The officially “not recommended” resource on the Black Magic Probe

Embedded Debugging with the Black Magic Probe

Thiadmer Riemersma

2024

This book is available in PDF format on the CompuPhase website and on GitHub, along with utilities and coding examples. See the chapter Further Information for a link.

The cover image is by Arek Socha.

ISBN 978-1-4478-3925-5

Typeset with TEX in the “Adobe Source” typeface family.

Contents

Introduction 1
- Hardware and Software 1
- Why bother, why choose the difficult route? 2
- About this Book 4
- License 5

The Debugging Pipeline 6
- GDB Architecture 6
- The Serial Wire Debug Protocol in a Nutshell 8
- Embedded Debugging: Points for Attention 11
- Requirements for Front-ends 13

Hardware Overview 14
- Accessories 16

Setting up the Black Magic Probe 20
- Microsoft Windows (USB set-up) 20
- Linux (USB set-up) 22
- Wi-Fi set-up for ctxLink 24
- Connecting the Target 26
- Checking the Set-up 27
- Running Commands on Start-up 28
- Design for Debugging 30

Debugging Code 32
- Prerequisite Steps 33
- Loading a File and Downloading it to the Target 33
- Starting to Run Code 38
- Getting help and information 39
- Listing Source Code 39
- Downloading code into the microcontroller 40
- Stepping and Running 41
- Breakpoints and watchpoints 43
- Examining Variables and Memory 45
- The Call Stack 48
- Inspecting Machine Code 48
- Debug Probe Commands 49
- Edit-Compile-Debug Cycle 55
- Debugging Optimized Code 57
- Semihosting: target to host I/O bridge 57
- The BlackMagic Debugger Front-end 62

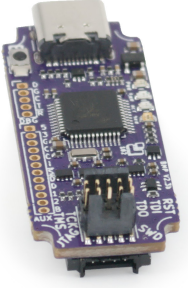
Run-Time Tracing 74
Levels of Tracing 74
Secondary UART 75
Semihosting 76
SWO Tracing 76
Real Time Transfer (RTT) 87
Tracing with Command List on Breakpoints 90
The Common Trace Format 92
Binary Packet Format 94
A Synopsis of TSDL 94
Generating Trace Support Files 105
Integrating Tracing in your Source Code 107
Mixing Common Trace Format with Plain Tracing 109
Applications for Run-Time Tracing 110
Code Assertions 110
Tracing Function Entry and Exit 113
Code Profiling 117
Sampling on ARM Cortex 117
Calltree Analysis 120
Firmware Programming 122
Using GDB 122
Using the BlackMagic Flash Programmer 124
Updating Black Magic Probe Firmware 133
Building from Source 135
Troubleshooting 138
Check whether the system detects the probe 138
Check whether the probe detects the target 139
Target scan hangs 143
GDB crashes on “attach” 144
Attach regularly fails 145
Failure to erase Flash memory 145
Spying on the communication 145
How to Reset the Black Magic Probe 146
TRACESWO Capture 146
RTT capture 147
TTL-Level UART 148
GDB on Microsoft Windows 148
Microcontroller Driver Support 150

Tcl Primer 153
Syntax 153
Flow Control Structures 156
Numbers 158
Lists and Strings 159
Variables and Arrays 159
Expressions 160
User procedures 161
Comments 163
Exception handling 164
Binary data 164
Built-in commands 166
Further Reading 171
Application-Specific Extensions 171
Unified Connector: Debug + UART 178
Linking TRACESWO to UART-RxD 180
Further Information 182
Hardware 182
Software 183
Articles, Books, Specifications 184
Index 186

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 microcontrollers. At the software level, it provides a "gdbserver" implementation and Flash programmer support for ranges of microcontrollers of various brands. The hardware was designed by 1BitSquared in collaboration with Black Sphere Technologies. The embedded software of the Black Magic Probe is an open-source project (the hardware is less open: schematics are available for legacy versions of the hardware, but not for the current version).

At the time of writing, the Black Magic Probe hardware is version 2.3, and the firmware is at version 1.9.1. The firmware in the Black Magic Probe is in ongoing development, so the firmware in newly purchased probes may be behind the current release, see also Updating Black Magic Probe Firmware on page 133). Derivatives of both hardware and firmware exist, with sometimes different capabilities or limitations. This book focuses on the *native* hardware, and firmware version 1.6 or later —but it includes notes on two commercially available derivatives where applicable: ctxLink and the Jeff Probe.



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 microcontroller. 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 microcontroller 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 microcontrollers, this typically means the GDB from the arm-none-eabi toolchain.[1]

While you do not need a debugger front-end, it is beneficial to get one. When you are running on Linux, you may get by with GDB's integrated Text User Interface —it's rudimentary, though. See Requirements for Front-ends (page 13) for tips to select a front-end.

Why bother, why choose the difficult route?

Advice that I have repeatedly seen on blogs and answers on stackoverflow (and others), is to make the software modular, debug each module on a desktop PC or laptop, and to then assemble the embedded application from these fully tested and debugged modules. The implied argument is that embedded software is fundamentally the same as desktop software, but you have the cream of the crop in development tools on desktop systems.

Allow me to draw a parallel from a different field: From the earliest days of medicine, the focus has been on studying the physiology of *men*. It was assumed that the female body responds to medication and drugs in the same way as the male body. Up to the 1960s, clinical trials for a new drug were done on sometimes thousands of men, and zero women. Women, after all, would supposedly only bring "confounding issues" to the trials, due to their alleged emotional instability and fluctuating hormone levels. It has led to absurdities like a clinical trial in 2015 for Addyi, a drug to treat female sexual dysfunction, involving 23 men and 2 women —a drug exclusively for women, tested almost exclusively on men. All the while, the presumption that the female body is that of a man (though with confounding issues) is entirely unfounded. Sex bias in medicine isn't based on facts, but rather a symptom of complacency and indifference.

[1] With a caveat for GDB versions 11 and later. These releases of GDB have an issue in the handling of the Remote Serial Protocol that causes a failure when attaching to a target microcontroller. See page 144 for details and solutions.

Embedded devices are a varied lot, but as a general rule, they are *not* just a PC with confounding issues. Software that runs fine on a desktop system may fail on the target microcontroller. Not every microcontroller handles unaligned memory access alike, for example. Embedded devices commonly have (integrated) peripherals that desktop systems lack, and on an embedded device, those peripherals will be driven with the SPI or I²C protocols, rather than USB.

The recommendation to develop & debug embedded software on a desktop, on the dogma that it should then run alike on the target device, is similarly based on an invalid assumption and an ill-advised desire to stick with the familiar tools and environment. It will actually work on specific cases, such as a generic data structures library, but it is a bad strategy overall.

In my consulting work, I get occasionally to listen to a "war story" by a fellow developer, about failures, glitches and missed interrupts. "*But in a simulator on a PC it ran flawlessly, so...*" Often, the issue was circumvented rather than solved. For example, to "fix" a case of an occasionally missed interrupt, the developer set an interrupt on both rising and falling edges of a pulse because it hadn't happened yet that the MCU missed two interrupts in succession. Sometimes the hardware was redesigned to use an MCU of a different brand or architecture (but yet, without evidence that the original MCU was at fault). Frequently, the frustration about the wasted time and resources hadn't subsided yet —one developer claimed to have "even switched to Torx" [screws], so much he had come to loathe Philips.[2]

I was not there to debug their systems, so we will never know the truth. The point is, developing code intended to run on an embedded device and testing it exclusively on a desktop system, is as absurd as developing a drug exclusively for women and testing it on men. And while these companies found workarounds, the real point is the wasted time and resources, both of which cost money.

[2] It didn't help me laughingly pointing out that the well known cross-slotted screw head is Phillips —double "*l*" and entirely unrelated to the semiconductor brand Philips (now NXP).

About this Book

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,[3] and which is highly recommended. This guide does not delve into the hardware and software design of the Black Magic Probe, either. The software of the Black Magic Probe is open-source, hosted on GitHub, and information on the hardware is best obtained from 1BitSquared (the manufacturer).

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

The next chapters focus on setting up the hardware and software for the Black Magic Probe, followed by 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.

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, specialized, utility.

The final chapters are on updating the firmware of the Black Magic Probe itself; on troubleshooting tips for when things don't work right; and on reference documentation for a variety of miscellaneous topics. Throughout this book are brief references to specifications, accessories, useful software, and other sources of information. The final chapter collects all these references with the details of where/how to obtain/them.

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

License

The Debugging Pipeline

Developing embedded software on small microcontrollers 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 microcontroller'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

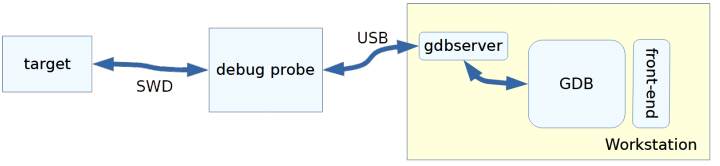
GDB is the GNU Debugger for programs built with GCC. It is also a debugger framework, supporting third-party front-ends and machine/protocol-specific back-ends.

GDB's user interface is, by today's standard, rather rudimentary. Instead, 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 who 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 (it is also broken on Microsoft Windows, and there is no plan to fix it).

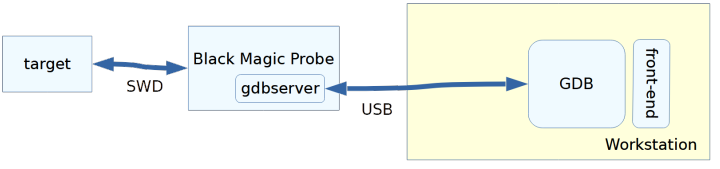
To debug a different system than the one where the debugger runs on, GDB provides the *Remote Serial Protocol* (RSP). This is a simple text-based protocol with which GDB on the workstation communicates with a debugger "stub" on the target system. This stub acts as a server that GDB connects to, over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

Directly implementing a gdbserver on the *target* is impractical for microcontrollers such as the ARM Cortex M series. These microcontrollers 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 interface* (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.[1] The main openocd program implements gdbserver, and it 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 of this design is that its gdbserver has in-depth knowledge of the capabilities of the debug probe as well as what the debug probe has discovered 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 ctxLink debug probe functions identically to the Black Magic Probe when connected to the USB —in fact, it even uses the same `VID:PID` codes. However, ctxLink also offers connection over a Wi-Fi link. In relation to the diagram above, this changes very little: in essence, you only need to change the caption of the "USB" link to "Wi-Fi" (disregarding that there is also a wireless switch or access point thrown in the mix). But the implication is that while the range of a USB-connection is limited, ctxLink makes

[1] In a "hosted" set-up, the Black Magic Probe also uses this configuration: the main firmware of the Black Magic Probe (with the embedded gdbserver) runs as a desktop application on the workstation, and the Black Magic Probe hardware is reduced to function as a dumb probe. See section Check whether the probe detects the target on page 139 for more information about this option.

the debug probe accessible over the local network and (after configuring the router) over the internet. Thereby, ctxLink enables debugging over a technically unlimited distance.

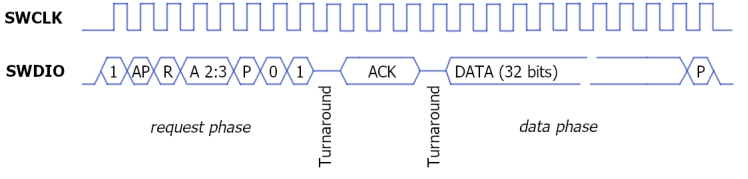
Bypassing GDB

While the *Remote Serial Protocol* (RSP) is specifically designed for GDB (to communicate with gdbserver in a debug probe), it is well-defined and well-documented, and therefore other tools can use it without requiring GDB. In fact, the BMTrace (page 85) and BMFlash (page 124) utilities do exactly this. These utilities use a fairly limited subset of the capabilities of gdbserver.

Troll is a source-level debugger for ARM Cortex architecture using GDB's RSP, and it is thereby compatible with the Black Magic Probe. The Troll debugger is still an experimental project; see Further Information on page 182 for a link to the project.

The Serial Wire Debug Protocol in a Nutshell

The Serial Wire Debug protocol (SWD) is designed as an alternative to the JTAG protocol, for microcontrollers with a low pin count. It is part of the ARM Debug Interface specification version 5, abbreviated as ADI5. At the physical layer, it needs two lines at the minimum (plus ground), as opposed to five for JTAG. These two lines are the clock (SWCLK, driven by the debug probe) and a bidirectional data line (SWDIO). Tracing output goes over a third line: TRACESWO, but using an unrelated protocol (independent of SWCLK) —see section TRACESWO Protocol on page 10.



The SWCLK signal is driven by the debug probe, regardless of the direction of the transfer. Each transaction 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.

The target microcontroller polls the SWDIO pin on a rising edge of SWCLK, and also drives the pin on a rising edge —but respecting a "hold time"

delay of at least 1 ns. When idle, the SWCLK and SWDIO pins are driven low by the debug probe.[2]

As is apparent, the direction of the SWDIO line switches between input and output at least once during a transaction, on both sides. The SWD protocol calls this the *turnaround*. A turnaround takes a single clock cycle, during which neither the debug probe nor the microcontroller drive the SWDIO pin (both tri-state the pin). The pictured example is for a *write* transaction; in a *read* transaction, there is no turnaround after the ACK — however, if another transaction follows head-to-tail, a turnabout is added after the data phase.

The request phase is a sequence of 8 bits. First comes a start bit (always 1). The AP bit is 0 if this transfer is for the Debug Port (DP), and 1 if it is for an Access Point (AP) such as MEM-AP, which provides access to core memory and peripheral registers. 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 parity 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, another parity bit follows (calculated such that the sequence of 33 bits has even parity).

