up). 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 Techno-logies, as well as with this guide. Alternatively, you can set up the CDC driver for the Black Magic Probe with the free utility “Zadig” by Akeo Consulting (see also Further Information on page 79). When using Zadig, you need to set up both interfaces 0 (“Black Magic GDB Server”) and 2 (“Black Magic UART Port”) to “USB Serial (CDC)”. You may need to first select List All Devices in the Options menu to see the interfaces of the Black Magic Probe.



Black Magic GDB Server (Interface 0)

Driver: usbser (v10.0.18362.1) → USB Serial (CDC)

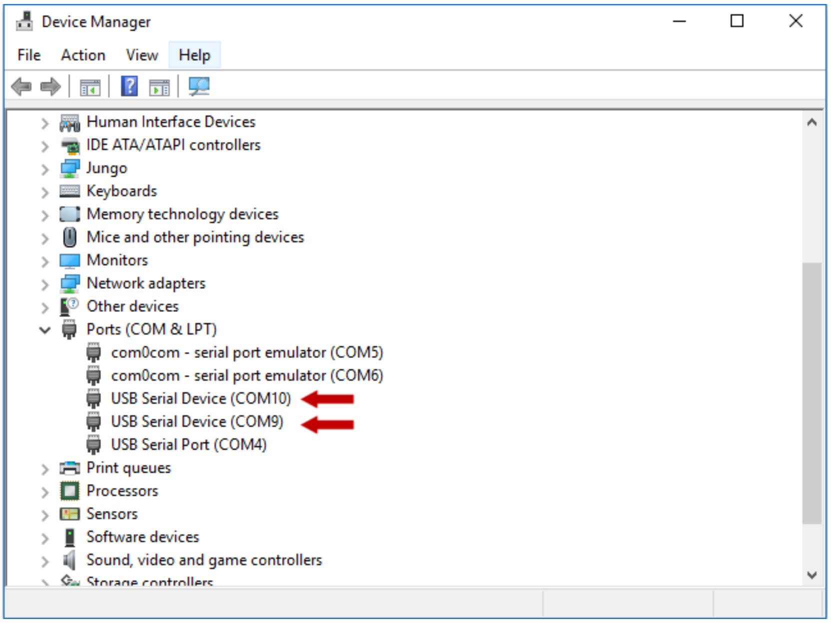
USB ID: 1D50 6018 00

WCID ?

Updates: Unable to acces version data

Zadig 2.4.721

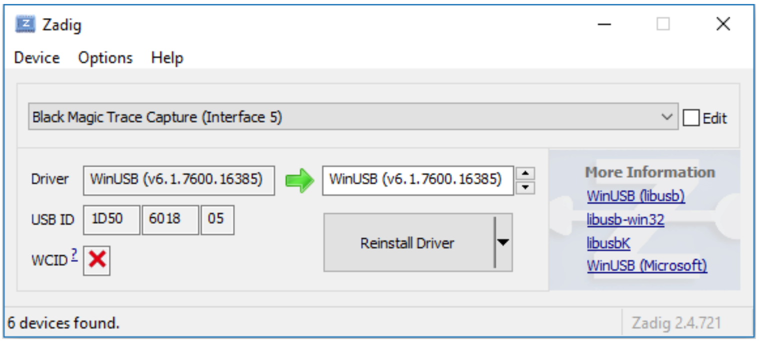
Once the CDC driver is configured, two COM ports are assigned to the Black Magic Probe. You can find out which ports in the Device Manager, where they are listed under the item “Ports (COM & LPT)”. Alternatively, you can run the `bmscan` utility on the command line (this is one of the utilities that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



- Human Interface Devices
- IDE ATA/ATAPI controllers
- Jungo
- Keyboards
- Memory technology devices
- Mice and other pointing devices
- Monitors
- Network adapters
- Other devices
- Ports (COM & LPT)
  - com0com - serial port emulator (COM5)
  - com0com - serial port emulator (COM6)
  - USB Serial Device (COM10)
  - USB Serial Device (COM9)
  - USB Serial Port (COM4)
- Print queues
- Processors
- Sensors
- Software devices
- Sound, video and game controllers
- Storage controllers

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 either a WinUSB device or a libusbK device. The most convenient way to do so is by running the aforementioned “Zadig” utility (see Further Information on page 79).



Black Magic Trace Capture (Interface 5)

Driver WinUSB (v6.1.7600.16385) → WinUSB (v6.1.7600.16385)

USB ID 1D50 6018 05

WCID ? ✗

6 devices found. Zadig 2.4.721

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 or libusbK 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`.

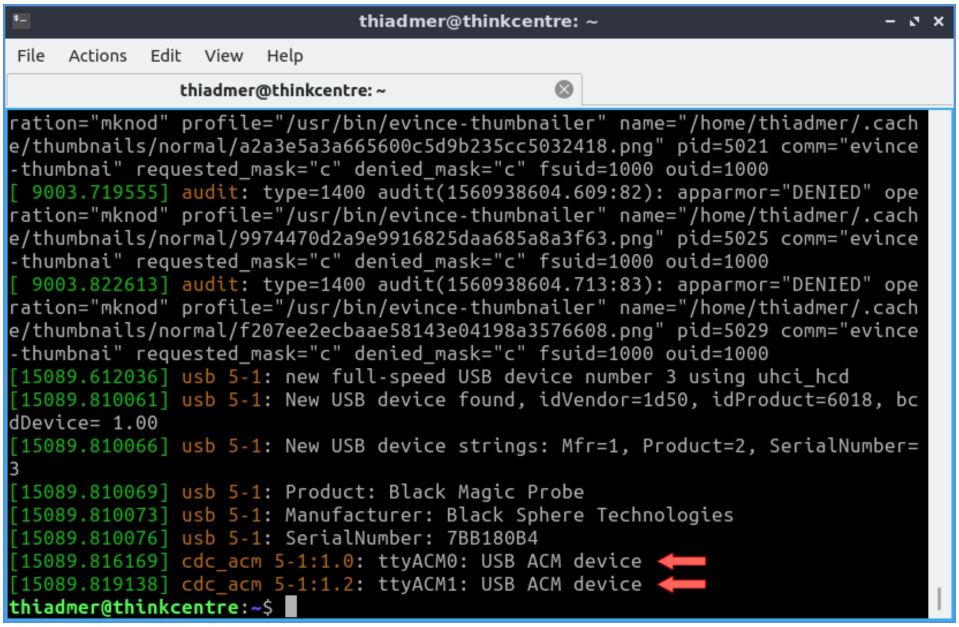
The choice between WinUSB and libusbK depends on the PC-hosted software that you wish to use for trace capture. The firmware upgrade tool `dfu-util` (see Updating Black Magic Probe Firmware on page 75) supports both WinUSB and libusbK. The debugger front-end and trace viewer that accompany this guide also support both WinUSB and libusbK, and in this case WinUSB is preferred (because it is pre-installed). The Windows port of the Orbuculum tool-set (see Monitoring Trace Data on page 51), however, is based on libusb and requires the libusbK driver.

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. For example, if the Black Magic Probe is the only virtual serial port connected to the workstation, the assigned device names will be `/dev/ttyACM0` and `/dev/ttyACM1`.

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 (see also the arrows in the picture below). Alternatively, you can run the `bmscan` utility from inside a terminal (`bmscan` is a companion tool to this guide).



```
ration="mknod" profile="/usr/bin/evince-thumbnailer" name="/home/thiadmer/.cach
e/thumbnails/normal/a2a3e5a3a665600c5d9b235cc5032418.png" pid=5021 comm="evince
-thumbnai" requested_mask="c" denied_mask="c" fsuid=1000 ouid=1000
[ 9003.719555] audit: type=1400 audit(1560938604.609:82): apparmor="DENIED" ope
ration="mknod" profile="/usr/bin/evince-thumbnailer" name="/home/thiadmer/.cach
e/thumbnails/normal/9974470d2a9e9916825daa685a8a3f63.png" pid=5025 comm="evince
-thumbnai" requested_mask="c" denied_mask="c" fsuid=1000 ouid=1000
[ 9003.822613] audit: type=1400 audit(1560938604.713:83): apparmor="DENIED" ope
ration="mknod" profile="/usr/bin/evince-thumbnailer" name="/home/thiadmer/.cach
e/thumbnails/normal/f207ee2ecbaae58143e04198a3576608.png" pid=5029 comm="evince
-thumbnai" requested_mask="c" denied_mask="c" fsuid=1000 ouid=1000
[15089.612036] usb 5-1: new full-speed USB device number 3 using uhci_hcd
[15089.810061] usb 5-1: New USB device found, idVendor=1d50, idProduct=6018, bc
dDevice= 1.00
[15089.810066] usb 5-1: New USB device strings: Mfr=1, Product=2, SerialNumber=
3
[15089.810069] usb 5-1: Product: Black Magic Probe
[15089.810073] usb 5-1: Manufacturer: Black Sphere Technologies
[15089.810076] usb 5-1: SerialNumber: 7BB180B4
[15089.816169] cdc_acm 5-1:1.0: ttyACM0: USB ACM device
[15089.819138] cdc_acm 5-1:1.2: ttyACM1: USB ACM device
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 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 `55-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. The contents of this file are:

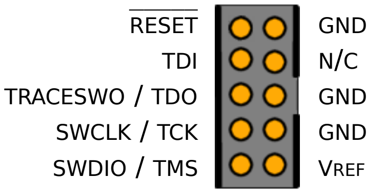
```
# Standard mode
ACTION=="add", SUBSYSTEM=="usb_device", SYSFS{idVendor}=="1d50", SYSFS{idProduct}=="6018", MODE="0666"
ACTION=="add", SUBSYSTEM=="usb", ATTR{idVendor}=="1d50", ATTR{idProduct}=="6018", MODE="0666"