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 microcontroller'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 microcontrollers do not support JTAG, but the protocol requires that the JTAG-to-SWD switch is still performed.

[2] On the Black Magic Probe hardware 2.3 with firmware 1.9 & later, the SWCLK and SWDIO pins are toggled to high-impedance when *detaching* from a target.

TRACESWO Protocol

The TRACESWO protocol is independent of the SWD protocol. You can trace without debugging, as well as debug without tracing. The trace data is transmitted over a single line using 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) respectively. The TRACESWO protocol is handled by the *Instrumentation Trace Macrocell* (ITM) of the ARM Cortex core.

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 microcontroller 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 insusceptible to clock drift. These properties make Manchester encoding the option of choice for microcontrollers 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 transmits sequences of 1 to 8 bytes on the TRACESWO pin. 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 of data (least-significant bit first). After a start bit and up to 64 data 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 (transmitted low byte first). The protocol uses trailing-zero compression on the payload, which means that if only one or two bytes are transmitted, these form the low bytes of the 32-bit word and the high bytes of that word are assumed zero.

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.

Events generated by the *Data Watchpoint & Trace* unit (DWT), that the ITM passes through, use a packet header as well. The three low bits in the header are set to combinations that are invalid for a trace message. Thus, trace monitoring applications can test the three low bits to check whether to process or reject a packet.

Embedded Debugging: Points for Attention

Desktop computers and single-board computers run programs 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 target program. The debugger also quickly puts the original instruction back into RAM, so that when you resume running the target, it will execute the original instruction.

Current microcontrollers often have limited SRAM, but a larger amount of Flash memory. The program for microcontroller projects therefore

typically runs from Flash memory. For the purposes of running code, you may regard Flash memory as ROM; technically, it is re-writable, but re-writing 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 microcontroller and tell the microcontroller 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). However, while the number of software breakpoints is virtually unlimited, ARM Cortex microcontrollers provide only few hardware breakpoints; typical values are below:

Core	Breakpoints	Watchpoints
Cortex M0 / M0+	4	2
Cortex M3 / M4	6	4
Cortex A / R	6	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 behavior 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, where the last command in the list immediately continues 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 sus-

pected and run another session —all without needing to edit and rebuild the code. See section Tracing with Command List on Breakpoints on page 90 for details.

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.

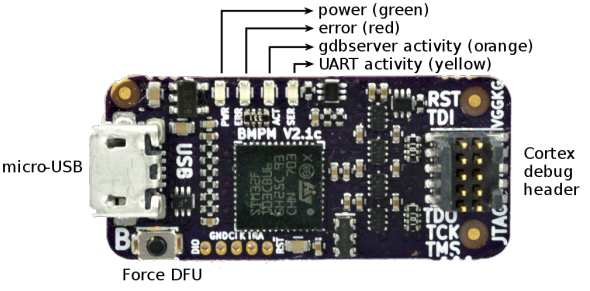
The front-end should not do away with the console, though. Some of the more advanced commands of GDB are not easily represented with icons and menu selections. This is especially true for remote debugging, and even more so for remotely debugging embedded systems. Without the ability to set or read the 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 microcontrollers.

In a misguided attempt to increase "user-friendliness," KDbg, Nemiver and the Eclipse IDE 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 on Linux: DDD & gdbgui work well, and GDB's internal TUI is adequate. The TUI is not available on Windows builds of GDB, and DDD has not been ported to Windows. However, gdbgui runs in a browser, there is a "Cortex Debug" extension for the Visual Studio Code editor, and two (commercial) alternative front-ends for Microsoft Windows are WinGDB[3] and VisualGDB (both function as plugins to Microsoft's Visual Studio). Finally, a few GDB front-ends specifically designed for the Black Magic Probe exist. One of these was developed along with this book, and it is covered extensively in section The BlackMagic Debugger Front-end on page 62. For an alternative, see Further Information on page 182 and specifically the front-end "turbo."

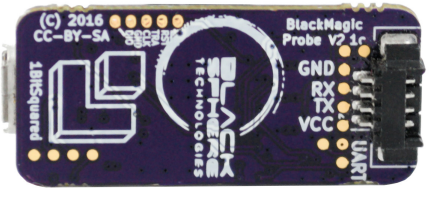
[3] WinGDB appears to have been discontinued as of 2023.

Hardware Overview

There are two versions of the *native* Black Magic Probe hardware in current use. Version 2.1, the Black Magic Probe "mini," was available from 2016 to 2022. It is a 33 × 15 mm PCB, with a micro-USB connector for linking it to a workstation and a 2×5-pins 1.27 mm pitch "debug" header for connection to the target microcontroller. See section Connecting the Target on page 26 for details on the Cortex Debug header.

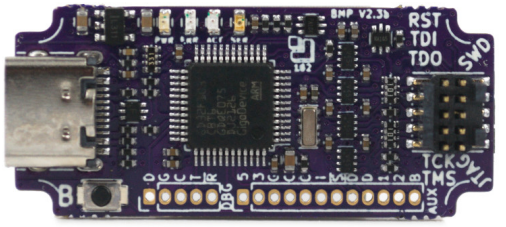


Next to the two connectors, the Black Magic Probe has an on-board switch (that you will only use to upgrade the firmware to the Black Magic Probe, see Updating Black Magic Probe Firmware on page 133) and four LEDs that signal power and activity status.

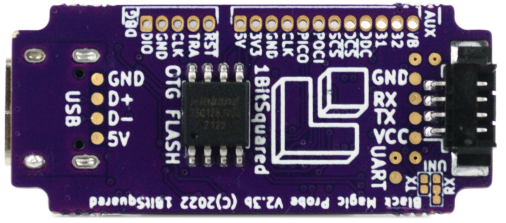


On the reverse site, the Black Magic Probe has a third connector, for a secondary TTL-level UART. This is a 4-pins 1.25 mm pitch "PicoBlade" connector. The function of the four pins is annotated in the silk-screen text on the back. A 145 mm cable with four colored wires and a suitable PicoBlade connector is provided with the Black Magic Probe.

Hardware version 2.3 was introduced in 2023 (after the last batch of version 2.1 sold out), and is the current release of the Black Magic Probe. At 39 × 17 mm, it is slightly larger than its predecessor. The USB connector has been upgraded to USB-C. Other changes to the hardware design, such as the extra Flash memory on the bottom side for "on-the-go" programming, may become more important in future releases of the firmware —but they are currently unused. The LEDs and the on-board switch are the same as on the earlier version, and positioned in the same positions.

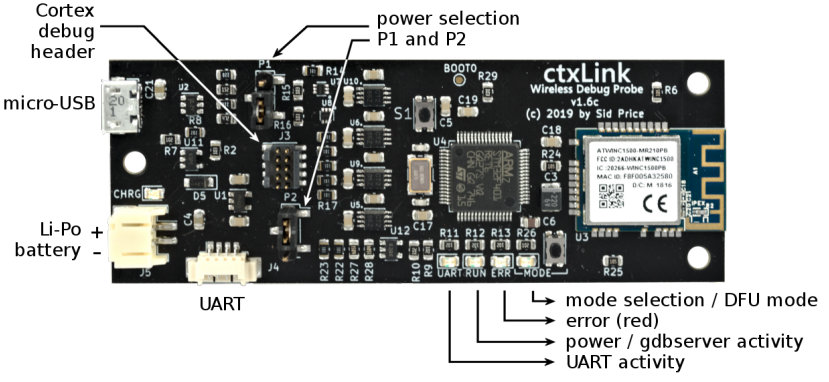


Like version 2.3, there is a PicoBlade connector for a TTL-level UART.



ctxLink

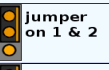
The ctxLink probe is larger than the Black Magic Probe, at 89 × 33 mm. All components and connectors are on the top side — the PCB uses only surface-mount connectors, thus the bottom side is completely flush. The PCB has three 3.2 mm mounting holes.



Like the Black Magic Probe:, the ctxLink probe has a USB port, a Cortex Debug header, and a TTL UART connector (a 4-pins 1.25 mm pitch “PicoBlade” connector). It also has four LEDs; three of them have the same functions as on the Black Magic Probe, the fourth shows the connection mode. In addition to the shared features, the ctxLink probe has a connector for a rechargeable battery (a JST PH-series, 2-pin with a pitch of 2 mm)

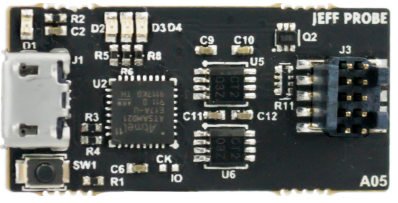
and a Wi-Fi module.

The ctxLink probe can be powered from multiple sources. It uses two jumpers, P1 and P2, for selecting the power source (see the image above for the locations of P1 and P2). These jumpers should be appropriately set before connecting the ctxLink to power.

Power selection	P1 (source)	P2 (voltage)
USB-connection, USB-adapter, or Li-Po battery (3.7 V)		
Powered from Target (5 V)		
Powered from Target (3.3 V)		

Jeff Probe

The Jeff Probe is a low-cost clone of the Black Magic Probe that is nevertheless *mostly* compatible with the original, both in hardware and software. It has the same connectors, at roughly the same positions, and it has the same specifications for target voltage levels.

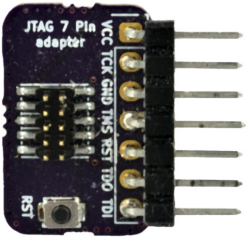


The reason that I describe it as only "mostly" compatible, is that it does not support TRACESWO. The current firmware release, at the time of writing, *does* support the Real Time Transfer (RTT) protocol, as a possible alternative to TRACESWO (see page 87). The Jeff Probes as being sold may contain an older firmware (version 1.6), so in order to take advantage of RTT, you must upgrade the firmware —see Updating Black Magic Probe Firmware on page 133.

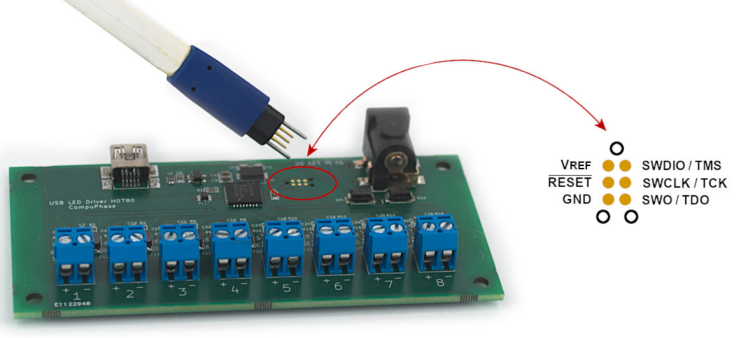
Accessories

The Black Magic Probe has a 2×5-pins 1.27 mm pitch debug header for connection to the target, and if the target board has the same connector, it can be connected with the provided ribbon cable. For the other cases, an adapter board or "break-out" board is needed.

A common adapter board is one where that converts between the 10-pin Cortex Debug header and the 20-pin JTAG header. An example of a break-out board is pictured below; it makes the signals of the Cortex Debug available on a single-row 7-pin header (of the ten pins of the Cortex Debug header, three are ground and one is not-connected, so seven pins cover all signals).



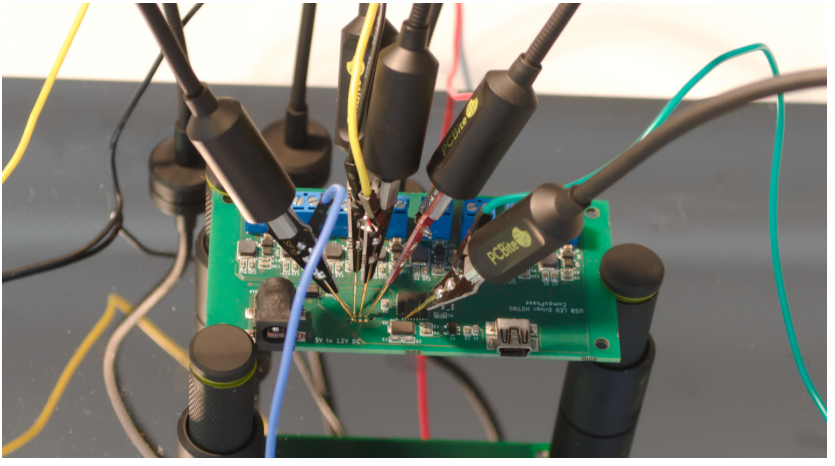
Our favorite debug connector is the decal for the tag-connect cable. This cable has a plug with six pogo-pins, plus three fixed pins that serve to align the plug. The benefit of the tag-connect cable is that it requires less space on the target board than for most other connectors, and that the matching "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.



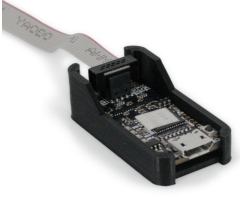
The freeconnect project is an open-source design for a set of pogo-pin connectors that are compatible with the tag-connect cable. See chapter Further Information on page 182 for a link to the project.

We have found a set of needle probes an indispensable accessory for debugging, especially after a mishap. Like downloading code that inadvertently disables the SWD port. To restore access to the target microcontroller, you will then need to put it up in bootloader mode — as described in section Design for Debugging on page 30. With a few needle probes on

the test pads, or directly on microcontroller pins, you can do so conveniently. We have good experience with the PCBite probes by Sensepeek; again, see chapter Further Information on page 182 for a link.