# DFU mode
ACTION=="add", SUBSYSTEM=="usb_device", SYSFS{idVendor}=="1d50", SYSFS{idProduct}=="6017", MODE="0666"
ACTION=="add", SUBSYSTEM=="usb", ATTR{idVendor}=="1d50", ATTR{idProduct}=="6017", MODE="0666"
```

The provided `udev` rules file does not configure stable device names for the `ttyACM` devices for the Black Magic Probe. If so desired add the following lines to the rules file (`55-blackmagicprobe.rules`):

```
SUBSYSTEM=="tty", ATTRS{interface}=="Black Magic GDB Server", SYMLINK+="ttyBMPGDB"
SUBSYSTEM=="tty", ATTRS{interface}=="Black Magic UART Port", SYMLINK+="ttyBMPUart"
```

Connecting the Target

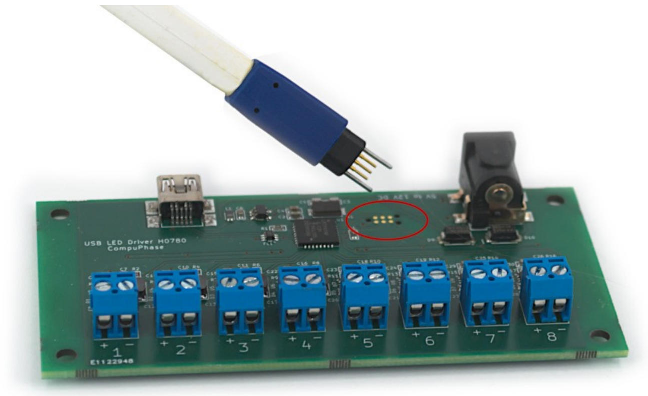
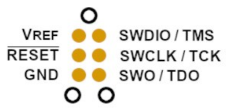
The Black Magic Probe has a 2×5-pins 1.27mm pitch IDC header. This is the Cortex Debug header for JTAG and SWD. If your target board has the same connector, the two can be readily connected with the provided ribbon cable.



For target boards that do not have 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 same 2×5-pins 1.27mm pitch Cortex Debug header on one side and a 7-pins 2.54mm pitch IDC header (single row) on the other. Note that the Cortex Debug header 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 “JTAG 7-pin adapter” (this is the silkscreen text on the board).

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 33). 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 programming/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 PCB & Software Design for Debugging & Programming on page 19 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
  SWO interface:  {9A83C3B4-0B99-499E-B010-901D6C2826B8}
```

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 22.

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
```

PCB & Software 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 may 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.

Depending on your micro-controller, a way to circumvent this is to enable the option `connect_srst` in the Black Magic Probe (see page 33 for the `connect_reset` command). The debug port of the ARM Cortex is designed such that it is active while the remainder of the micro-controller is in reset, so that a debug probe can attach to it. This is precisely what the `connect_reset` option does: it pulls the reset pin on the connector low when starting a SWDP scan and only releases it after the `attach` command.

Alternatively, you can often use system-specific pins to force a micro-controller into boot mode. The LPC series of micro-controllers from NXP have a $\overline{\texttt{BOOT}}$ 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, a tiny push-button or even a test pad, so that you can recover from an accidental pin redefinition.

If the pin redefinition of the SWD pins is by design, because you need these pins for other purposes, this will thwart your ability to debug the code. If possible, arrange the design such that the SWD pins are used for a non-essential function. Then, you can implement the firmware such that it redefines the SWD pins only when *not* running under control of a debugger. While debugging, you will miss the functionality that would otherwise be driven by redefined SWD pins, but you can debug the rest.

Two methods are available for the firmware to detect whether it is running under a debugger. The first is to test that the low bit of the *Debug Halting Control & Status Register* (DHCSR) is set. This works on a Cortex M3/M4/M7 micro-controller, but on the Cortex M0/M0+ micro-controller architecture, this register is not accessible from firmware (it is accessible from the JTAG/SWD interface).

```
if ((CoreDebug->DHCSR & 1) == 0) {
    /* not running under a debugger, free to redefine pins */
}
```

The alternative is to have a weak pull-up on the SWCLK pin and probe it (as a general-purpose I/O pin) on start-up. The Black Magic Probe pulls the clock line low (provided that it senses a voltage on the V_{REF} pin). This does require some pin juggling, though: you first have to configure the SWCLK pin as an "input" I/O pin (with a pull-up) to be able to read it, and depending on the value read, either quickly change it back to SWCLK pin, or set it to its intended configuration. Also, if the pin is connected to other circuitry that drives the pin low, this trick won't work.

Accessories

The tag-connect cable for the SWD interface and the break-out board with a single-row 7-pins 2.54mm pitch IDC header were already mentioned. See chapter Further Information on page 79 for the part numbers and links to the manufacturers or distributors. Also available is an adapter board that adapts the 10-pin Cortex Debug Header to a standard 20-pin JTAG header.

The Black Magic Probe comes without enclosure, but if you have access to a 3D printer, it is recommended to print one. A few designs are freely available, see chapter Further Information on page 79.

design by Michael McAvoy

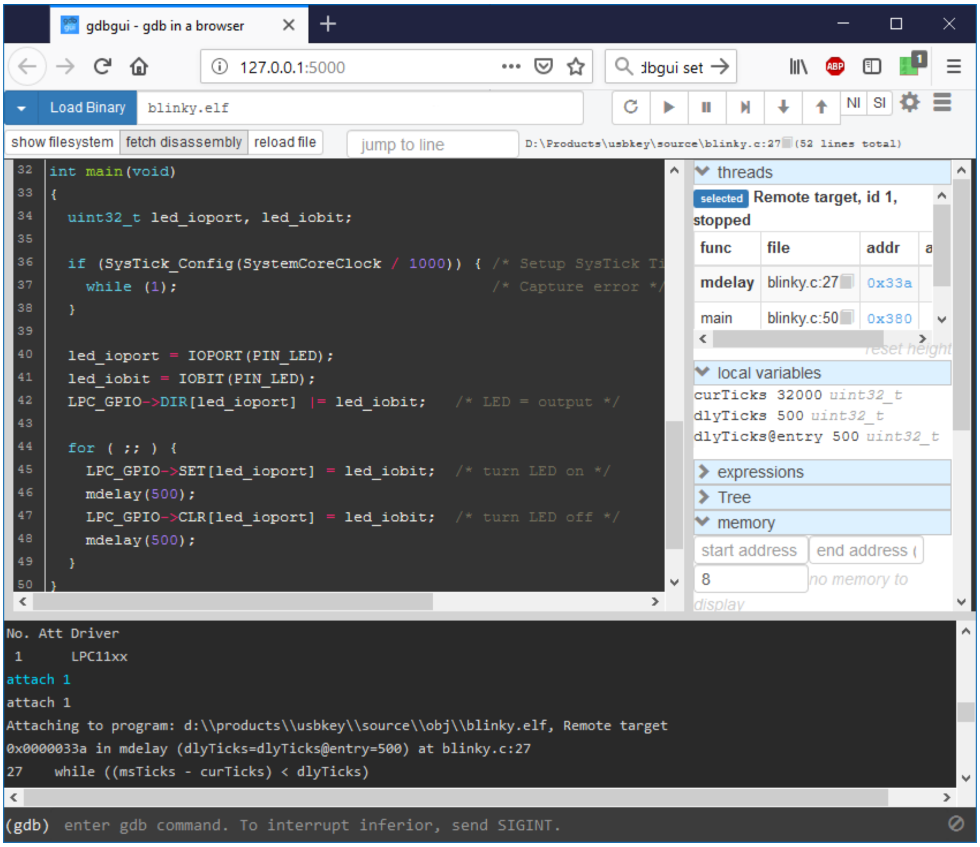
design by Emil Fresk

It feels fitting to print these enclosures in black, but you are of course free to choose any colour.

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 4), 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".



D:\Products\usbkey\source\blinky.c:27 (52 lines total)

```
int main(void)
{
  uint32_t led_ioport, led_iobit;

  if (SysTick_Config(SystemCoreClock / 1000)) { /* Setup SysTick Ti
    while (1);                                  /* Capture error */
  }

  led_ioport = IOPORT(PIN_LED);
  led_iobit = IOBIT(PIN_LED);
  LPC_GPIO->DIR[led_ioport] |= led_iobit;   /* LED = output */

  for ( ;; ) {
    LPC_GPIO->SET[led_ioport] = led_iobit;  /* turn LED on */
    mdelay(500);
    LPC_GPIO->CLR[led_ioport] = led_iobit;  /* turn LED off */
    mdelay(500);
  }
}
```

Remote target, id 1, stopped

func	file	addr
mdelay	blinky.c:27	0x33a
main	blinky.c:50	0x380

local variables

```
curTicks 32000 uint32_t
dlyTicks 500 uint32_t
dlyTicks@entry 500 uint32_t
```

memory: 8

```
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 18. 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.

Flash Memory Remap

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 SYSMEMREMAP register to 2 from GDB (as is apparent, SYSMEMREMAP 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 18:

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

document mmap-flash
  Set the SYSMEMREMAP 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 SYSMEMREMAP 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.

Reset Code Protection

On the STM32Fxx family of micro-controllers, the load command may give the following error:

```
(gdb) load
Error erasing flash with vFlashErase packet
```

This implies that read/write protection is set in the option bytes. No new code can up downloaded unless the option bytes are erased first — which in turn wipes the entire Flash memory. To erase the option bytes, use the monitor command.

```
(gdb) monitor option erase
0x1FFFF800: 0x5AA5
0x1FFFF802: 0xFFFF
0x1FFFF804: 0xFFFF
```

```
0x1FFFF806: 0xFFFF
0x1FFFF808: 0xFFFF
0x1FFFF80A: 0xFFFF
0x1FFFF80C: 0xFFFF
0x1FFFF80E: 0xFFFF
```

After erasing the option bytes, the micro-controller must be power-cycled to reload them. GDB will loose the connection to the target, so after power-cycling, you must rescan and re-attach to the target again.

When code protection is enabled on the LPC micro-controller series, Flash memory must also be fully erased before new firmware can be downloaded. These micro-controllers do not use option bytes, however. The following `monitor` command accomplices this:

```
(gdb) monitor erase_mass
```

Unfortunately, only a subset of the target drivers of the Black Magic Probe support this command. See also Using the BlackMagic Flash Programmer on page 73 as an alternative tool for downloading firmware via the Black Magic Probe. The `bmflash` utility has an option to erase the entire flash memory even for target drivers that do not support the `monitor erase_mass` command.

Verify Firmware Integrity

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 commands listed below are a subset of the full GDB command & parameter set for listing the source code of a target. These are the most common commands.

`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 +`").

`list -`	Show the preceding lines (above the current position).
`info sources`	List the names of the source files for the target executable.
`info line *address`	Print the line number and source file associated with the address. The address parameter must start with "`0x`" if it is in hexadecimal.

Stepping and Running

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

`start`	Start or re-start the program and break at function `main`. If no function "main" exists, it is the same as the `run` command.
`run`	Start or re-start the program (from the beginning).
`continue / c`	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 `continue 10` will run 10 more iterations before stopping at the breakpoint again.
`step / s`	Step a single source line, step *into* 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.
`next / n`	Step a single source line, step *over* 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.
`until / u`	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 `until` command, and then it runs until that line is reached.
`finish / fin`	Step out of the current function and stop at the location from where it was called.

The `step` command will not step *into* functions *without* symbolic information, *such as a* function from the standard library. Instead, `step` will step *over* the function call, and behave identical as `next` in this case. You can also instruct GDB to skip stepping into particular functions, with the `skip` command. When a function has a call to another function in its parameter list, the `step` command will step into the nested function first, and this nested function might have a completely uninteresting implementation. The `skip` command is very flexible; the two most common variants are below.

`skip function` *`name`*	Skip the stated function (or skip the current function if no name is given).
`skip file` *`name`*	Skip all functions in the stated file (or skip all functions in the current file, if no name is given).

When stepping through optimized code, the current line may jump back and forth on occasion, because the compiler has re-arranged the generated machine code. See section Debugging Optimized Code on page for 41 details.

Breakpoints and watchpoints

When creating a breakpoint or watchpoint, it gets assigned an ID. This is simply a unique number to identify the breakpoint or watchpoint. Several of the commands listed below take the breakpoint ID as a parameter.

`break` *`line`*
`b` *`line`*
	Set a breakpoint at the line number in the current source file.

`break` *`file`*`:`*`line`*
`b` *`file`*`:`*`line`*
	Set a breakpoint at the line number in the specified source file.

`break` *`function`*
`b` *`function`*
	Set a breakpoint at the start of the named function.

`tbreak ...`	Sets a one-time breakpoint, which auto-deletes itself as soon as it is reached. The `tbreak` command takes the same parameter options as the `break` command.
`watch` *`expr`*	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.
`info break`	Show the list of breakpoints and watchpoints, together with the sequential index numbers (sometimes called the breakpoint IDs) that each breakpoint got assigned.

`delete`
`delete` *`id`* `...`
	When given without parameters, this command deletes all breakpoints.
	Otherwise, if one or more breakpoint IDs follow the command (separated by spaces), the command deletes the breakpoints with those IDs.

`clear`	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.

`clear` *`line`*
`clear` *`file:line`*
`clear` *`function`*
	Delete a breakpoint on the given line or function. It allows the same options as the `break` command.

`disable` *`id`* `...`	Disables the breakpoints with the given IDs. There may be one or more IDs on the command list (separated by spaces).
`enable` *`id`* `...`	Enables the breakpoints with the given IDs. There may be one or more IDs on the command list (separated by spaces). You may also use `enable once` to enable the breakpoints, but disable them when they are reached.
`cond` *`id expr`*	Attaches a condition to the breakpoint with the given ID. The condition is what you would write between the parentheses of an "`if`" statement in the C language. For example: `cond 3 count == 5` causes breakpoint 3 to only halt execution when variable `count` equals 5 (assuming, of course, that variable `count` is in scope). When the expression is absent on this command, the condition is removed from the breakpoint (but the breakpoint stays valid).

`command` *`id`*
`...`
`end`
	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 53.

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.

Setting a breakpoint plus a condition on a breakpoint may be combined in a single step. To do so, put the keyword "`if`" followed by condition expression at the end of the `break` command. For example:

```
break blinky.c:168 if count == 5
```

The Cortex-M micro-controllers can also break on specific exceptions or interrupts. An exception trap is set with the `monitor vector_catch` command, see page 33. When the exception is caught, the micro-controller will halt on the first instruction of the exception/interrupt handler.

Examining Variables and Memory

In addition to the commands below, most front-ends show a variable's value when hovering the mouse cursor over it. Front-ends typically also allow to set "variable watches", which is the equivalent to the `display` command. The `gdbgui` front-end even allows you to add a graph for numeric variables, to give you a visualization of the value of the variable over time.

`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 GDB `print` command records each value that it prints in its "value history". Each entry in that value history is labelled. The first label is `$1` and the number is incremented for each successive `print` command. You can use these labels on subsequent `print` commands. The value history records the values at the time these were printed.

The Call Stack

`backtrace` *`num`*
`bt` *`num`*
	Show a list with the call-stack that lead to the current execution point. The call stack is optionally limited to the given number of levels.

`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` *`idx`*	Move to the given frame index (the `backtrace` command prints these index numbers).

After changing to a different stack frame (with the `up`, `down` or `frame` commands), 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 *Serial Wire Debug* devices (using the SW-DP protocol). The command prints the I/O voltage and the list of targets. See also the `tpwr` command (below) for the I/O voltage and the `targets` command for the device list.

`traceswo`
`traceswo rate`
	Enable the SWO capture pin to for trace capture. The `rate` parameter is the bitrate of the SWO trace protocol. It is required for asynchronous encoding, and redundant for Manchester encoding. The original Black Magic Probe only supports Manchester encoding.

`targets`	Show the detected targets. This is the same list as the one returned by the jtag_scan and swdp_scan commands. For each detected micro-con-

	troller, 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 15 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 the option to keep the target micro-controller in reset while scanning and attaching to it.
`hard_srst`	Resets the target by briefly pulling the $\overline{\text{RESET}}$ pin low on the 2×5 pin header (see page 15 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. *ARM Cortex-M* 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` Interrupt/exception service errors; an assortment of exceptions that don't fall in another category. `bus` Bus fault. `stat` Fault state error. `chk` Divide by zero, misaligned memory access, etc. `nocp` Missing coprocessor (on coprocessor instruction). `mm` Memory Manager fault. `reset` Core reset. Cortex-M0 and M0+ micro-controllers only support reset and hard fault exception catching. A hard reset cannot be caught, though.
`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`.	*Kinetis*
`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.	*nRF51xxx series*
`gpnvm_get`	Get value of the GPNVM register.	*SAM3N, SAM3S, SAM3U, SAM3X* *SAM4S*
`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).	*SAM3N, SAM3S, SAM3U, SAM3X* *SAM4S*
`lock_flash`	Lock Flash memory against accidental change.	*SAMD*
`unlock_flash`	Unlock Flash memory.	*SAMD*
`user_row`	Print the user row from Flash.	*SAMD*
`mbist`	Run the “Memory Built-In Self Test” (MBIST).	*SAMD*
`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.	*STM32Fxx, STM32L0x, STM32L1x, STM32L4xx*
`eeprom`	Set values in EEPROM (non-volatile memory). The first parameter is one of: `byte` — 8-bit value.	*STM32L0x, STM32L1x*