The Black Magic Probe comes without an enclosure, but if you have access to a 3D printer, it is recommended to print one. An enclosure gives electrical insulation (the Black Magic Probe has a series of exposed test pads at the bottom), as well as mechanical protection. Especially the header for the Cortex Debug connector is somewhat fragile. A few printable designs of enclosures are freely available, see chapter Further Information on page 182. It feels fitting to print these enclosures in black, but you are of course free to choose any color.



for hardware version 2.1
by Michael McAvoy

for hardware version 2.1,
by Emil Fresk

designs for 2.1 & 2.3 versions
by author

Likewise, designs for 3D printed enclosures for the ctxLink probe are available on Sid Price's GitHub page. When using ctxLink with a rechargeable battery, an enclosure is recommended, because it protects the battery as well. The ready-to-print STL files are for a Lithium Polymer (Li-Po) "`503562`"-style battery, which stands for 5.0 mm thick, 35 mm wide and 62 mm long. When using a different battery size, you may need to adjust the design of the enclosure; the design files for AutoDesk Fusion 360 are provided.

The Li-Po battery itself is also a useful accessory for the ctxLink probe, especially for those situations where it is cumbersome to pull a (long) cable to the remote target. The ctxLink probe has a 2-pin JST PH-series connector for the battery (2 mm pitch). For the polarity, see the picture at page 15. The ctxLink probe charges the battery with a constant current of 500 mA. Since charging current of Li-Po batteries should not exceed 1C, where C is the capacity in Ampere-hours, the deduction is that a 500 mAh capacity is the minimum to be suitable for ctxLink. For a prolonged lifetime of the battery, it is recommended to charge at 0.5C. Therefore, a 1000 mAh battery (or higher) is recommended.

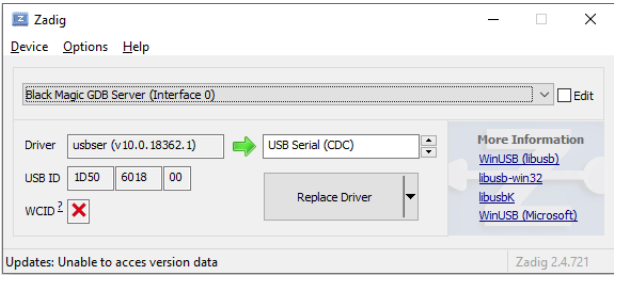
Depending on the project, a galvanic isolation adapter for the USB port may be advisable. The Black Magic Probe provides ESD-protection on the pins of the Cortex Debug header and the UART, but these pins are also rated for an absolute maximum voltage of 6 V. If you are working on boards that carries voltages well above that 6 V limit (e.g. a LED-driver for eight white LEDs in series may exceed 30 V), there is the risk that this voltage breaks out to the low-voltage logic of the board. A multimeter probe that slips and shorts out two pins, may be enough to burn a chip on the target board. If you are unlucky, it may also burn an attached debug probe, but with an isolator, at least you will *not* burn the USB port of your laptop or workstation.

Setting up the Black Magic Probe

Details for adding the Black Magic Probe to your workstation as a USB device, depend on the operating system that you are using, and the data link that you use. This chapter therefore starts with three sections: Microsoft Windows (USB set-up), Linux (USB set-up), and Wi-Fi set-up for ctxLink. You can skip the sections not relevant to you.

Microsoft Windows (USB set-up)

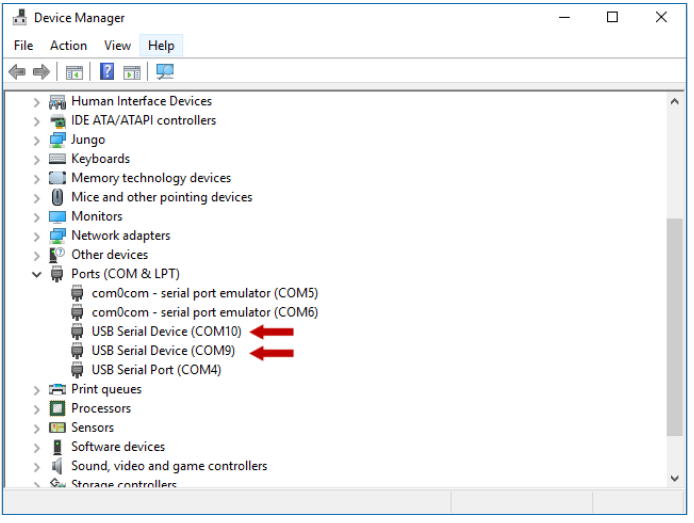
On connecting the Black Magic Probe to a USB port on a workstation, four devices are added. The principal ones are two (virtual) serial ports (COM ports). One of these is for gdbserver and the other is the generic TTL-level UART. The other two are vendor-specific interfaces for firmware update (via the 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 project site for the Black Magic Probe, as well as with this book.

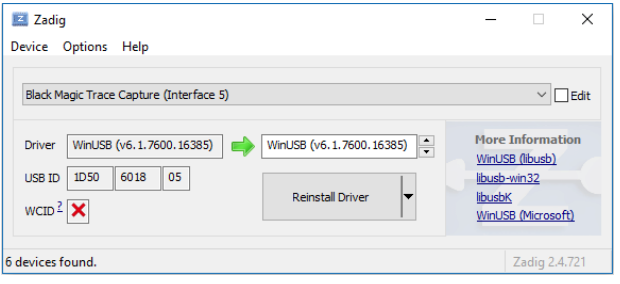
Alternatively, you can set up the CDC driver for the Black Magic Probe with the free utility "Zadig" by Akeo Consulting (see also Further Information on page 182). When using Zadig, you need to set up both interfaces 0 ("Black Magic GDB Server") and 2 ("Black Magic UART Port") to "USB Serial (CDC)." You may need to first select List All Devices in the Options menu to see the interfaces of the Black Magic Probe.

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 book).



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.[1] The most convenient way to do so is by running the aforementioned "Zadig" utility (see Further Information on page 182).



You need to register both interfaces 4 ("Black Magic Firmware Upgrade") and 5 ("Black Magic Trace Capture"). 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.

[1] More on the choice between WinUSB and libusbK on the next page.

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 that the Black Magic Probe has different USB IDs (VID:PID pairs) in DFU mode versus normal mode ("run mode"). In DFU mode, the ID is `1D50:6017`; in run mode, it is `1D50:6018`.

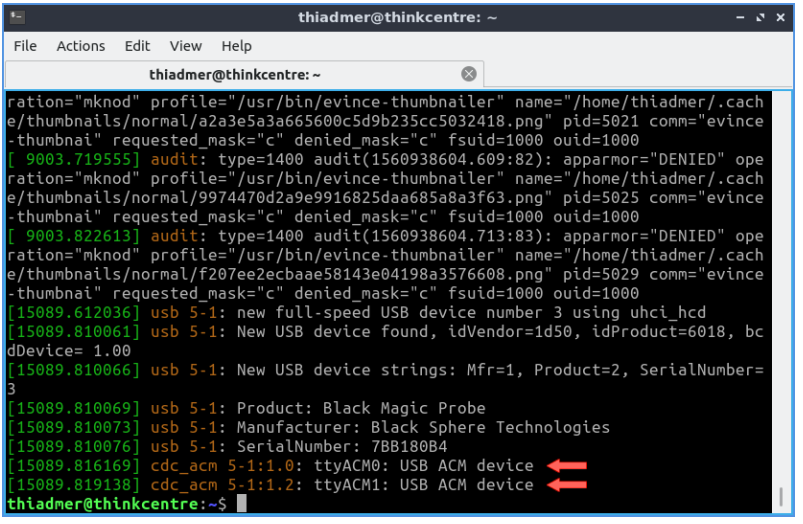
The choice between WinUSB and libusbK depends on the PC-hosted software that you wish to use for trace capture. The firmware upgrade tool dfu-util (see Updating Black Magic Probe Firmware on page 133) supports both WinUSB and libusbK. The debugger front-end and trace viewer that accompany this book also support both WinUSB and libusbK, and in this case WinUSB is preferred (because it is pre-installed). The Orbuculum tool-set (see section Monitoring Trace Data on page 84), however, is based on libusb, and the Microsoft Windows port therefore requires the libusbK driver.

Linux (USB set-up)

After connecting the Black Magic Probe to a USB port, two virtual serial ports appear. One of these is for gdbserver and the other for the generic TTL-level UART. 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 following picture). Alternatively, you can run the BMScan utility from inside a terminal (BMScan is a companion tool to this book, see also page 27).



To be able to access the serial ports, a non-root user must in most cases be included in the `dialout` group. 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. Reportedly, some distributions use the `plugdev` group rather than the `dialout` group.

No driver needs to be installed for the firmware update and trace capture interfaces, but if you wish to use those features as a non-root user (so without 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.

When you copy the file `55-blackmagicprobe.rules` (printed below) into the directory `/etc/udev/rules.d`, it allows any user to access the trace capture interface of the Black Magic Probe.

```
# 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"
```

After adding the udev rules, you must reload the rules (or alternatively: refresh the session by logging out and logging in again).

```
sudo udevadm control --reload-rules
sudo udevadm trigger
```

Wi-Fi set-up for ctxLink

When ctxLink is connected to a workstation via USB, the set-up is the same as for the Black Magic Probe. To use the Wi-Fi interface for debugging, the first issue to decide on is how to power the ctxLink. The options are to use a net adapter with a USB-micro connector, a 3.7 V Li-Po battery, or to power ctxLink from the target. See page 16 for setting the jumpers for the power selection of ctxLink.

The "mode" LED periodically performs a blink sequence, which indicates both the Wi-Fi status and the battery status. See the picture at page 15 for the "mode" LED and the associated button.

Pulses per sequence	Status
none (LED off)	Wi-Fi not active, power good
1	Battery low
2	Connected to a Wi-Fi access point
3	WPS configuration in progress
4	HTTP Provisioning in progress

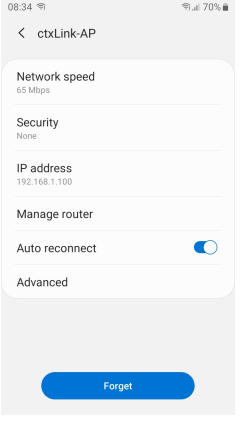
The "mode" button enables you to activate either of the Wi-Fi configuration modes, or to cancel a Wi-Fi configuration, by pressing it for a particular duration: 4 seconds for WPS configuration, 6 seconds for HTTP Provisioning, 7 seconds or more to cancel either configuration mode.

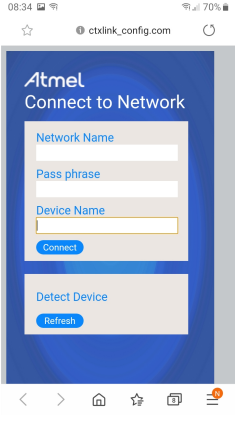
To add the ctxLink to a wireless LAN via WPS (Wi-Fi Protected Set-up), first start the WPS function at the access point (e.g. the wireless router). This is typically done by pushing a button at the router. Then press the "mode" button on the ctxLink for 4 seconds (more accurately: between 3 and 5 seconds). On release of the button, the mode LED will blink in sequences of 3 pulses until the set-up completes. Note that an access point typically shuts WPS off after 2 minutes, so you have to start WPS configuration on the ctxLink fairly quickly after starting it on the access point.

If WPS is not an option, the alternative is to use HTTP Provisioning. With this method, you temporarily set up the ctxLink as an access point, after which you connect to that access point with a laptop or smartphone. You will then be presented with a form that allows you to enter the SSID and pass-phrase of the Wi-Fi access point that ctxLink must connect to.

The first step is to press the "mode" button for 6 seconds (to be precise: between 5 and 7 seconds). Then, use a laptop or smartphone to connect to the new access point with the name "ctxLink-AP." Possibly you will be asked to confirm that you want to log in. On the form that appears next, fill in (or select) the network name ("SSID") of the access point and the pass-phrase (those are the SSID and pass-phrase of the Wi-Fi router that you want ctxLink to connect to), and click on the "connect" button. After a short while, the "mode" LED of the ctxLink should start blinking is sequences of two pulses, as an indication of a successful connection.

On some systems, the form for the network name and pass-phrase does not automatically pop up. To open it explicitly, you can tap on the link ctxLink-AP, which gives you the screen at the left (in the picture below). In this screen, choose the option "Manage router," to arrive at the form, as shown at the right. Alternatively, you can open a browser and browse to `192.168.1.1`.





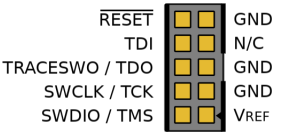
Once connected to the access point, the ctxLink acquires an IP address via DHCP. One way to retrieve this address is to look it up in the list of the DHCP server (typically the wireless router). The ctxLink announces itself as `ctxLink_0001` to the DHCP server. In case an access point does not show the device names, the MAC address of the ctxLink is printed on the Wi-Fi module.

You can also use the BMScan utility to scan the network for ctxLink devices

and return their IP addresses; see section Checking the Set-up on page 27 for more information on BMScan.

Connecting the Target

The Black Magic Probe has a 2×5-pins 1.27 mm 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. Otherwise, you will need an adapter board or a break-out board.



Hardware version 2.3 of the Black Magic Probe allows for a variation on the above pin-out, by way of two "jumpers." See the chapter Unified Connector: Debug + UART on page 178 for details.

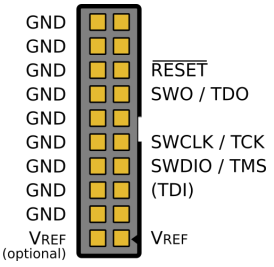
When using a break-out board (see page 16 for an example), you may get away with wiring less than the seven distinctive pins/signals from the Cortex Debug header. Of the pins on the debug connector, SWCLK, SWDIO and GND are essential. These must always be connected to the target.