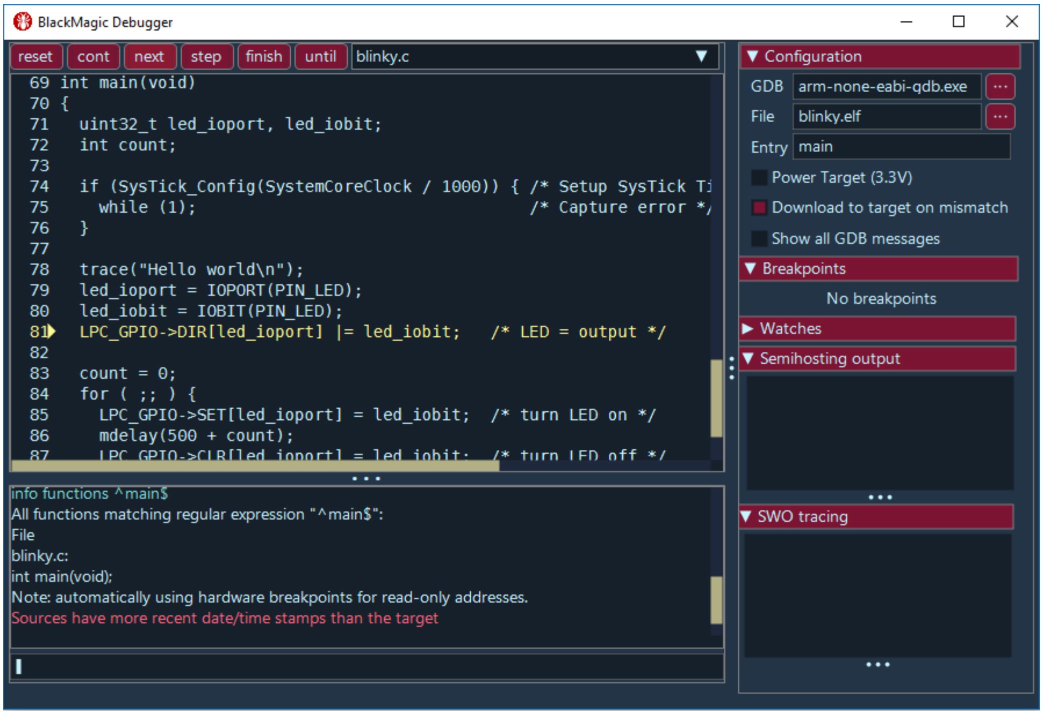
`halfword`	16-bit value.
`word`	32-bit value.

The second parameter is the address in the EEPROM.

The third parameter is the value (with the size as specified in the first parameter).

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 mis-match. The Prerequisite Steps described on page 23 are handled automatically. Another distinguishing feature of `bmdebug` is that it combines traditional debugging with run-time tracing.



```
int main(void)
{
  uint32_t led_ioport, led_iobit;
  int count;

  if (SysTick_Config(SystemCoreClock / 1000)) { /* Setup SysTick Ti
    while (1);                                  /* Capture error */
  }

  trace("Hello world\n");
  led_ioport = IOPORT(PIN_LED);
  led_iobit = IOBIT(PIN_LED);
  LPC_GPIO->DIR[led_ioport] |= led_iobit;   /* LED = output */

  count = 0;
  for ( ;; ) {
    LPC_GPIO->SET[led_ioport] = led_iobit;  /* turn LED on */
    mdelay(500 + count);
    LPC_GPIO->CLR[led_ioport] = led_iobit;  /* turn LED off */
```

```
info functions ^main$
All functions matching regular expression "^main$":
File
blinky.c:
int main(void);
Note: automatically using hardware breakpoints for read-only addresses.
Sources have more recent date/time stamps than the target
```

GDB arm-none-eabi-gdb.exe
File blinky.elf
Entry main
Power Target (3.3V)
Download to target on mismatch
Show all GDB messages

No breakpoints

Starting up

After loading an ELF file, bmdebug stops at function main in that code. You may set an alternative function as the entry point of the executable. If the entry point function (typically "main") cannot be found, bmdebug keeps the micro-controller in halted state, so that you can set a breakpoint at some code of interest before giving the run command (or pressing the "cont" button).

Unlike the bmflash utility (see page 73), 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 26 for more information on the checksum for LPC micro-controllers.

GDB Console and Command Line

The bottom-left section of the user interface for bmdebug is the GDB console and the command line for input to GDB. The GDB console shows the output of GDB. Some messages from GDB are filtered out by default. You can set the option "Show all GDB messages" in the "Configuration" section in the sidebar to see all output.

The command line keeps a history of commands that are typed in. The Ctrl+R key combination scrolls though earlier commands on the command line. Another feature is autocompletion of commands or parameters, on the TAB key. This is especially convenient when the parameter of a command is a file or a function: just type in the first few letters of the function or file name and press TAB. Pressing TAB multiple times cycles through all candidates.

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 27). 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
```

You may also type a function name or a line number as the parameter to the `list` command. This will make the source view jump to that line or to the start of the given function. The Ctrl+G key combination is a shorthand for the `list` command, and if you type only the first letters of a file or function, pressing TAB will autocomplete the name.

The standard keys for scrolling through the source are available (Arrow Up/Down, Page Up/Down, Ctrl+Home and Ctrl+End).

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

`find` *`text`*	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 Ctrl+F inserts the `find` command on the edit line.
`find`	Repeats the last search. Function key F3 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 exceptions 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 29) can also be used on the console line. The command line allows you to set temporary breakpoints and watchpoints as well.

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 31). 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 42 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.

Note that while the `bmdebug` front-end supports both Manchester encoding and asynchronous encoding, the hardware implementation of the debug probe determines which of the two you can use. The original Black Magic Probe only supports Manchester encoding; some derivatives support asynchronous encoding.

`trace` *`clock bitrate`*
`trace passive`
	Enable tracing in Manchester encoding.
	
	If the clock of the target micro-controller and bit rate are set, the `bmdebug` front-end configures the target for SWO tracing. The `clock` and `bitrate` parameters may have a MHz or kHz suffix. For example, the clock may be specified as either 12mhz or 12000000. The `bitrate` parameter may also use the “kbps” unit.
	
	If “`passive`” is set as the command parameter, SWO tracing is turned on in the Black Magic Probe, but the target is not configured. The firmware of the target must itself configure SWO tracing. The parameter “`passive`” may also be written as “`pasv`”.

`trace async` *`clock bitrate`*
`trace async passive` *`bitrate`*
	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.
	
	If “`passive`” or “`pasv`” is set as the command parameter, SWO tracing is turned on in the Black Magic Probe, but the target is not configured (see also the preceding command). In the case of asynchronous encoding, the bit rate must still be set for passive mode.

`trace disable`	Disables SWO tracing.
`trace enable`	Enables SWO tracing using previously stored settings.

`trace 8-bit`
`trace 16-bit`
`trace 32-bit`
`trace auto`
	Sets the width of the data in an SWO tracing packet (in relation to leading-zero compression). This value must match the value that the target uses. The ubiquitous implementation is 8-bit data widths (which is the default setting).
	
	When the parameter is auto, the debugger derives the data width from the incoming data.
	
	See page 48 for more information.

`trace` *`filename`*	Set the metadata file for decoding the Common Trace

	Format (see page 55). When no file is explicitly set, 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.

`trace channel` *`index`* `enable`
`trace chan` *`index`* `enable`
`trace ch` *`index`* `enable`
	Enables display of the given channel (range 0..31).

`trace channel` *`index`* `disable`
`trace chan` *`index`* `disable`
`trace ch` *`index`* `disable`
	Disables display of the given channel.

`trace channel` *`index name`*
`trace chan` *`index name`*
`trace ch` *`index name`*
	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` *`index #colour`*
`trace chan` *`index #colour`*
`trace ch` *`index #colour`*
	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 info`	Show the current configuration and all active channels.

The `bmdebug` front-end saves target-specific settings, such as the settings for SWO tracing in a file with the same name as the target ELF file, but with the added file extension "`.bmcfg`". The settings of this file are reloaded when you load the ELF file again in `bmdebug`. Therefore, to enable SWO tracing and restore all settings and channel configurations from a previous session, the following command is sufficient:

```
trace enable
```

Edit-Compile-Debug Cycle

While stepping through code or analysing trace output, you may spot something that needs to be fixed. However, you do not need to leave the debugger to edit and re-compile the code. It is recommended that you switch to your editor or IDE and rebuild it, and then reload it in GDB. This way, breakpoints and other settings are preserved. The code still restarts at `main`, though.

With the `bmdebug` front-end, the recommended way to reload the ELF file is to use the button "`reset`" at the top left of the source view, or function key `F2`. This buttons not only reloads the file in GDB, it also downloads the file into the target (provided that the “Download to target on mismatch” option is toggled on (in the “Configuration” section in the sidebar).

The `gdbgui` front-end keeps all source files cached until the “reload file” button is clicked. Likewise, the `bmdebug` front-end loads all source files right after GDB loads the debugging symbols for the ELF file and keeps them in memory. As a result, if you edit a source file, those changes will not appear in `bmdebug` until the ELF file is reloaded (through the “`reset`” button or F2). The rationale for this operation is that it keeps the source code, as presented in `bmdebug` in line with the debugging information in the ELF file. The upshot is that you can edit the source code for a program without hesitation while continuing to debug it. A pitfall with `gdbgui`, though, is that if you re-run the program (which reloads the symbolic information), but forget to reload each source file (with the “reload file” button), the source and the executable are still out of sync.

Debugging Optimized Code

When stepping through the code, the current line may on occasion jump over a few lines and then jump back up later. This is especially the case with optimized code. The reason is that GDB steps sequentially through the machine code, and at each point where it stops, it looks up the line number in the source file that matches the address where it stopped. The GCC compiler may have rearranged the code that it generated, in order to get a more optimal result. While it is common advice to compile with optimizations disabled, GDB is actually very capable to debug optimized code — if you can live with an occasional surprising order of execution.

Another optimization that the GCC compiler may perform is to inline small functions. Whether it has done so, is not immediately obvious, because GDB is smart enough to simulate a call to the inlined function when stepping through the code. That is, you can step into an inlined function, even though there isn’t a call in the machine code. What you cannot do, however, is place a breakpoint on the inlined function: the function does not exist as a separate block of instructions. Instead, you must place the breakpoint at the point (or points) where the inlined function is called.

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 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.

Levels of Tracing

The ARM CoreSight architecture has hardware support for both low-level tracing and high-level tracing. Specifically, the Cortex micro-controllers provide for three trace sources:

- *Instruction trace*, which creates a log of every instruction executed by the micro-controller. It is generated by the *Embedded Trace Macrocell* (ETM).
- *Data trace*, to monitor changes of variables or memory. It is generated by the *Data Watchpoint & Trace* (DWT).
- *Software trace*, or “debug message”, which sends out *printf* or *transmit* statements that are embedded in the source code of the firmware. Software trace is

also called instrumented trace, because it requires the firmware to be "instrumented" with trace instructions.

The tracing techniques in this chapter mostly fall in the last category: software trace. The exception, in a way, is Tracing with Command List on Breakpoints (see page 53), because it does not require instrumenting the source code.

The main drawback of code instrumentation is that it makes the firmware 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.

Secondary UART

The Black Magic Probe combines the gdbserver interface with a TTL UART interface (on the same USB connection). If the target board as 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 benefit of the UART is that it only a *single* pin —configuring RxD 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.

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 `TRACESWO` pin branched out (see page 47 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 errors 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 35).

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/M0+ micro-controller architecture, the CoreDebug registers are only accessible from the JTAG/SWD interface, however, 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.

The HardFault handler approach for run-time debugger detection works on all Cortex architectures, it is not restricted to Cortex M0/M0+. On projects build with

CMSIS and libopencm3, a user-defined exception handler automatically replaces the default implementation, provide that it has the correct name. For CMSIS, it is `HardFault_Handler()`, for libopencm3 it is `hard_fault_handler()`.

```
__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.

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 15). The ARM Cortex M0 and M0+ architectures lack 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++);
}
```

Apart from being limited to channel 0, the above function is also inefficient. With tracing disabled, the function still runs over all characters in the message and calls a function. Moreover, as explained in section TRACESWO Protocol (page 7), this protocol transmits *packets* of 1 to 4 bytes and it prefixes each packet with a header byte. With the CMSIS implementation of `ITM_SendChar()`, each packet has a payload of only a single byte. As a result, the effective transfer speed of SWO tracing has just been halved (sending one byte now sends two: a header byte and a payload byte).

A more flexible and efficient 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. If that drops through, it collects up to 4 characters from the message into a packet. The packet header byte that the *Instrumention Trace Macrocell* (ITM) prefixes to it, now accounts for 20% of overhead, rather than 50%.

The ITM has a FIFO to hold packets, that you access via the single register `PORT`. Before storing every next packet in the FIFO for the trace subsystem, it waits in a while loop until the FIFO has space to hold the packet.

```
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 */
  {
    /* collect and transmit characters in packets of 4 bytes */
    uint32_t value = 0, shift = 0;
    while (*msg != '\0') {
      value |= (uint32_t)*msg++ << shift;
      shift += 8;
      if (shift >= 32) {
        while (ITM->PORT[channel].u32 == 0UL)
          __NOP();
        ITM->PORT[channel].u32 = value;
        value = shift = 0;
      }
    }
    /* transmit last collected characters */
    if (shift > 0) {
      while (ITM->PORT[channel].u32 == 0UL)
        __NOP();
      ITM->PORT[channel].u32 = value;
    }
  }
}
```

One complication is the leading-zero compression used by the SWO Trace protocol (again see section TRACESWO Protocol on page 7). When the data stream contains a packet with a zero byte as payload, there is no automatic way to know whether that zero should possibly be expanded to a 16-bit or 32-bit value. Text messages do not contain zero bytes, so that is our escape here, but the above becomes relevant in chapter The Common Trace Format (page 55), which uses a binary stream.

SWO Tracing must first be configured in the micro-controller, which can be done either in the firmware code on the micro-controller, or via the debugprobe. Joseph Yiu, author of *The Definitive Guide to ARM Cortex-M3 Processors*, argues 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 *device-specific* configuration that is 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 below. Once set, it remains enabled (there is no way to disable the capture of SWO tracing in the Black Magic Probe, except for unplugging and re-plugging it).

```
(gdb) 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 original “native” 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 200 kb/s.

Some implementations of the Black Magic Probe firmware on other hardware instead support asynchronous encoding, which allows for higher bit rates, but in this case, the bit rate must be set on the `traceswo` command.

```
(gdb) monitor traceswo 2250000
```

The target must be able to configure the same bit rate, within an error margin of 3%. Also note that the debug probe may have additional limits on the supported bit rates. For example, on Black Magic Probe clones that use the STM32F10x micro-controller, the bit rate must be 4.5 Mb/s divided by an integer value, and with a maximum of 2.25 Mb/s.[1]

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 *Instrumention 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.

1 With the a PLL configured at 72 MHz, the USART of the STM32F10x is limited to 4.5 Mb/s. However, the USB peripheral of the STM32F10x overflows at a continuous data stream of 4.5 Mb/s, which is why the “traceswo” bit rate is limited to half that rate.

```
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).

STM32F10x series

```
void trace_init_STM32F10x(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 */
}
```

STM32F4xx series[1]

```
void trace_init_STM32F4xx(void)
{
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN; /* enable GPIOB clock */
  GPIOB->MODER = (GPIOB->MODER & ~0x000000c0) | 0x00000080; /* alternate func
                                                               for PB3 */
  GPIOB->AFR[0] &= ~0x0000f000;        /* set AF0 (==TRACESWO) on PB3 */
  GPIOB->OSPEEDR |= 0x000000c0;        /* set max speed on PB3 */
  GPIOB->PUPDR &= ~0x000000c0;         /* no pull-up or pull-down on PB3 */
  DBGMCU->CR |= DBGMCU_CR_TRACE_IOEN;  /* enable IO trace pins */
}
```

1 Adapted from the GDB scripts of the Orbuculum project.

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

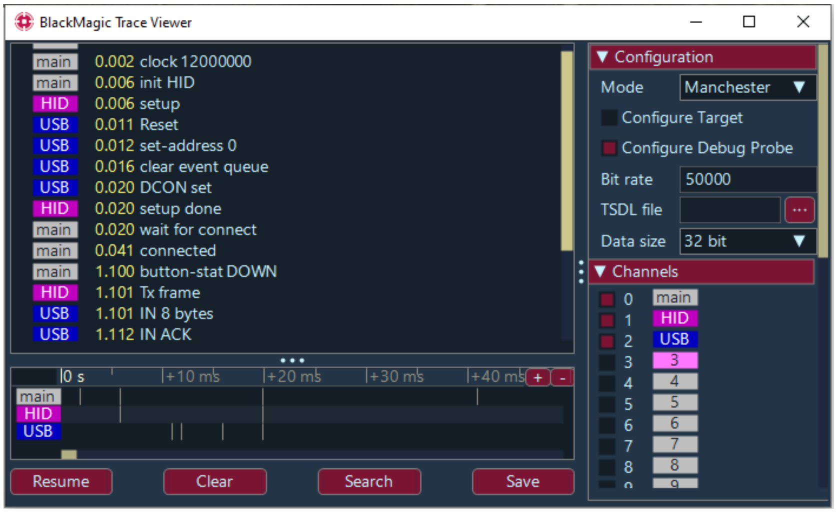
```
void trace_init_LPC15xx(void)
{
  LPC_SYSCTL->TRACECLKDIV = 1;
  LPC_SYSCTL->SYSAHBCLKCTRLSET = 1 << 13;
  LPC_IOCON->PIO0_15 = 0x82;
}
```

Monitoring Trace Data

For capturing the trace data, the Orbuculum project was already mentioned. The main program, `orbuculum`, does the hardware capture and provides the data (after some internal processing) onto a TCP/IP port. Other utilities in the project connect to this TCP/IP port for post-processing and visualization. This client-server architecture allows multiple tools or viewers to access the trace data simultaneously. The packet data that the `orbuculum` server makes available on the TCP/IP port has the same format as that of the Segger J-Link probe, thereby allowing you to use the Segger software tools with the Black Magic Probe. At the time of this writing, Orbuculum runs on Linux and MacOS, and a Windows port is under development.

A stand-alone graphical trace viewer for SWO tracing using the Black Magic Probe is `bmtrace`: the *BlackMagic Trace Viewer* (a companion tool to this guide). It runs under Microsoft Windows and Linux. The `bmtrace` utility does not require GDB, because it uses the *Remote Serial Protocol* (RSP) to configure the target and the Black Magic Probe. The `bmtrace` 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 55.

As described earlier, SWO tracing can use either modes Manchester or Asynchronous, and the configuration section of `bmtrace` allows to choose either (note again that the native Black Magic Probe only supports Manchester mode).



```
main  0.002 clock 12000000
main  0.006 init HID
HID   0.006 setup
USB   0.011 Reset
USB   0.012 set-address 0
USB   0.016 clear event queue
USB   0.020 DCON set
HID   0.020 setup done
main  0.020 wait for connect
main  0.041 connected
main  1.100 button-stat DOWN
HID   1.101 Tx frame
USB   1.101 IN 8 bytes
USB   1.112 IN ACK
```

Mode: Manchester
Configure Target
Configure Debug Probe
Bit rate: 50000
TSDL file:
Data size: 32 bit

0	main
1	HID
2	USB
3	3
4	4
5	5
6	6
7	7
8	8

The `bmtrace` utility optionally configures the target for SWO tracing, and it sets up the Black Magic Probe for SWO tracing as well. For the target configuration, it needs to know the clock that the target micro-controller runs on, as well as the data rate (bit rate) of the transfer.

You can select to skip the target configuration. The target configuration for SWO (both generic and device-specific) then has to be done from GDB, or be performed in the firmware code like in the code snippets starting on page 50. Setting up the Black Magic Probe for SWO tracing can also be disabled. If both these options are disabled, `bmtrace` functions as a "passive listener": it captures SWO trace messages, but does not interact with the target or the Black Magic Probe and does *not* connect to the serial port of Black Magic Probe's gdbserver. The "passive listener" mode allows you to use `bmtrace` in combination with GDB (which then connects to gdbserver).

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 55), 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 29). 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 interrupts the code flow briefly, but lets it continue to run. However, information that the breakpoint was hit appears 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 values of specific variables, or a `backtrace` command to show the call stack that lead to the breakpoint being reached.

The Common Trace Format

As explained in the chapter on Run-Time Tracing (page 42), 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 the current hammer won’t do. 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.

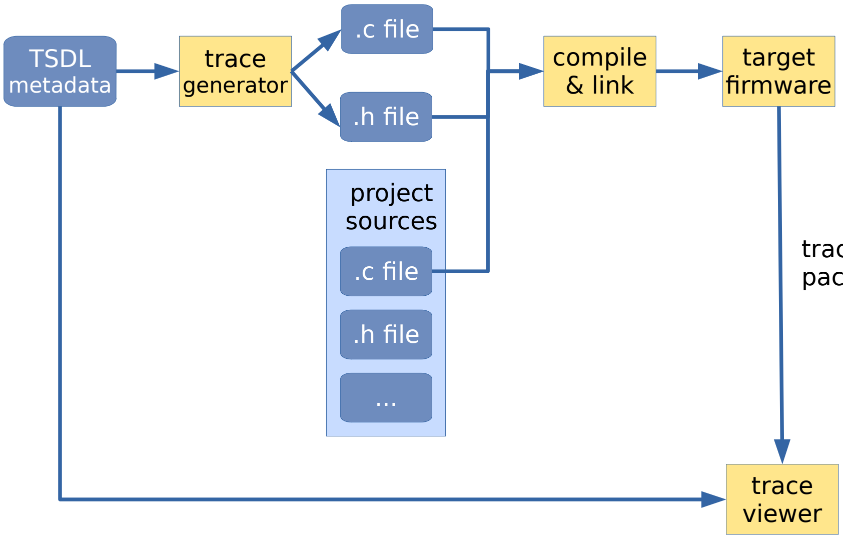
The metadata file defines the names of trace “events”, the streams that these events belong to and the names and types of any parameters of each event. This is all recorded in a declarative language with a C-like syntax: the *Trace Stream Description Language* (TSDL). The metadata is shared (directly 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.

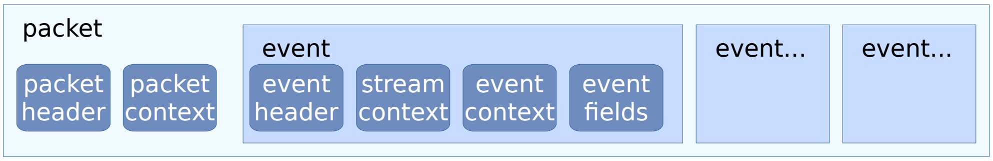
Two tools exist that generate OS-independent C code for CTF support: `barectf` by the same authors as CTF, and `tracegen` (which is a companion tool to this guide). Both tools use the metadata to generate individual C functions to build a binary CTF "packet" for each particular trace event. The generated 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 51) 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.

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
```

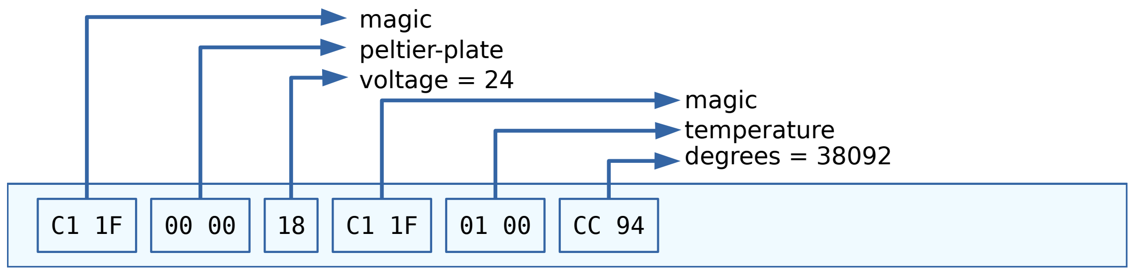
Merely a single byte needs to be transmitted for a descriptive parametrized event, it does not get much more compact than that. However, this is an exceptional case. When there is more than one event, an event header is needed so that the various events can be distinguished. This leads to the need for 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 *clock* 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 autonumbering). 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 58 with the shorthand notations.

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

typealias 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 61 is a file with the name "`peltier.tsdl`", you can run the following command:

```
tracegen -s peltier.tsdl
```

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.tsdl
```

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 48:

```
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 */
  {
    /* collect and transmit characters in packets of 4 bytes */
    uint32_t value = 0, shift = 0;
    while (size-- > 0) {
      value |= (uint32_t)*data++ << shift;
      shift += 8;
      if (shift >= 32) {
        while (ITM->PORT[channel].u32 == 0UL)
          __NOP();
        ITM->PORT[channel].u32 = value;
        value = shift = 0;
      }
    }
    /* transmit last collected characters */
    if (shift > 0) {
      while (ITM->PORT[channel].u32 == 0UL)
        __NOP();
      ITM->PORT[channel].u32 = value;
    }
  }
}
```

The above example assumes 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 60. If the event header does not include a timestamp, there is no need to implement this function (as it will not be called).

Mixing Common Trace Format with Plain Tracing

While the benefit of compactness of Common Trace Format is clear, it adds overhead in the programming effort. Instead of just calling `trace()` with a quick message as a parameter, the programmer now has to spell out the details of the trace message, including any parameters, in a separate TSDL file, and run another tool to create a C file that must be linked with your code. It is more work, and this extra work is worth it for the trace messages that you plan to keep in the code, for regression testing and quality control. For a quick throw-away test, however, this overhead stands in the way.

Fortunately, the two approaches can be mixed when using SWO tracing. A CTF trace message belongs to a stream, which maps to a channel (or *stimulus port*) of the ITM (*Instrumention Trace Macrocell*), see SWO Tracing on page 47. The trace viewer `bmtrace` (and the trace view in the `bmdebug` front-end) use the criterion that if a packet is received on a channel that is present in the TSDL file as a stream, that packet is decoded as CTF. Otherwise, the packet is assumed to contain plain text.

Hence, it suffices to reserve a channel for non-CTF (plain text) trace packets. A channel which you never use in TSDL files for stream IDs. Channel 30 is a pragmatic choice, because channel 31 is regularly reserved by an RTOS for tracing and profiling and auto-numbering of stream IDs by `tracegen` or `barectf` starts at 0.

Applications for Run-Time Tracing

When it comes to where and how to use run-time tracing, the application that immediately springs to mind is to print out the program state, or the value of variables, at specific places in the code. This is the embedded equivalent of "printf-style" debugging. Run-time tracing has a wider scope than this, however.

Code Assertions

The function of an assertion is to display an error message and abort the program when its parameter evaluates to *false*. The goal of an assertion, however, is to always sit silent, because if it *fails* (and prints the error message), there is a bug in your code.

Without going into details (see the book *Writing Solid Code* by Steve Maguire[1] for that), note that assertions should therefore *not* replace error checking. You put assertions in your code to test things that you *know* must be true, if the code was called with the correct input parameters, of which you *know* that these were checked by the caller. The answer to the question why on earth you would test what you already know, is that you may not know what you *think* you know.

In desktop software, the use of assertions is mainstream, because their use is straightforward, their presence declares pre-conditions, post-conditions and invariants in the code (as an informal expression of the formal specification), and it combines well with unit testing. In embedded development, assertions are less commonplace, and the reason (or at least one of the reasons) is that embedded systems lack a universal console (display) to print the "assertion failed" messages to.

Run-time tracing offers an alternative to the console. Of the methods described in chapter Run-Time Tracing on page 42, semihosting has the advantages that it is always available when running under a debugger and it requires no additional set-up in the debugger or debug probe. The relative low performance of semihosting is not an issue: an assertion only transmit output when it fails — when there is a bug.

There are a few pitfalls in the use of assertions. The most important one is that the assertions should not have a side effect. Changing a variable inside the expression of an assertion is right out of the question, but C functions with side effects, like

1 Maguire, Steve; *Writing Solid Code*; Microsoft Press, 1993; ISBN 978-1556155512; or the second edition by Greyden Press, LLC, 2013; ISBN 978-1570740558.

strtok() should be avoided inside an assertion either. Apart from that, lengthy operations carry a risk as well, especially in time-sensitive or performance-critical code. Ideally, an assertion should take negligible time (and resources) for testing its condition.

Assertions grow the code size; especially the default implementation of the assert macro grows the code because it adds the expression and the filename that the macro occurs in as strings to the code. For desktop programs, this is a minor issue, because desktop workstations and laptops have ample memory, but embedded systems are regularly quite constrained. The solution is to re-implement the assert macro to be more economical with code space.

A first step is to eliminate the expression as a string. The filename and line number are sufficient to locate the expression that caused the "assertion failed" notification; duplicating the expression that failed in that notification is redundant. Speaking of filenames, each time you add another assert() in a source file, the filename is stored as a string literal. You will want to merge these duplicate strings, so that only a single copy is stored and all assert macros reference that single copy. The GCC option to do this is -fmerge-all-constants.

The filenames can also be eliminated altogether, by printing the *address* where the assertion failed instead of the filename and line number. An example implementation of an assert macro with minimal overhead is below. This macro implements assert as a statement, as opposed to the standard C library that implements it as a conditional *expression*. The rationale is that this allows the GCC compiler catch unintentional assignments in the condition; the standard implementation of assert stays silent when you write "assert(var = 0)", even though an assignment inside an assert is always wrong. The "if" statement has both *then* and *else* parts (with the *then* part as an empty statement) in order to avoid a dangling-else problem.