The RESET pin is recommended as well, as it allows you to reset the target device from the firmware. In some situations (such as when the target microcontroller redefines the SWCLK & SWDIO pins), it may be required to keep the microcontroller in reset while connecting to it.

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 50). The maximum current that can be drawn from the Black Magic Probe is 100 mA.

Finally, the TRACESWO pin is for debug tracing and profiling. This pin is therefore needed if you use either of these features. The TDI line is used for JTAG scan, not for SWD debugging. This pin is often omitted on break-out boards.

When the target board has a 20-pin JTAG header, you can also use a break-out board and wire the pins individually. For debugging, the TDI pin is not used, and therefore it does not need to be connected in most cases.



Checking the Set-up

When the Black Magic Probe is connected to a USB port, the green and orange LEDs (labelled "PWR" and "ACT" respectively) should be on. The ACT LED may be bright or dim, depending on the firmware version and the operating system.

If you have not checked which serial port the Black Magic Probe uses for its gdbserver, run BMScan on the command line.

```
d:\Tools> bmscan

Black Magic Probe [Version: 1.8.2, Hardware Version 3, Serial: 7BB180B4]
  gdbserver port: COM9
  TTL UART port:  COM10
  SWO interface:  {9A83C3B4-0B99-499E-B010-901D6C2826B8}
```

The above command works for a ctxLink debug probe connected to USB too. For a ctxLink that is set up for Wi-Fi, you can scan the local network for the debug probe with the following command:

```
d:\Tools> bmscan ip

ctxLink found:
  IP address: 192.168.0.214
```

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 microcontroller (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 number is above 9, the string "`\\.\`" must be prefixed to the port name. For example, COM port 10 is specified as "`\\.\com10`". On 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") may increase in brightness (this depends on the firmware version of the Black Magic Probe). This LED responds to the `DTR` signal set by GDB; this was a physical line on the RS232 port, but now just a status command on a virtual interface.

The next step is to scan for the target microcontroller. There are two ways to do this: `swdp_scan` for microcontrollers 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 microcontroller. 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 32.

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 on 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, see also section GDB on Microsoft Windows on page 148).

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 aborts parsing the `.gdbinit` file on the first error, so the remainder of the file is not executed either. My recommendation is, therefore, to only add user-defined commands in `.gdbinit`.

```
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
```

The above definition gives you a shorthand for conveniently connecting to the Black Magic Probe with a single command.

```
(gdb) bmconnect com9
```

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:

```
(gdb) set auto-load local-gdbinit
```

You can also load a GDB "command file" explicitly with the "`source`" command, as below (and you can include such statements in the `.gdbinit` to load auxiliary command files):

```
(gdb) source ../share/orbcode/gdbtrace.init
```

Design for Debugging

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 microcontrollers 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 anymore, but also that no new code can be flashed into the microcontroller.

Depending on your microcontroller, a way to circumvent this is to enable the option `connect_rst` in the Black Magic Probe (see page 51 for the command description). The Debug Access Port of the ARM Cortex is designed such that it may stay active while the remainder of the microcontroller is in reset, so that a debug probe can attach to it. This is precisely what the `connect_rst` option does: it pulls the reset pin on the connector low while performing a SWDP scan, as well as during the `attach` command. Whether or not the ARM Cortex debug port is enabled during reset, depends on the microcontroller, however. For example, NXP's low-end microcontrollers with a Cortex M0(+) core use the $\overline{\text{RESET}}$ pin to switch between JTAG and SWD (disabling SWD while $\overline{\text{RESET}}$ is low).

As an alternative, you can often use system-specific pins to force a microcontroller into boot mode. The LPC series of microcontrollers from NXP have a $\overline{\text{BOOT}}$ pin that forces the microcontroller 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 —though BOOT0 must be pulled high, rather than low. 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 microcontroller is therefore to branch out the "boot" pin(s) to a jumper or 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 SWD pins, but you can debug the rest.

There are a few methods for the firmware to detect that it is running under a debugger. A straightforward test is to check whether the low bit of the *Debug Halting Control & Status Register* (DHCSR) is set. This works on a Cortex M3/M4/M7 microcontroller, however; on the Cortex M0/M0+ microcontroller architecture, this register is not accessible from firmware (it is accessible from the JTAG/SWD interface).

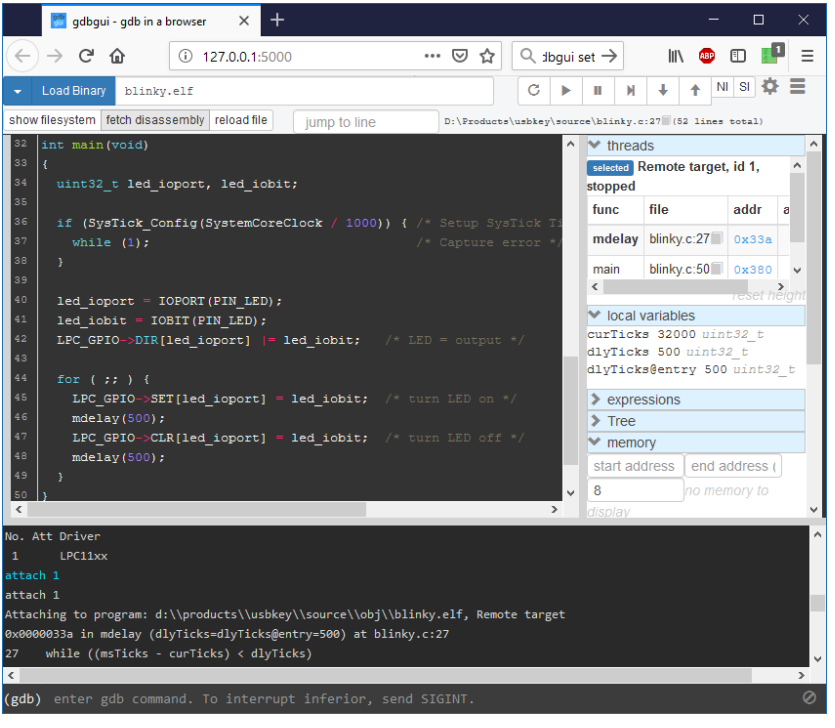
```
if ((CoreDebug->DHCSR & 1) == 0) {
    /* not running under a debugger, free to redefine pins */
}
```

An alternative is to have a weak pull-up on the SWDIO pin and probe it (as a general-purpose I/O pin) on start-up. The idle state SWDIO and SWCLK lines is defined as low. Thus, the Black Magic Probe pulls SWDIO low (provided that it senses a voltage on the VREF pin). This does require some pin juggling, though: you first have to configure the SWDIO pin as an "input" I/O pin (with a pull-up) to be able to read it, and depending on the value read, quickly change it back to SWDIO. Two side remarks: firstly, if the SWDIO pin is connected to other circuitry (that may drive the pin low), this trick won't work —but sharing the SWDIO pin with other functions is bound to be challenging. Secondly, I choose the SWDIO pin rather than SWCLK because the SWD specification recommends a pull-down on SWCLK, and a pull-up on SWDIO.

Debugging Code

Debugging code for embedded systems has its own challenges, in part due to the way that microcontroller 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 pertinent to 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 6), you will probably prefer a front-end to do any non-trivial debugging. Below is a screen-capture of gdbgui connected to the Black Magic Probe, and ready to debug "`blinky`."



The 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 pop-up 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, and stored in a `.gdbinit` file, see Running Commands on Start-up on page 28. 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. The default format for the debug symbols is typically DWARF, and this would also be the preferred format.

You can specify the target executable file on the command line when launching GDB, or you can set it with the `file` command. When using the `file` command, the filename parameter may include 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 microcontroller series, an additional step is recommended before the `load` command. NXP designed the microcontrollers 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 core 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 microcontrollers that this applies to,[1] have the feature to remap address range `0...511` to either Flash, RAM or ROM memory, via the register `SYSMEMREMAP` or `MEMMAP`. According to the NXP manuals, 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 microcontroller is halted by the debug probe, `SYSMEMREMAP` is still 0. Then, if you download new code in the microcontroller, the bottom 512 bytes will be sent to ROM, and be lost.

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

[1] It does not apply to the LPC546xx series, for example, which have the internal bootloader at address `0x03000000`, so that there is no conflict with the user code.

```
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 28:

```
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 now 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 microcontroller range.

NXP series	Register	Address	Flash map
LPC800, LPC1100, LPC11U00, LPC1200, LPC1300	SYSCON_SYSMEMREMAP	0x40048000	2
LPC1500	SYSCON_SYSMEMREMAP	0x40074000	2
LPC1700	SCB_MEMMAP	0x400FC040	1
LPC2100, LPC2200, LPC2300, LPC2400	SCB_MEMMAP	0xE01FC040	1
LPC4300	M4MEMMAP	0x40043100	0
LPC54100	SYSCON_SYSMEMREMAP	0x40000000	2

The "`mmap-flash`" snippet in `.gdbinit` therefore needs to be adapted for the particular microcontroller as well. A more complete version of the `mmap-flash` user-defined command is in the `.gdbinit` file that comes with this book.

Reset Code Protection

Code protection means that the Flash memory can no longer be read, through a debug probe or programmer. How code protection is implemented, varies between brands and microcontroller families, and so the method to remove it is also specific to each family.

- *STM32 microcontroller series*

In the STM32 family, code protection is called "**r**ea**d**out **p**rotection" (RDP). When the `load` command gives the following error, that is a clue that RDP is set in the option bytes.

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

To check whether code read protection is set, use the "`monitor option`" command.

```
(gdb) monitor option
usage: monitor option erase
usage: monitor option <addr> <value>
0x1FFFF800: 0x5AA5
0x1FFFF802: 0x00FF
0x1FFFF804: 0x00FF
0x1FFFF806: 0x00FF
0x1FFFF808: 0x00FF
0x1FFFF80A: 0x00FF
0x1FFFF80C: 0x00FF
0x1FFFF80E: 0x00FF
```

The above reply is for the case that the code is *unprotected*. As a side note, option bytes are written in pairs: value and complement. The option is only valid if the complement matches. For unprotected code, the value for the option is `0x5A`, and its complement is `0xA5`.

When read-out protection is set, older firmware shows the list of option words, as above, but with a different value than `0x5AA5` in the first option word. Current firmware responds as follows:

```
(gdb) monitor option
Device is Read Protected
Use "monitor option erase" to unprotect, erasing device
```

The Flash memory cannot be read, but in addition, no new code can be downloaded unless the option bytes are erased first —which you do by adding the "`erase`" parameter to the `monitor option` command.

```
(gdb) monitor option erase
0x1FFFF800: 0x0000
0x1FFFF802: 0x0000
0x1FFFF804: 0x0000
0x1FFFF806: 0x0000
0x1FFFF808: 0x0000
0x1FFFF80A: 0x0000
0x1FFFF80C: 0x0000
0x1FFFF80E: 0x0000
```

After erasing the option bytes, the microcontroller must be power-cycled to reload them (the output of the `option erase` command does not reflect the true values of the option bytes; after reset, you will see that the first option word is actually set to `0x5AA5` instead of `0x0000`). Note that GDB will lose the connection to the target on a power-cycle, so you must rescan and re-attach to the target again.

The side effect of clearing RDP, is that it wipes the entire Flash memory (which is, of course, by intent).

A caveat is that the STM32 family has two levels of RDP, and only RDP Level 1 can be reverted by erasing the option bytes. RDP Level 2 cannot be undone. In fact, RDP Level 2 disables the SW-DP port, so you will not be able to list (or erase) the option bytes.

To set code protection on a STM32Fxx microcontroller, use the `monitor option` command, with the address of the first option word and the new value. The magic for RDP Level 2 is `0x33CC`; any value other than `0x5AA5` or `0x33CC` means RDP Level 1. The new level is picked up after a power-cycle.

```
(gdb) monitor option 0x1ffff800 0x00ff
```

- *SAMD microcontroller series*

When code protection is enabled on the SAMD microcontroller series, Flash memory must be fully erased, in order to remove the read protection. On an up-to-date release of GDB, you can give the command:

```
(gdb) flash-erase
```

- *LPC microcontroller series*

When code protection is enabled on the LPC microcontroller series, Flash memory must also be fully erased before new firmware can be downloaded. However, as discussed on page 41, the `flash-erase` command will often fail.

More fundamentally, though, enabling code protection on the LPC series disables the SW-DP port. So you cannot even *attach* to the microcontroller in GDB. Instead, your options are to erase Flash memory either via the serial bootloader (ISP), or from within your firmware (that is, you have added a "self-destruct" function in the firmware that erases Flash memory via an IAP call —and which is triggered by a special command or a special status on power-up).

If you accidentally download firmware with code protection onto an LPC microcontroller, and you notice this before resetting (or power cycling)

it, you can still erase Flash memory with a GDB command. You only have to erase the first sector, so the following will work:

```
(gdb) monitor erase_range 0 1024
```

See page 53 for details on the `monitor erase_range` command. Note that it requires firmware version 1.9.

Verify Firmware Integrity

To verify that the code in the microcontroller 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 microcontrollers from NXP: these microcontrollers 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 during downloads, despite that it is already correctly set —but that is harmless. After downloading, the code in the microcontroller 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 book.

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.

Getting help and information

Oft overlooked, but the `help` and `info` commands are among the most useful for a command-line tool like GDB.

`help`	Show a list of topics you can get help on.
`help` *`topic`*	Show help on a topic, usually including a list of relevant commands. For the list of topics, type "`help`" without any parameter.
`help` *`command`*	Show the syntax and parameters of the command, plus a brief description of its purpose or function. For a list of commands, type "`help all`".
`info`	Show a long list of topics you can get information about.
`info` *`topic`*	Show the information GDB has on the topic. Topics can range from variables and arguments, stack, targets and the sources that built it, breakpoints, watchpoints, etc.