```
#define assert(condition) \
    if (condition) \
        {} \
    else \
        assert_fail()
```

The core of the functionality of the assert is implemented in the assert_fail() function. There is only a single implementation of this function, whilst there are potentially many calls to it, through the assert macro sprinkled throughout your code. Therefore, it saves code space to let the assert_fail() function determine the address of the assertion failure, rather than doing it in each macro invocation and passing it as a parameter to assert_fail().

```
__attribute__ ((always_inline)) static inline uint32_t __get_LR(void)
{
  register uint32_t result;
  __asm__ volatile ("mov %0, lr\n" : "=r" (result));
  return result;
}

static void addr_to_string(uint32_t addr, char* str)
{
    int i = sizeof(addr) * 2;   /* always do 8 digits for a 32-bit value */
    str[i] = '\0';
    while (i > 0) {
        int digit = addr & 0x0f;
        str[--i] = (digit > 9) ? digit + ('a' - 10) : digit + '0';
        addr >>= 4;
    }
}

__attribute__ ((weak)) void assert_abort(void)
{
  __BKPT(0);
}

void assert_fail(void)
{
  register uint32_t addr = (__get_LR() & ~1) - 4;
  char buffer[] = "Assertion failed at *0x00000000\n";
  addr_to_string(addr, buffer + 23);
  trace(buffer);
  assert_abort();
}
```

The above snippet implements four functions, the last of which is `assert_fail()`. The first thing this function does is to get the value of the *Link Register*, which holds the address that `assert_fail()` returns to (or that it would return to). That address points behind the call to the function, which is why the size of one instruction is subtracted from it. The lowest bit is also cleared, because that bit is a flag for the ARM Cortex *Thumb mode*. This address is then converted to ASCII and sent out as a trace message.

The last action of `assert_fail()` is to call `assert_abort()`. The default implementation is a software breakpoint, but the purpose of `assert_abort()` is to reset all peripherals to a safe state. If the assertion is inside embedded code for a 3D printer, for example, `assert_abort()` turns all movement and shuts heating off. Because of the “weak” linkage attribute on the default implementation of `assert_abort()`, it is overruled by a user-defined function with the same name.

While on the subject, `__BKPT()` is a CMSIS macro. Other micro-controller support libraries will likely have a similar function for software breakpoints. Otherwise, a simple implementation for GCC is:

```
#define __BKPT(value)    __asm__ volatile ("bkpt "#value)
```

The `trace()` function in `assert_abort()` is a placeholder; it should be replaced by a function that does the actual output of the strings, by the method of your choosing. Unless that function happens to be called `trace()` and has only the string to transmit as a parameter, you need to adjust the code accordingly.

The last step is to look up the filename and line number for an address, with help of symbolic information. When the code is loaded in GDB, this information can be obtained with the `info` command. Note that the asterisk is necessary.

```
(gdb) info line *0x08000505
```

On the command line, you can use the utility `addr2line` to get the filename and line number from an address (on a typical toolchain for the ARM Cortex, the full name may be `arm-none-eabi-addr2line`).

```
arm-none-eabi-addr2line -e blinky.elf 0x08000505
d:\Tools\blinky\blinky.c:168
```

The `bmdebug` front-end (page 35) automatically looks up file and line information for messages that are printed through semihosting, when those messages contain an address as a hexadecimal number and with an asterisk in front. In particular, when the embedded host writes the following via semihosting:

```
Assertion failed at *0x08000505
```

The `bmdebug` front-end will display it in its semihosting view as:

```
Assertion failed at blinky,c:168
```

Tracing Function Entry & Exit

When your code runs in a debugger and halts at a breakpoint, quite often one of the things you want to find out is how you got there: the `backtrace` command is for that purpose. However, when a trace message pops up, the code doesn’t halt and you don’t have the of the context of the message.

The solution is to trace all entries to all functions, as well as exits from them. The GCC compiler has a command-line option to instrument function entries and exits with a call to functions that you must implement.

The GCC option for function-level instrumentation is `-finstrument-functions`. This option inserts a call to an "entry" function at each start of a function, and another call to an "exit" function just before the return. A template for these functions is below:

```
__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *this_fn, void *call_site)
{
  /* ... */
}

__attribute__((no_instrument_function))
void __cyg_profile_func_exit(void *this_fn, void *call_site)
{
  /* ... */
}
```

The first parameter of both functions is the address of the function; the second the address of the function that made the call. Both these addresses can be looked up with the symbolic information, as was addressed in the previous section on the `assert` macro.

To avoid a function from being instrumented, you set the `no_instrument_function` attribute on it. This attribute is required on the entry and exit functions themselves, to avoid unbounded recursion. The same applies to any function called from the entry and exit functions. In addition to the attribute specifications in the source code, GCC also has command line options to block instrumentation for specific functions or for all functions in specific files, look for `-finstrument-functions-exclude-file-list` and `-finstrument-functions-exclude-function-list`.

The implementation of the entry and exit functions will typically be a call to a function that outputs a trace message. In this particular case, low overhead is of the essence, and therefore it is particularly suited for the Common Trace Format (see page 55). If we ignore the `call_site` parameter (which is technically redundant, because you will have received an "entry" message for that caller too), an example implementation for the metadata for the entry and exit functions is in the snippet below. Note that this is just a part of a TSDL file, lacking the definitions of the event header and of optional streams.

```
typealias integer {
    size = 32;
    signed = false;
    base = symaddress;
} := code_address;

event "Enter function" {
    fields := struct {
        code_address addr;
    };
};

event "Leave function" {
    fields := struct {
        code_address addr;
    };
};
```

The main feature of the above code is the definition of the `code_address` type, and especially the declaration “`base = symaddress`” (`symaddress` may be abbreviated to `symaddr`). This declaration signals that any parameter with this type is a symbol address. This signals the trace viewer to look the address up in the symbolic information.

Currently, the `bmdebug` and `bmtrace` utilities print the function name instead of an address, when the “base” for the respective parameter is set to `symaddress`.

The functions generated by `tracegen` for these TSDL events can now be called from the `__cyg_profile_func_enter` and `__cyg_profile_func_exit` functions. Note that you will probably want to add the option `-no-instr` option on the `tracegen` command line, so that it adds the `no_instrument_function` attribute to all generated functions.

```
tracegen -s -t -no-instr blinky.tsdl
```

Firmware Programming

As show in chapter Debugging Code on page 22, 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 “\”, see the second snippet below). 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
```

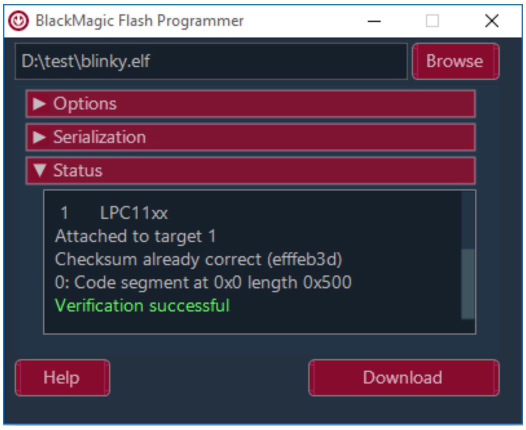
You need to change `COM9` to the serial device that is appropriate for your system, and `blinky.elf` to the appropriate filename. In Linux, you may use the `bmscan` utility to automatically fill in the device name for the gdbserver virtual serial port:

```
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 26.

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.



```
1     LPC11xx
Attached to target 1
Checksum already correct (efffeb3d)
0: Code segment at 0x0 length 0x500
Verification successful
```

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 micro-controller series from NXP (see page 26).

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 "`.text`" or "`.rodata`". If the section name is empty, the offset is from the beginning of the ELF file.
Match	In this mode, the `bmflash` 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. The "match" string can be an ASCII string, like "`$serial$`". It can also contain binary values, which you specify with \ddd or \x*hh* where *ddd* is

	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 "`.bmcfg`" added to it.

The `bmflash` utility currently cannot rewrite option bytes (on micro-controllers that use them). This has the implication that `bmflash` cannot re-program STM32Fxx micro-controllers that have code protection set. See the section Reset Code Protection at page 25 to clear the option bytes (and thereby disable code protection). On LPC micro-controllers (from NXP), `bmflash` can clear code protection if you enable the option to fully erase all Flash memory before downloading the new firmware code.

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 79.

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 Error: Reference source not found. 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 79 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).

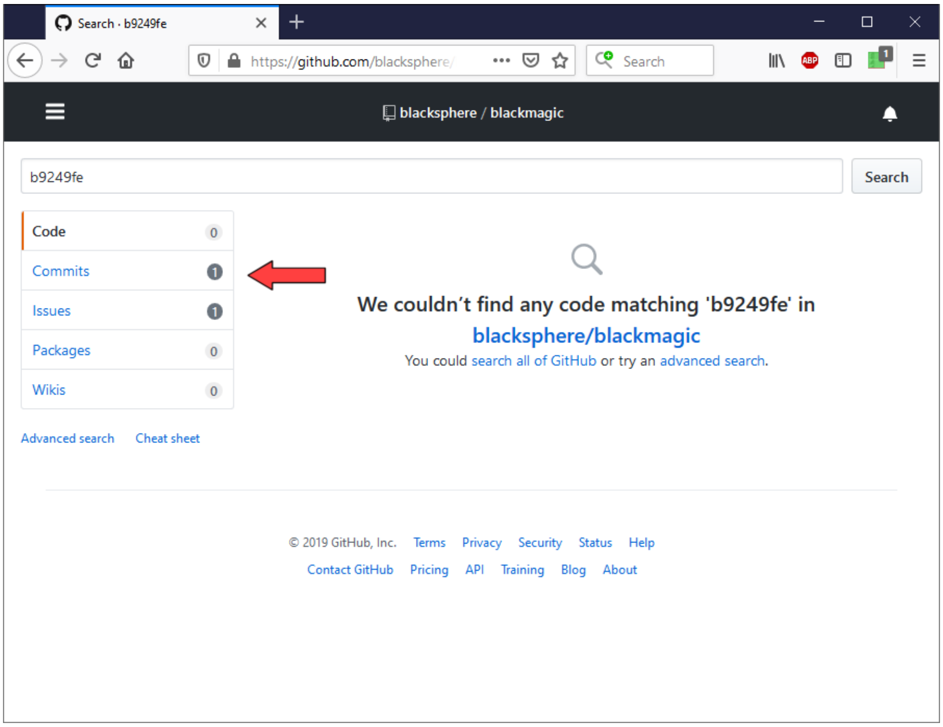
You can check which firmware version you have with the GDB `monitor` command (after connecting it as an extended-remote target).

```
(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`”, where the hexadecimal number fbf1963 is the crux. More recent releases of the firmware use a longer description, where the GitHub hash follows the letter “g” (e7e3460 in the example below)

```
(gdb) monitor version
Black Magic Probe (Firmware v1.6.1-379-ge7e3460) (Hardware Version 3)
Copyright (C) 2015  Black Sphere Technologies Ltd.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

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” (see arrow in the picture below) you will get a summary of the relevant commit, plus the date of that commit.



We couldn’t find any code matching 'b9249fe' in blacksphere/blackmagic

You could search all of GitHub or try an advanced search.

Micro-Controller Driver Support

Micro-controllers must frequently need some configuration to set up specific GDB functions or SWO tracing. For example, the section Flash Memory Remap on page 24 addressed a step that is needed before you can download code into the micro-controllers of the LPC families from NXP. In that section, we also recommended to define a command for that step in to .gdbinit file.

The utilities bmflash and bmdebug run MCU-specific scripts to remap memory, and the utilities bmtrace and bmdebug also run MCU-specific scripts to configure SWO tracing. These utilities contain the scripts embedded in the executable, and they establish which script to run by evaluating the name of the MCU driver that the Black Magic Probe returns on attaching to it. However, the Black Magic Probe is continuously enhanced and extended, and micro-controller support is growing. To that end, the predefined hard-coded scripts can be extended or overruled.

Script definitions for new (or modified) scripts must be stored in a file with the name "bmscript" (no file extension). On Microsoft Windows, this script must be stored in the "BlackMagic" directory in the (roaming) "Application Data" folder. The "INI" files for the diverse utilities are stored here as well. On Linux, the bmscript file must be stored in the ".local/share/BlackMagic" directory below the home directory of the current user.

The syntax of the definitions in the bmscript file is similar to that of .gdbinit, but it is not compatible with it. Only define statements can occur in bmscript, and these define statements must conform to either a register definition, or a script definition.

```
define SYSCON_SYSMEMREMAP [ lpc8xx, lpc11xx, lpc12xx, lpc13xx ] = {int}0x40048000
define SYSCON_SYSMEMREMAP [ lpc15xx ] = {int}0x40074000
define SCB_MEMMAP [ lpc17xx ] = {int}0x400FC040
define SCB_MEMMAP [ lpc21xx, lpc22xx, lpc23xx, lpc24xx ] = {int}0xE01FC040

define memremap [ lpc8xx, lpc11xx, lpc12xx, lpc13xx ]
    set SYSCON_SYSMEMREMAP = 2
end

define memremap [ lpc15xx ]
    set SYSCON_SYSMEMREMAP = 2
end
```

```
define memremap [ lpc17xx ]
    set SCB_MEMMAP = 1
end

define memremap [ lpc21xx, lpc22xx, lpc23xx, lpc24xx ]
    set SCB_MEMMAP = 1
end
```

As is apparent in the above example, each register and each script has a list of micro-controller driver names between square brackets after the name. These driver names are the names that the Black Magic Probe reports when it scans the attached target. The name may end with an asterisk, for a wildcard. For example, if "`STM32F1*`" appears in this list, it matches `STM32F101T8` as well as `STM32F103C8`.

The list of MCU drivers is a filter for the definition. Because of this filter, there is no conflict to define the same register name or script name twice, provided that there is no overlap in MCU driver names.

The names of the registers may be freely chosen, but the names of the scripts are predefined by the `bmflash`, `bmtrace` and `bmdebug` utilities. The scripts that are currently defined are:

- `memremap`, to make sure that the micro-controller's Flash memory map conforms to the ELF file layout.
- `swo_device`, for the MCU-specific configuration for SWO tracing.
- `swo_generic`, for the configuration for SWO tracing that is common to all ARM Cortex micro-controllers.
- `swo_channels`, to set the mask for enabled channels; this script is also common to all ARM Cortex micro-controllers.

You will typically only add (or replace) the first two of this list, but you can override the generic scripts for a particular micro-controller as well.

The only operations allowed on the registers (within a script) are assignment with "`=`", "`|=`" and "`&=`", which function in the same way as in GDB (and the C language). The values at the right hand side may use decimal and hexadecimal notation, a "`~`" may prefix the value to denote the bitwise inversion of the value.

Further Information

Hardware

Black Magic Probe: The official Black Magic Probe hardware is available from:

1BitSquared	http://1bitsquared.de/products/black-magic-probe
adafruit	https://www.adafruit.com/product/3839
elektor	https://www.elektor.com/

3D Printed Enclosures for the Black Magic Probe can be found on Thingiverse. A simple clip that offers some protection for the 10-pin Cortex Debug header (see page 15) is “thing” 2387688 (by Michael McAvoy); a full enclosure with openings for the connectors, LEDs and button is “thing” 2836934 (by Emil Fresk).
https://www.thingiverse.com/

tag-connect: cables with a pogo-pin plug, specifically suited for firmware programming and debugging. The cable suitable for the ARM Cortex SWD interface are TC2030-CTX and TC2030-CTX-NL. See also Connecting the Target on page 15.
https://www.tag-connect.com/

Software

Black Magic Probe: The GitHub project for the Black Magic Probe holds the firmware, documentation and schematics.
https://github.com/blacksphere/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.blacksphere.co.nz/blackmagic/

Notes on building the firmware are in the wiki of this GitHub project. However, these notes are Linux-centric. For building on Microsoft Windows, see the additional notes on Sid Price’s blog, specifically:
http://www.sidprice.com/2018/05/23/cortex-m-debugging-probe/

Zadig: A utility for installing the drivers for SWO tracing and firmware update, see chapter Setting up Black Magic Probe on page 11.
https://zadig.akeo.ie/

libusbK: A project with drivers, support DLLs and development files for generic USB device access.

http://libusbk.sourceforge.net/UsbK3/

gdbgui: Various GDB front-ends were mentioned in chapter Requirements for Front-ends (page 9), 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 (SWO tracing, exception trace, performance profiling, …), see section SWO Tracing on page 47.

https://github.com/orbcode/orbuculum

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

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

Articles, Books, Specifications

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 55.

https://diamon.org/ctf/

The Art of Debugging with GDB, DDD, and Eclipse; Norman Matloff & Peter Jay Salzman; No Starch Press, 2008; ISBN 978-1593271749.

Writing Solid Code, second edition; Steve Maguire; Greyden Press, LLC, 2013; ISBN 978-1570740558.

The Definitive Guide to the ARM Cortex-M3, second edition; Joseph Yiu; Newnes Press, 2009; ISBN 978-1856179638.

Index