The difference between the `help` and `info` commands is that `help` gives information on how to do things in GDB, and `info` informs you about the status of GDB or the program loaded into it. For example, "`help break`" gives you a page that describes how to set a breakpoint, whereas "`info break`" lists the breakpoints that are set, plus the properties of each of these.

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.

The DWARF format stores the paths to the source files along with the debug symbols. Thus, if you debug the firmware on the same workstation as that you built it, GDB automatically locates the source files. However, if you start debug the firmware on a different workstation than on which it was built, or if the source files moved since the firmware was built, the (new) paths to the source code needs to be set in GDB.

directory *path* **directory** *path*:*path* (*can be abbreviated to* **dir**)	Add a search path for source files. Multiple paths may follow the command; the paths are separated with a colon (":") in Linux and a semicolon (";") in Windows.

A front-end often has its own implementation for displaying the source files —which does not rely on GDB. Thus, you may need to use a menu function or option in the front-end, instead of the `directory` command.

Downloading code into the microcontroller

These commands were covered in section Loading a File and Downloading it to the Target on page 33.

file *path*	Set the path of the ELF file to debug (if it was not already given on the GDB command line). The directory separator in the `path` parameter must be a slash ("/") —even on Microsoft Windows
load **load** *filename*	Download the code sections of the ELF file into the microcontroller. In the common case, no parameter is given with the `load` command, in which case the file set with the `file` command is downloaded. However, you may specify the file to download explicitly.
compare-sections	Check whether the code in the microcontroller matches the ELF file being debugged; see Verify Firmware Integrity on page 38.

flash-erase	Erases the Flash memory regions on the target. See notes below; see also the `monitor erase_mass` command on page 53.

See also chapter Firmware Programming (see p.Firmware Programming) for programming Flash in a batch file or command script, or for downloading other file formats than ELF executables.

To erase all Flash memory on the target microcontroller, GDB provides the `flash-erase` command. However, it relies on the memory information that the debug probe provides, and the Black Magic Probe on occasion groups together "families" of microcontrollers that share the same "driver." Those microcontrollers have different sizes of Flash and SRAM memories, though, which Black Magic Probe solves by reporting the *maximum* memory for the series. Hence, unless you are working with the most capacious microcontroller of a family, GDB will have attempt to erase more Flash memory than the microcontroller has, and which will consequently fail with an error message.

It is therefore more reliable to use the `monitor erase_range` command (see page 53), and pass the range of Flash memory explicitly as parameters. Note that the `monitor erase_range` command requires firmware version 1.9.

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). You will usually set one or more breakpoints first.
continue (*can be abbreviated to* **c**)	Continue running (from the current execution point). A count may follow the command, but it is only relevant if execution 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. It amounts to the same as setting an ignore count on the breakpoint.

step (*can be abbreviated to* **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 (*can be abbreviated to* **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 (*can be abbreviated to* **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 (*can be abbreviated to* **fin**)	Continues execution until it steps out of the current function, then stops at the location from where the function 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 always step over particular functions, with the `skip` command (it *skips* stepping *into* the function). The purpose of `skip` is easiest explained with an example:

```
▶   transmit_data(array, setup_connection());
```

When a function has a call to another function in its parameter list, the `step` command will step into the nested function first. In this example, it will step into `setup_connection()` and only go into `transmit_data()` afterward. Suppose we want to step into `transmit_data()`, but that needing to step through `setup_connection()` first is a chore. To do this, mark `setup_connection()` to be skipped:

```
(gdb) skip setup_connection
```

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 57 for details.

Altering execution flow

In case that you need to break out of an endless loop, or a case where you know that continuing running the remainder of the function will do no good, you can alter the execution flow.

jump *line* **jump** *file:line*	Start running at the given location. You can use this command to break out of an endless loop, or to jump back a few lines to re-examine the control flow.
return **return** *expression*	Skips to the return address of the current function (without executing the code between the current location and the return point). The program stays in stopped state.

In brief: `jump` is like `continue`, but starting from a different location; and `return` is like `finish`, but without executing the code.

See also the `set` command on page 46, because you can often also change control flow (or exit a loop) by changing a variable.

Breakpoints and watchpoints

When creating a breakpoint or watchpoint, it gets assigned an ID. This is simply a sequentially incrementing number to identify the breakpoint or watchpoint. Several of the commands listed below, for example to disable, enable or delete breakpoints, take the "breakpoint ID" as a parameter.

break *line* (*can be abbreviated to* **b**)	Set a breakpoint at the line number, in the current source file.
break *file:line*	Set a breakpoint at the line number in the specified source file.
break *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.
rwatch *expr*	A watchpoint that triggers when the variable (or memory location that the expression points to) is *read*. This requires hardware breakpoints.
awatch *expr*	A watchpoint that triggers on *access*—either read or write. It requires hardware breakpoints.
info break	Show the list of breakpoints and watchpoints, together with the sequential index numbers (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* **disable** *id …*	Delete a breakpoint on the given line or function. It allows the same options as the `break` command. 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.
ignore *id count*	Ignores the first `count` times that the breakpoint is hit. Note that when using the `continue` command after being stopped on a breakpoint, you can set the "ignore" count as a parameter on the `continue` command.
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 (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 90.
dprintf *location, format, …*	Sets a breakpoint at the location (with the same syntax as the `break` command), and attaches a "`printf`" command to it. After evaluating any expressions that are listed behind the format string and printing it on the console, this breakpoint automatically continues. See section Tracing with Command List on Breakpoints on page 90.

For embedded development, enabling and disabling breakpoints is all

the more useful, because hardware breakpoints (and watchpoints) are a scarce resource. Most Cortex-M microcontrollers offer 6 hardware breakpoints and 2 hardware watchpoints.[2] 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 microcontroller offers, you keep them defined, but disable the ones that are not immediately relevant for the next step in debugging the code.

For a hardware watchpoint, the data type of the expression cannot be wider than the word size of the microcontroller. For example, the word size is 32-bit on an ARM Cortex-M microcontroller, and the expression to watch can therefore be up to four bytes wide.

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 microcontrollers can also break on specific exceptions or interrupts. An exception trap is set with the `monitor vector_catch` command, see page 55. When the exception is caught, the microcontroller 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 setting "variable watches" (which is the equivalent to the `display` command), and they may also automatically list all local variables and their values (the equivalent of the `info locals` 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`* **`print /`***`fmt various`* **`print`** *`var@count`* (*can be abbreviated to* **`p`**)	Show the contents of the variable. GDB can parse C-language expressions to show array elements or dereferenced pointers, like in: `print var[6]` show the value of an array element `print *ptr` dereference the pointer and show the value The format to print the variable in (decimal, hexadecimal, or other) is a single letter; see the list on page 47. The "`var@count`" syntax interprets `var` as the start of an array and prints count elements.

[2] See the table at page 12.

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* **display** */fmt var* (*can be abbreviated to* **disp**)	Watch the variable. Show the variable's value each time that the execution is halted. The format to print the variable in (decimal, hexadecimal, or other) is a single letter; see the list on the next page.
undisplay *num* (*can be abbreviated to* **undisp**)	Remove the watch with the given sequence number.
x *address* **x** */options address*	Display the memory at the given address. The options start with a slash, followed by zero or more digits, and then followed by one or two letters. The digits are the count of elements, the first letter is the format and the second letter the size of each element in bytes.
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.
find */opt start*, *end*, *expr* **find** */opt start*, *+len*, *expr*	Search through memory for a sequence of bytes. The range to search may be specified either by start & end addresses, or by start address and length. Options start with a slash; the first option is a letter to set the expression size, see the size modifiers in the list on see the list on the next page; the second option is the maximum number of matches to return. Nota Bene: note the commas to separate the parameters.

The GDB `print` command records each value that it prints in its "value history," and assigns it a label. The first label is `$1` and it is incremented on each successive `print` command. You can use these labels in expressions.

While the `print` command is typically used to show variables, it is able to evaluate C-style expressions. As such, you can use the GDB `print` command as a built-in calculator.

```
(gdb) p sampleDelay
$1 = 50
(gdb) p $1 / (double)ticksPerSecond
$2 = 0.050000000000000003
```

The `x` command displays the memory at any given address. The address can be an expression that evaluates to an address —which includes variables. If no options are given, GDB uses its defaults, and at start-up the default display format is: a single 32-bit value displayed in hexadecimal. The number of elements (bytes or words) to display can be set after a slash; for example, "`x /4 0x1234`" displays four elements starting at the

given address. This element count then also becomes the new default for the `x` command.

The `print`, `display` and `x` commands each allow a format specification behind the slash (or, for the `x` command, behind the count). The format code consists of a single letter. In case of the `x` command, a second letter may be added to indicate the size of each item —the four last entries in the list below.

```
o    octal
x    hexadecimal
d    decimal
u    unsigned decimal
t    binary
f    floating point
a    address
c    character
s    string (zero-terminated)
i    instruction
b    size modifier: byte (8-bit value)
h    size modifier: halfword (16-bit value)
w    size modifier: word (32-bit value)
g    size modifier: "giant" word (64-bit value)
```

As is the case for the count of elements in the `x` command, the given display format becomes the default for any subsequent `x` command. However, options *not* set in the display format, are reset to their default. For example, you you first say "`x /4 0x1234`" to display four elements starting at address `0x1234`, and then say "`x /h`" to change the elements to 16-bit halfwords, only a single value is displayed —the element count is reset to 1. To display four 16-bit values, use instead "`x /4h`", and this then becomes the new default for the `x` command.

Microcontrollers include peripherals, such as SPI and I^2C serial buses and general purpose I/O pins. The registers for these peripherals are memory-mapped in the ARM architecture. However, by default, GDB won't show data outside the range for program and data memory. That is, the address of a peripheral register is considered an invalid memory address. To view (or set) peripheral registers, first issue the following command:

```
set mem inaccessible-by-default off
```

After this command, GDB considers any address outside the memory map as RAM. The BMDebug front-end (see page 62) automatically runs this command, and other GDB front-ends may do so too. Otherwise, you can include the command in the `.gdbinit` file, so that GDB runs it at start-up.

See section Running Commands on Start-up on page 28 for information on the `.gdbinit` file.

The Call Stack

A stack frame stores the local variables, arguments and the return address for each sub-routine (or function). The scope of local variables and arguments is restricted to the sub-routine that they are declared in.[3] Stack frames form a list, that a debugger can walk up and down. Moving up the stack frame allows you to look at the local variables in the routine that the current routine (that contains the execution point) was called from.

`backtrace` *num* (*can be abbreviated to* **`bt`**)	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. You can go up multiple levels by adding the count, as a parameter.
`down`	Move back to a lower frame. You can go down multiple levels by adding the count, as a parameter.
`frame` *idx* (*can be abbreviated to* **`f`**)	Move to the given frame index (the backtrace command prints these index numbers). The frame command *without* parameter prints the active frame index.

GDB numbers the stack frames sequentially, starting from zero for the sub-routine that the current execution point is in. With the command "`frame 0`", you will return to the frame that corresponds with the execution point.

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 that contains the execution point.

Inspecting Machine Code

GDB is primarily used as a source-level debugger, but at times, you may want to look at what happens at the CPU level.

[3] This is a simplification —more accurately, local variables have a scope that runs from their declaration to the end of the compound block that the declaration appears in.

disassemble **disassemble** *start,end* **disassemble /s** (*can be abbreviated to* **disas**)	When used without start & end arguments, it shows the assembly of the function that the execution point is in. The alternative is to specify an address range (start, end) to disassemble. The `/s` option mixes the assembly code with the source code, for reference.
set disassemble-next-line on **set disassemble-next-line off**	When GDB halts execution, it shows the source code line that it stopped on (if that source code line is available). When the option `disassemble-next-line` is switched on, GDB will in addition show the disassembly for the source line at the execution point.
stepi	Like the step command, see page 42, but stepping a single instruction (instead of a source line).
nexti	Like the next command, see page 42, but stepping a single instruction (instead of a source line).
info registers	Print the names and values of the registers of the microcontroller.

GDB treats registers as special variables. You can refer to a register (print its value, assign a new value to it, ...), by prefixing its name with a `$`. In other words, you can add a watch on register `r0` by giving the command:

```
(gdb) display $r0
```

Debug Probe Commands

GDB has a pass-through command to configure or query a `gdbserver` implementation: `monitor` (`monitor` can be abbreviated to `mon`). 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 —divided into several categories. Some of these commands are only available on particular microcontroller series; and the applicable microcontroller series is noted in those cases. Likewise, if a command is only available in a particular firmware version, this is also noted on the command.

Information and status

`monitor` **help**	Show a summary of the commands that the debug probe supports (basically this list, but restricted to commands relevant to the detected target).
`monitor` **version**	Show the current version of the firmware and the hardware.
`monitor` **serial**	Show the serial number of the probe. *Jeff Probe,* Show the serial number of the target. *EFM32, Gecko, SAMD*

`monitor` **`morse`**	When the Black Magic Probe encounters an error that it cannot handle otherwise, it will start to blink the red LED (labelled "err") in a Morse code pattern. In case your Morse code decoding skill is a little rusty, you can instead use this `morse` command to return the error message in plain text on the GDB console. But in fact, the only such error message is "target lost."

Target and protocol configuration

`monitor` **`tpwr enable`** `monitor` **`tpwr disable`**	Enables or disables driving the VCC pin on the 2×5 pin header to 3.3 V. See page 26 for the pin-out of the connector. When the Black Magic Probe drives the VCC pin, it can power the target (maximum current: 100 mA). 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 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 3.3 V TTL levels.
`monitor` **`hard_srst`** `monitor` **`reset`**	Resets the target. *firmware 1.6. . . 1.8* The $\overline{\text{RESET}}$ pin on the 2×5 pin header *firmware 1.9* is briefly pulled low (see page 26 for the connector). This command was renamed from `hard_srst` to `reset` in firmware version 1.9.
`monitor` **`tdi_low_reset`**	Pulls the TDI pin low, before proceeding to *firmware 1.9* reset the target by briefly pulling the $\overline{\text{RESET}}$ pin low.
`monitor` **`frequency`** *`value`*	Sets maximum `SWCLK` frequency. *firmware 1.8* The value is in Hz, but may use a "k" or "M" suffix, to denote kHz or MHz respectively; for example, 1M stands for 1 MHz. A lower frequency for the SWD protocol may be needed depending on the target microcontroller, or on the wiring between the debug probe and the target. When no parameter is given, the command returns the active frequency. However, due to calculation errors, both the set frequency and the returned output are at best *roughly approximate.*

Target scanning

`monitor` **`auto_scan`**	Scan either JTAG or SWD protocols *firmware 1.9* Performs a `jtag_scan` first, followed by an `swdp_scan` if the JTAG scan does not detect devices.

`monitor` **`connect_srst`** *`option`* `monitor` **`connect_rst`** *`option`*	*firmware 1.6. . . 1.8* *firmware 1.9* The *`option`* parameter must be “`enable`” or “`disable`.” When enabled, the target is kept in reset while scanning and attaching to it. See section Design for Debugging on page 30 for more information. This command was renamed from `connect_srst` to `connect_rst` in firmware version 1.9.
`monitor` **`halt_timeout`** *`delay`*	Set time to wait for device to halt. *ARM Cortex-M* The time to wait for the Cortex-M core to halt, so that the debug probe can attach to it. This value is in milliseconds. The default is 2000 ms.
`monitor` **`jtag_scan`**	Scan the devices on the JTAG chain. Like the `swdp_scan` command (see below), this command prints the I/O voltage and the list of targets.
`monitor` **`swdp_scan`** `monitor` **`swdp_scan`** *`id`*	Scan for Serial Wire Debug devices (using the SW-DP protocol). A target “ID” value may optionally be given. The command prints the I/O voltage and the list of targets. It causes a detach if the probe is already attached to a target. See also the `tpwr` command (below) for the I/O voltage, and the `targets` command for the device list.
`monitor` **`targets`**	Show the detected targets. This is the same list as the one returned by the `jtag_scan` and `swdp_scan` commands. Unlike the `jtag_scan` and `swdp_scan` commands, this command does not cause the currently attached target to be detached. For each detected microcontroller, it displays the driver (the driver is often specific to a microcontroller family).

SWO (trace capture)

`monitor` **`traceswo`** `monitor` **`traceswo`** *`rate`*	Enable the SWO capture pin to for trace capture. The rate parameter is the bitrate of the SWO trace protocol. It is used only for asynchronous encoding and on firmware version 1.7 and later, it defaults to 115.2 kbps. Note that the native Black Magic Probe only supports Manchester encoding. The ctxLink probe supports only asynchronous encoding. For capturing SWO output using the BlackMagic Debugger front-end, see the Trace Views on page 69.
`monitor` **`traceswo decode`** *`channels`*	Decode SWO in the Black Magic Probe. *firmware 1.7* The trace output is transmitted over the virtual UART, so that it can be viewed on a serial terminal. The SWO channels to decode can be appended as a

	space-separated number list to the command. If absent, all channels are active.

Note that clones of the Black Magic Probe, and especially those in the low price range, may not support SWO tracing. The Jeff Probe, for example, lacks support for SWO tracing.

Real Time Transfer

`monitor` **`rtt`** `monitor` **`rtt enable`**	Enable RTT and scans memory on the target for channel data structures.
`monitor` **`rtt disable`**	Disable RTT.
`monitor` **`rtt status`**	Reports the status of RTT and the channels that are active. The returned message shows an on/off status for the RTT function, plus a yes/no status for whether the "control blocks" for all enabled channels were found. Notably, if the status shows "`rtt: on found: no`", RTT is enabled (and may be partially functioning) but not all channels have yet been discovered.
`monitor` **`rtt poll`** *`max min err`*	The debug probe polls the queues of the RTT queues regularly; this command lets you set the maximum and minimum interval times (in milliseconds), plus the maximum number of errors before RTT disables itself.
`monitor` **`rtt channel`** *`num ...`*	Enables the channels given by the numbers (and disables all channels not in the list). The numbers must be between 0 and 15. By default, output channels 0 & 1 and input channel 0 are enabled (mimicking `stdout`, `stderr` & `stdin`). If no channel numbers follow this command, it resets these defaults.
`monitor` **`rtt ram`** *`start end`*	Sets the region of memory to scan on the target for the RTT control blocks to the start and end addresses. The values must use hexadecimal format. The default is to scan the full RAM range of the target microcontroller.
`monitor` **`rtt ident`** *`name`*	Sets the signature of the control block to search for. RTT has a default signature ("`SEGGER RTT`"), but this can be overruled by the target firmware. The debug probe needs the signature in order to discover the channels. Underscore characters in the name are replaced by spaces. If no name is given, the default signature is restored.
`monitor` **`rtt cblock`**	Reports the details of the discovered channels ("control blocks").

Support for Real Time Transfer is an optional feature on the Black Magic Probe and generally requires firmware release 1.8 (or later). See section Real Time Transfer (RTT) on page 87 for more information on the protocol,

and Building from Source on page 135 for instructions on how to enable RTT on your probe.

Target memory operations

`monitor` **`erase_mass`**	Erase entire Flash memory of the target. *firmware 1.9* (Not available on all target drivers.)
`monitor` **`erase_range`** *`addr num`*	Erase a range of Flash memory starting at or *firmware 1.9* before the given address, and up to or after the address plus the number of bytes to erase. Flash memory is arranged in *pages* and pages can only be erased completely (not in part). Hence, the address is converted to a page range. If the address start and end parameters do not fall on page boundaries, more Flash memory is erased than specified.

There are additional commands for erasing Flash memory, or for clearing & setting EEPROM, that are specific to a microcontroller series. These are listed in the next section. The above commands were added in firmware version 1.9 as a generic way to erase Flash memory —but even so, the `monitor erase_mass` command is, at the time of writing, not available for all microcontrollers that the Black Magic Probe supports.

Miscellaneous (MCU/architecture-specific)

The commands in this table are listed in alphabetical order —not grouped per target.

`monitor` **`convert_tdio`** *`enable`*	Map TDI/TDO to TxD/RxD *Jeff Probe* Toggles the TDI and TDO pins on the Cortex Debug connector to UART TxD and RxD functions (similar to, but not compatible with, the “Unified Debug Connector,” see page 178.
`monitor` **`eeprom`** *`type addr val`*	Set values in EEPROM (non-volatile memory). *STM32L0x,* The first parameter is one of: *STM32L1x* `byte` 8-bit value `halfword` 16-bit value `word` 32-bit value The second parameter is the address in the EEPROM. The third parameter is the value (with the size as specified in the first parameter).
`monitor` **`erase_bank1`**	Erase entire Flash memory in bank 1. *STM32L4xx*
`monitor` **`erase_bank2`**	Erase entire Flash memory in bank 2. *STM32L4xx*
`monitor` **`erase_sector`** *`addr length`*	Erase Flash sector. *Raspberry RP2040* Erases Flash memory on the Raspberry Pi Pico.
`monitor` **`erase_uicr`**	Erase the UICR registers. *nRF51xxx series*

`monitor` **`gpnvm_get`**	Get value of the GPNVM register. *SAM3N, SAM3S, SAM3U, SAM3X, SAM4S*
`monitor` **`gpnvm_set`** *`bit val`*	Set bit in the GPNVM register. *SAM3N, SAM3S, SAM3U, SAM3X, SAM4S* The first parameter is the bit number. The second parameter is the value for the bit (0 or 1). Bit 0 is the *security bit*, which enables code read protection.
`monitor` **`heapinfo`** *`hb hl sb sl`*	Set semihosting heapinfo values. *ARM Cortex-M* Four hexadecimal values must follow this command, for the heap base, the heap limit, the stack base and the stack limit. The startup code in the "newlib" C library uses a semihosting call to get the heap and stack space, if running under a debugger. This command lets you override the defaults for the heap and the stack.
`monitor` **`lock_bootprot`**	Set boot protections. *SAMD* Starting with firmware 1.8, a value may follow the `lock_bootprot` command; where a value of 0 protects the first 32 KiB of Flash, and each next higher value halves the size of the protected range (1 = 16 KiB, 2 = 8 KiB, . . . 6 = 512 B); a value of 7 removes the protection. If no value is present, boot protection is set to the first 32 KiB of Flash memory. See also `unlock_bootprot`.
`monitor` **`lock_flash`**	Lock Flash memory against accidental change. *SAMD* Starting with firmware 1.8, a 16-bit value may follow the `lock_flash` command; each *zero* bit in the value locks the respective region. If no value is present, all Flash regions are locked. See also `unlock_flash`.
`monitor` **`mbist`**	Run the "Memory Built-In Self Test" (MBIST). *SAMD*
`monitor` **`mkboot`** *`bank`*	Make Flash bank bootable. *LPC4300 Cortex-M4* The parameter is the bank number, 0 or 1.
`monitor` **`option`** *`address value`* `monitor` **`option erase`**	Set option bytes. *STM32L1x, STM32L4xx STM32Fxx, STM32L0x,* The first syntax, "`option address value`," stores a value at the address of an option byte (register). The second syntax, "`option erase`," is to erase the option bytes. If read protection set in the option bytes, erasing them implicitly erases the entire Flash memory.
`monitor` **`protect_flash`**	Enable Flash code read protection. *nRF51xxx series*
`monitor` **`read`**	Read target device parameters. *nRF51xxx series* 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	
`monitor` **`read_uid`**	Print the unique serial number of the microcontroller.	*LPC11xx, LPC15xx*
`monitor` **`redirect_stdout`** *`flag`*	Forward semihosting console output to the secondary UART interface. The *`flag`* parameter is `enable` or `disable`.	*ARM Cortex-M*
`monitor` **`reset_usb_boot`**	Reboot into UF2 bootloader. Resets the Raspberry Pi Pico as if the BOOTSEL button was pressed.	*Raspberry RP2040*
`monitor` **`sector_erase`** *`address`*	Erase the Flash page (sector) that the the specified address falls in.	*Kinetis, MSP432*
`monitor` **`serial`**	Print the serial MCU number.	*EFM32, SAMD*
`monitor` **`set_security_bit`**	Set the security bit in the target MCU. The security bit protects Flash memory from being read.	*SAMD*
`monitor` **`unlock_bootprot`**	Set boot protections to minimum.	*SAMD*
`monitor` **`unlock_flash`**	Unlock Flash memory.	*SAMD*
`monitor` **`unsafe`**	Allow programming the security byte. The parameter must be `enable` or `disable`.	*Kinetis*
`monitor` **`user_page`**	Print the user page from Flash.	*SAMD*
`monitor` **`vector_catch enable`** *`vec`* `monitor` **`vector_catch disable`** *`vec`*	Break on specific exceptions. The first parameter is `enable` or `disable`. The second parameter is the exception: `hard` Hard fault `int` Interrupt/exception service errors `bus` Bus fault `stat` Fault state error `chk` Divide by zero, misaligned memory access `nocp` No coprocessor (on coprocessor instruction) `mm` Memory Manager fault `reset` Core reset Cortex-M0 and M0+ microcontrollers only support "`reset`" and "`hard`" fault. A hard reset cannot be caught, though.	*ARM Cortex-M*

Edit-Compile-Debug Cycle

While stepping through code or analyzing trace output, you may spot a bug or some problematic code. The tendency is to leave the debugger, fix it while it is fresh in your head, rebuild and re-test. However, you do not need to leave the debugger to edit & rebuild the code. In fact, it is recommended that you leave the debugger open, and (after fixing and re-building it) simply reload the firmware in GDB —with the `load` command, see page 40. This way, breakpoints and other settings are preserved.

You can even run commands or tools directly from the GDB prompt. By typing a "!" as the first character on the GDB command line, what comes

behind the exclamation mark is then run in the shell or command processor. For example, the line below runs make to build the "test" target.

```
(gdb) !make test
```

After building new firmware from within GDB, you will still need to load it into the target, of course.

With the BMDebug front-end (see section The BlackMagic Debugger Front-end, page 62), the recommended way to reload the ELF file is to use the button "reset" at the top left of the source view, or the key combination Ctrl+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 ticked in the "Configuration" section in the sidebar).

The `gdbgui` front-end (page 32) 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.

Note that when the "Download to target on mismatch" option is disabled in the configuration, the `reset` command or button in BMDebug reloads the source files, but does not download the rebuild ELF file to the target. You will need to use the `load` command, or temporarily force reloading with the command:

```
(gdb) reset load
```

Another reset option that you may need in special occasions, such as when the code has accidentally redefined the SWCLK or SWDIO pins, is:

```
(gdb) reset hard
```

This option does a full reset of GDB, and either resets or power-cycles the target (depending on whether the "Power Target" option is set in the configuration).

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. You may not immediately notice this, 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.

Semihosting: target to host I/O bridge

Semihosting is a technique by way of which the target (or "device under test") can use the host for console and file I/O, via the intermediary of the debug probe.

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 microcontroller 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 that follows it, and enters semihosting state. It then analyzes 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.

```
__attribute__((naked))
int semihosting(uint32_t command, void *params)
{
  __asm__ (
      "mov  r0, %0 \n"
      "mov  r1, %1 \n"
      "bkpt #0xAB \n"
      "bx   lr \n"
```

```
    :
    : "r" (command), "r" (params)
    : "r0", "r1", "memory"
  );
}
```

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`. The snippet uses GCC syntax, for other compilers, the inline assembler may need to be adjusted.

The `semihosting()` function takes a command code and a pointer to a parameter block. The output of a command is stored back in the parameter block. If a command has no parameters (and no output), `params` should be set to `NULL`.

The available command codes are in the following table.[4]

Command	Value	Description
SYS_OPEN	0x01	Open a file on the host system; returns a handle.
SYS_CLOSE	0x02	Close a file opened with SYS_OPEN.
SYS_WRITEC	0x03	Write a single character to the debugger console (stderr).
SYS_WRITE0	0x04	Write a zero-terminated string to the debugger console (stderr).
SYS_WRITE	0x05	Write data to a file handle (the length of the data block is explicitly passed).
SYS_READ	0x06	Read data from a file handle.
SYS_READC	0x07	Read a single character from the debugger console.
SYS_ISERROR	0x08	Check whether the return code of a preceding semihosting call is an error or a normal status. Most commands return -1 on failure.
SYS_ISTTY	0x09	Check whether the file handle refers to a TTY device. GDB considers only file handles 0, 1 & 2 TTY devices.
SYS_SEEK	0x0A	Set the position in the file.
SYS_FLEN	0x0C	Return the length of a file.
SYS_TMPNAM	0x0D	Create a temporary file.
SYS_REMOVE	0x0E	Delete a file.
SYS_RENAME	0x0F	Rename (or move) a file.
SYS_CLOCK	0x10	Return execution time in hundredths of a second (centiseconds).
SYS_TIME	0x11	Return the number of seconds since the Unix Epoch (00:00:00 January 1st, 1970).
SYS_SYSTEM	0x12	Execute a command on the host. See also page 62.
SYS_ERRNO	0x13	Return the error code of the last operation.
SYS_GET_CMDLINE	0x15	Return any arguments passed to the firmware.

[4] A full description of the commands, parameters and return values, is in the document "*Semihosting for AArch32 and AArch64*" made available by Arm Ltd.

SYS_HEAPINFO	0x16	Return base addresses and sizes for stack and heap.
SYS_EXIT	0x18	Report that an exception has occurred (exit because execution has completed, is also considered an "exception").
SYS_EXIT_EXTENDED	0x20	Like SYS_EXIT, but with an extra field to give a more detailed status for the exit, such as an exit code.

The most common use of semihosting, is to print text on the debugger console, as in the next snippet.

```
void sys_write0(const char *text)
{
   semihosting(SYS_WRITE0, text);
}
```

When calling `sys_write0("Hello world")` from your code (and running it from GDB), this text will be printed on the GDB console. The command code `SYS_WRITE0` (value 4, see the previous table) is for printing a zero-terminated string on the debugger console. The Black Magic Probe translates this to a write to file handle 2, also known as `stderr`.

The more general file write operation, which you can also use to print text on the console by passing `stdout` or `stderr` for the file handle, is wrapped in a similar way.

```
int sys_write(int fd, const char *buffer, size_t size)
{
  uint32_t params[3] = { fd, (uint32_t)buffer, size };
  return semihosting(SYS_WRITE, params);
}
```

It is often desired to keep the output of the target separate from the other GDB output on the console. Starting with firmware 1.9, the Black Magic Probe has the "`monitor redirect_stdout`" command, which forwards writes to `stdout` & `stderr` to the secondary UART interface (see page 55 for the command). You then monitor the output using a serial terminal connected to the secondary UART. Alternatively, a front-end may also be able to separate semihosting output from GDB output, and present both in different views or windows. For example, the BMDebug front-end shows semihosting output in the "Semihosting output" view (see page 69). It does not need the "`monitor redirect_stdout`" command.

Depending on the standard libraries that you use, you may not need to implement the semihosting functions 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 an option to the linker command line, and a call to an initialization routine near the top of the `main()` function.

The option to add to the linker:

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

In the startup code (e.g. `main()`, call:

```
initialise_monitor_handles();
```

From that point on, calls to `printf()` are routed through semihosting... on the condition that a debugger is attached.

The caveat of semihosting is that if *no* debugger is attached, the software breakpoint triggers a *HardFault* exception —and typically stops the entire device in its tracks. Semihosting are therefore typically wrapped inside macros whose definition is conditional on the build: debug versus release, and you must be careful to never run a debug build outside a debugger.

An alternative is to determine at run-time whether a debugger is attached, and adjust the code `semihosting()` function to exit the function when running outside a debugger. On a Cortex M3/M4/M7 microcontroller, 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+ microcontroller architecture, the CoreDebug registers are only accessible from the JTAG/SWD interface, however. Code that runs on the M0/M0+ microcontroller cannot read `DHCSR`. 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 next instruction.

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, provided that it has the correct name. This is `HardFault_Handler()` for CMSIS, and `hard_fault_handler()` for `libopencm3`.

```
__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 */
      "mov  r0, #0 \n"      /* set error code (r0 == -1) */
      "sub  r0, #1 \n"
      "bx   lr \n"
  );
}
```

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

The *HardFault* handler sets the `r0` register to -1 when it skips a semihosting breakpoint. This allows you to check whether a debug probe is at-

tached, by invoking a semihosting operation that does not normally return -1. For example, calling `sys_iserror()` (as implemented below) would return 0 or 1 when a debug probe is attached to the target, but in absence of a debug probe, the *HardFault* handler makes it return -1.

```
int sys_iserror(int code)
{
  return semihosting(SYS_ISERROR, &code);
}
```

Error code 0 stands for "not an error," so with the parameter set to zero, function `sys_iserror()` returns 0 for "debugger attached" and -1 for "no debugger."

```
bool is_debugger_attached(void)
{
  return (sys_iserror(0) != -1);
}
```

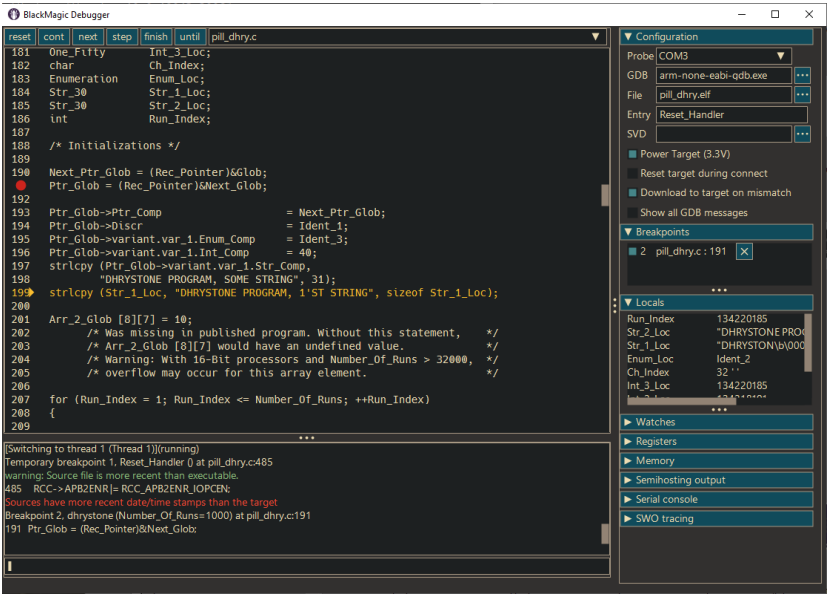
In closing, note that the semihosting interface makes it possible for firmware to read and write any files that GDB has permissions to access. It is the responsibility of the programmer to not naively run unknown firmware under a debugger —and certainly not as a root user. The mere protection that GDB offers against rogue firmware is that it will not honor `SYS_SYSTEM` operations (to run an arbitrary command on the host), unless this is specifically enabled.

```
(gdb) set remote system-call-allowed 1
```

The BlackMagic Debugger Front-end

The BMDebug utility is a front-end for GDB that is designed for the Black Magic Probe and ctxLink. It handles the Prerequisite Steps described on page 33 on start-up:

- automatically locates the debug probe and attaches to it;
- powers up the target, if this option is set in the configuration;
- verifies whether the code in the microcontroller matches the ELF file loaded in GDB, and downloads the ELF it on a mismatch (this is also an option in the configuration);
- verifies the date & time stamps of the source files against that of the ELF file, and issues a warning if the ELF file is out of date.



Apart from serving as a graphical front-end, BMDebug integrates a basic serial monitor and support for SWO tracing, so that it can combine traditional debugging with run-time tracing.

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 microcontroller 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 124), the BMDebug front-end is not able to calculate the header checksum for the LPC microcontroller 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 (LPC-specific) ELF file loaded in the debugger and the one downloaded in the target, and re-download it at every run. To avoid this, include a call to `elf-postlink` on the ELF file as part of the build process (e.g. the Makefile). See the discussion of the `elf-postlink` utility in section Verify Firmware Integrity (page 38 for more information on the checksum for LPC microcontrollers). 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), and instead use the `load` command to explicitly download new code. See also section Edit-Compile-Debug Cycle on page 55.

GDB Console and Command Line

BMDebug is a "thin" front end: it has controls and shortcuts for the basic operations of a debugger, but more advanced commands (like adding a condition to a breakpoint) need to be typed as a command. The output of those commands typically appears in the GDB console.

BMDebug has the GDB console and the command line (for input to GDB) in the bottom-left section of its main window. 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+↑ and Ctrl+↓ key combinations scroll through earlier commands on the command line, and Ctrl+R key pair searches in the command history for matching the text.

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 main view of the debugger is for the source code, and optionally the disassembly of the generated machine code. If this view says "NO SOURCE," the BMDebug front-end was unable to load the source file for the selected object file. This may be the case for code belonging to libraries that are provided only in inary form. It may also be the case that the ELF file was built on a different workstation than where it is debugged —or that the source files have moved to a different location since the ELF file was built. In these cases, you can set a path to the source files in the configuration section of the sidebar (at the right of the application window). You can also use the `directory` command, see page 40.

The source view shows the execution point with a rightward 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, using the standard keys for cursor movement (using Arrow Up/Down, Page Up/Down, Ctrl+Home and Ctrl+End). Every time the target microcontroller stops, BMDebug sets the cursor line to the execution point. Alternatively, you can also run to the cursor line with the button Until (or use function key 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 39). 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).

```
(gdb) 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.

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 text search is case-sensitive, and supports wildcards "?" and "*". 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.

Note that GDB has a `find` command as well, for searching through the target's memory for bytes or multi-byte values —see section Examining Variables and Memory on page 45. BMDebug looks at the syntax of the find command, to determine whether it should search for text in the source code, or whether to forward the command to GDB.

Running code

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

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

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

Breakpoints

You can set a breakpoint either by clicking in the left margin in the *source view*, or with function key F9, or with a `break` command in 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 in the sidebar.

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 microcontrollers provide 6 hardware breakpoints).

The `break` command (see Breakpoints and watchpoints on page 43) can also be used on the console line. The command line allows you to set temporary breakpoints and watchpoints as well.

Viewing Variables and Registers

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 "Locals" view in the right sidebar shows all local variables that are currently in scope. GDB uses heuristics (based on the variable type) to choose whether to display integer variables in decimal, hexadecimal or other. In BMDebug, you can select a different display format after right-click of the mouse on the value.

The "Watches" view in the 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 (pointer) redirections and 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 45). The BMDebug front-end handles the `display` and `undisplay` commands internally.

Standard registers of the microcontroller can be inspected in the "Registers" view in the right sidebar. Peripheral registers (and some core registers) are memory-mapped, and not in this view. Instead, BMDebug supports "System View Description" files (SVD files). These files contain the definitions of the core and peripheral registers of the microcontroller. When an appropriate SVD file is loaded, hovering over a register name in the source view shows the value of the register; likewise, you can add a watch to a peripheral register.

`info svd`	List all peripherals.
`info svd` *`peripheral`*	List all registers that are part of the peripheral, together with short descriptions of the registers.
`info svd` *`peripheral.register`*	Give documentation on the register, including the descriptions of bit fields.
`print` *`peripheral.register`*	Show the value of the peripheral register, and also decode the bit fields (if the register has bit fields).

That said, this feature depends on the source code and the SVD file to agree on the names of peripherals and registers. In practice, this means that SVD files combine neatly with CMSIS as the hardware abstraction layer, because the System View Description format is a subproject of CMSIS. The CMSIS project comes with the `SVDConv` utility that generates a C/C++ header file from an SVD file, which is how you can ensure that both the firmware and the debugger agree on the peripheral & register definitions. When using a different hardware abstraction layer, like libopencm3, SVD files may not be of much use.

Most microcontroller manufacturers provide SVD files for their microcontrollers on their websites. A collection of SVD files for various brands and series of microcontrollers is available on GitHub, see Further Information on page 182 for the link.

Viewing Assembly Code

BMDebug can show the disassembled machine code, interleaved with the source code. It uses its own disassembler (rather than the one in GDB), so that the assembly code can be annotated with peripheral and register names from SVD files (as covered above).

`assembly` **`assembly`** *`on / off`*	Switches assembly mode on or off. When used without parameters, the command toggles the mode.

disassemble **disassemble** *on / off*	The standard GDB disassemble command is redefined to be the equivalent to the assembly command.

When in assembly mode, function keys F10 and F11 step by machine instruction, rather than by source line. Specifically, F10 performs a `nexti` command in assembly mode, and a `next` command when in source more. Likewise, F11 executes a `stepi` or a `step` command, depending on the mode.

Compilers may generate instructions out of order, and they may interleave instructions from one statement with another. Small functions may be inlined, even if they are not marked as "`inline`." The GCC compiler does all of these tricks when optimization is enabled. The BMDebug front-end collects and displays all machine instructions that relate to a statement on a line. Thus, each sequence of machine instructions performs the steps for the C/C++ language statement above the sequence. However, these machine instructions do not necessarily run consecutively.

```
109
110 int main(void)
111 {
    080001c0    b5f0        push    {r4-r7, lr}
    080001c4    b087        sub     sp, #28         ; 0x1c
112   int i=0;
113   char str[20];
114
115   rcc_clock_setup_pll(&rcc_hse_configs[RCC_CLOCK_HSE8_72MHZ]);
  ▶ 080001c2    4838        ldr     r0, [pc, #224]  ; 0x80002a4 -> 0x8000a24
    080001c6    f000 fa35   bl      8000634         ; rcc_clock_setup_pll
116
117   systick_set_clocksource(STK_CSR_CLKSOURCE_AHB);
    080001ca    2004        movs    r0, #4
    080001cc    f000 fb3e   bl      800084c         ; systick_set_clocksource
118   systick_clear();
```

For example, the snippet above shows the first few instructions of a `main` function. It declares two variables and then calls a function to set up the clock and the PLL (Phase-Locked Loop). The execution point (and current line) are at that line: line 115, code addres `080001C2`. The instruction at that address ("`ldr r0, [pc, #224]`") is a 16-bit Thumb2 instruction (binary encoding is `4348`, hexadecimal). The next instruction would then be expected at address `080001C4` —yet, the instruction below it, is at address `080001C6`. There is a gap... In fact, the instruction at address `080001C4` is part of the stack frame set-up for the function, and printed as the second of two instructions for line 111. The C/C++ compiler has interleaved the instructions for the stack frame set-up with those for the first statement.

When stepping through code in assembly view, you will step by address. Thus, in this example, a step would execute the "`ldr r0, [pc, #224]`" and advance to address `080001C4` (which jumps up in the source view).

If you wish to move the current line to the next address (without execut-

ing an instruction), you can use the key combination Alt+ArrowDown. Similarly, use Alt+ArrowUp to move to the previous address. This is particularly convenient when scrolling through "calls" to inlined functions.

Viewing Memory

Viewing memory at some address that is not related to a symbol in the program, is quite common on microcontrollers. Embedded peripherals are often memory-mapped and a microcontroller may define special memory regions for buffers or queues. GDB has the "x" command that fits this purpose (see page 45). The BMDebug front-end improves on it by displaying the memory dump in a separate view, and by updating this view at each halting point. It functions like a *watch* on a memory range: bytes or words that have changed since the last refresh are colored red.

BMDebug supports the same options on the x command as GDB, but its defaults are different. Where GDB defaults to displaying a single 32-bit word, BMDebug defaults to displaying sixteen 8-bit bytes.

Note that BMDebug re-evaluates the *address* given for the "x" command at each halting points. Therefore, if the address is given as a register reference, the memory view will be updated to what the register points to. For example, with the command:

```
(gdb) x $sp /8w
```

the memory view will show the last eight words pushed to (or stored on) the stack, and if more words get pushed onto the stack, or popped off the stack, the memory view follows the address in register "sp."

Trace Views

Three trace views are provided: one for semihosting output, one for a serial monitor, and one for SWO tracing. See chapter Run-Time Tracing on page 74 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 serial monitor and 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 to GDB. Both the serial monitor and the SWO tracing view support The Common Trace Format (see page 92), for tracing with reduced overhead.

Serial monitor commands

serial *port bitrate* **serial** *bitrate*	Open the serial port at the given bitrate (Baud), to monitor received data. If the port name is omitted, the command uses the secondary TTL-level UART of the Black Magic Probe. The protocol settings that the serial port are set to, are: 8 data bits, 1 stop bit, no parity.
serial *filename*	Set the metadata file for the Common Trace Format (see page 92). The metadata file is in TSDL format. When a metadata file is set, incoming serial data is interpreted as Common Trace Format packets.
serial clear	Clear the viewport of the serial monitor (deletes all received text).
serial disable	Disable the serial monitor, closes the serial port.
serial enable	Open the serial monitor with the most recent settings (for port and bitrate).
serial info	Show the current configuration.
serial plain	Disable the Common Trace Format decoding and unload a previously loaded TSDL metadata file.
serial save *filename*	Save the contents of the serial monitor into a file. When Common Trace Format is active, the file is in CSV format (compatible with BMTrace). Otherwise, it is stored as a plain text file.
serial severity *level*	Set the severity level for the trace messages. This is only relevant if Common Trace Format decoding is active. The levels are: debug, info, notice, warning, error and critical. After setting a severity level, all messages at or above that level are shown, and all messages below that level are hidden.
serial stream *name* **enable** **serial stream** *name* **disable** **serial stream** *name* **exclusive**	Enable or disable a stream of the Common Trace Format. You can get a list of the stream names with "serial info" (assuming Common Trace Format is active). To enable (or disable) all streams, use "*" for the name. The "exclusive" option enables the selected stream, and disables all others.

On the topic of SWO tracing: note that while the BMDebug front-end supports both Manchester encoding and asynchronous encoding, the debug probe determines which of the two you can use. At the time of writing, the native Black Magic Probe supports only Manchester encoding;[5] the ctxLink probe supports only asynchronous encoding.

SWO tracing commands

[5] Version 2.3 of the hardware is designed to also support asynchronous encoding, but firmware support is still pending.

trace *clock bitrate*	Enable tracing in Manchester encoding.
trace passive	If the target microcontroller's clock 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. Use this option if the firmware of the target configures SWO tracing itself (in code). The parameter "passive" may also be written "pasv".
trace async *clock bitrate* **trace async passive** *bitrate*	Enable tracing in Asynchronous encoding with the given clock of the target microcontroller 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 clear	Clear the viewport for the SWO trace messages (delete all received messages).
trace disable	Disable SWO tracing.
trace enable	Enable SWO tracing using previously configured settings.
trace 8-bit **trace 16-bit** **trace 32-bit** **trace auto**	Set the width of the data in an SWO tracing packet (in relation to trailing-zero compression). This value must match the value that the target uses. The ubiquitous implementation is for 8-bit data width (which is the default setting). When the parameter is auto, the debugger derives the data width from the incoming data. See the notes on "zero compression" on page 11 for more information.
trace *filename*	Set the metadata file for the Common Trace Format (see page 92). When no file is explicitly set, BMDebug looks for a file with the same base name as the ELF file and a ".tsdl" extension, and it searches for it in the same directory as the ELF file, as well as in the directories where the source files are.
trace plain	Disable the Common Trace Format decoding and unload a previously loaded TSDL metadata file.
trace channel *index* enable **trace chan** *index* enable **trace ch** *index* enable	Enable the display of the given channel (range 0..31).

trace channel *index* disable **trace chan** *index* disable **trace ch** *index* disable	Disable the 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* #color **trace chan** *index* #color **trace ch** *index* #color	Set the background color of the channel marker. The color 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.
trace save *filename*	Save the contents of the trace monitor into a file, and in in CSV format (compatible with BMTrace).
trace severity *level*	Set the severity level for the trace messages. This is only relevant if Common Trace Format decoding is active. The levels are: debug, info, notice, warning, error and critical. After setting a severity level, all messages at or above that level are shown, and all messages below that level are hidden.

BMDebug saves files for SWO tracing in the same format as the one used by the stand-alone trace viewer BMTrace trace viewer (page 85). You can therefore analyze trace data captured in a debug session with this utility. BMTrace offers more filtering and bookmarking options, which may make the analysis of a large trace file easier. Data captured in the serial monitor is saved in a format compatible with BMTrace as well, provided that CTF decoding was active (on the serial monitor). If the serial monitor operated in "plain" mode, the captured data is saved in text format.

Session data

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." When opening an ELF file in BMDebug, it automatically reloads the settings from this configuration file. Therefore, to enable SWO tracing and restore all settings and channel configurations from a previous session, the following command is sufficient:

```
(gdb) list trace enable
Active configuration: Manchester encoding, passive, data width = 8-bit
```

Help and info

BMDebug adds a few topics to the `help` and `info` commands —see Getting help and information on page 39 for these commands. When typing `help`

without parameters, these topics are listed under the sub-head “Front-end topics.”

Run-Time Tracing

The standard "stop & stare" style of debugging, where you step through code one line at a time, may not be suitable for an embedded system. When the code hits a breakpoint or is in "step"-mode, the microcontroller stops, and this may be *too little* or *too much* (even both at the same time). The microcontroller 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 microcontroller does *too little* in this case: it does not stop the linear actuator, but it also does *too much*: it blocks the ISR that handles the safety end stop from running.

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

This chapter has an overview of the various methods for tracing that the Black Magic Probe offers. Each of these has its own advantages and disadvantages. The next chapter then 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 microcontrollers provide for three trace sources:

- Instruction trace, which creates a log of every instruction executed by the microcontroller. 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* unit (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 90) 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 provides a TTL-level UART (as its secondary virtual serial interface). If the target board as the TxD and RxD lines of a UART branched out of the microcontroller, 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 or monitor.

Sending trace messages over a UART is a boilerplate technique, because it works everywhere: all microcontrollers 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 requires 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.[1]

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

[1] This refers to the number of pins on the microcontroller. With regard to the wiring, the ground wire must be connected in addition to the TxD and (optionally) RxD pins. When using the secondary UART of the Black Magic Probe, the device's power should normally be connected to the VCC pin of the UART connector of the Black Magic Probe —see page 148.

Semihosting

Semihosting is covered in section Semihosting: target to host I/O bridge on page 57. Writing messages to the debugger console via semihosting is a common tracing method.

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 the next section for SWO Tracing).

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 `sys_write0()` or `printf()`. Source code that implements semihosting calls for GCC (and not depending on `newlib`) is provided in the "examples" subdirectory in the archive accompanying this book.[2]

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 26). The ARM Cortex M0 and M0+ architectures lack support for SWO tracing, but see section SWO Tracing on the Cortex M0/M0+ on page 82 for a workaround.

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. By convention, the last channel (channel 31) is reserved for use by an RTOS. 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.

[2] Also available at https://github.com/compuphase/Black-Magic-Probe-Book.

```
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 10), 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 in reality sends two: a header byte and a payload byte).

A more flexible and efficient function is below. It starts by checking that 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 test drops through, it collects up to 4 characters from the message into a 32-bit word, before storing it in the FIFO of the *Instrumentation Trace Macrocell* (ITM). The FIFO is accessed via the register `PORT`, which is in fact an array of 32 registers. Before storing every next packet in the FIFO for the trace subsystem, the function waits in a while loop until the FIFO has space to hold the new 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)
                    {}    /* null statement */
                ITM->PORT[channel].u32 = value;
                value = shift = 0;
            }
        }
        /* transmit last collected characters */
        if (shift > 0) {
            while (ITM->PORT[channel].u32 == 0UL)
```

```
                {}
            ITM->PORT[channel].u32 = value;
        }
    }
}
```

The `PORT` register allows 8-bit, 16-bit and 32-bit accesses, and this relates to the trailing-zero compression used by the SWO Trace protocol (again, see section TRACESWO Protocol on page 10). In fact, the implementation in the above snippet could be optimized a little further still: when transmitting the last collected bytes, it now always sends a 32-bit payload —due to the assignment to `PORT[].u32`.

For trace viewers, zero compression adds the complexity that on reception of a packet with a 1-byte or 2-byte payload, there is no automatic way to know whether it should possibly be expanded to a 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 92), which uses a binary stream.

SWO Tracing must first be configured in the microcontroller, which can be done either in the firmware (i.e. source code), or via the debug probe. 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 microcontrollers require additional device-specific configuration that is not standardized by ARM. Configuring the tracing in the source code (at least partially) may therefore be unavoidable.

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). 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.

The asynchronous protocol generally allows for higher bit rates. The clock speed cannot be recovered from the data stream though, so for asynchronous encoding, 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 the Black Magic Probe (and clones that use the STM32F10x microcontroller), the bit rate must be 4.5 Mb/s divided by an integer value, and with a maximum of 2.25 Mb/s.[3]

The choice between the two protocols may be limited by the debug probe. The native Black Magic Probe supports only Manchester encoding,[4] while the ctxLink probe (and a few other derivatives of the Black Magic Probe) instead support asynchronous encoding exclusively.

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

```
void trace_init(int protocol, uint32_t bitrate, uint32_t channelmask)
{
    uint32_t clockfreq =  (protocol == 1) ?  2 * bitrate : bitrate;

    CoreDebug->DEMCR = CoreDebug_DEMCR_TRCENA_Msk;

    TPI->CSPSR = 1;           /* protocol width = 1 bit */
    TPI->SPPR = protocol;     /* 1 = Manchester, 2 = Asynchronous */
```

[3] Running 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.

[4] Hardware version 2.3 of the Black Magic Probe is adapted to support asynchronous encoding as an option, but firmware support is "pending" at the time of writing.

```
    TPI->ACPR = CPU_CLOCK_FREQ / clockfreq - 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) */
}
```

Parameter "`protocol`" must be 1 for Manchester encoding, or 2 for asynchronous encoding. Parameter "`channelmask`" is a bit mask where a "1" bit enables the respective channel. Note that for Manchester encoding, the clock frequency is twice the bit rate, because there may be transitions halfway the bit period.

An extra device-specific initialization step must often precede the generic initialization. Examples for a few microcontroller series are below. Note that some microcontrollers 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 */
}
```

If `AFIO_MAPR_SWJ_CFG_1` is not defined in the device header file of your development suite, note that it is "`(2 << 24)`."

STM32F4xx series[5]

```
void trace_init_STM32F4xx(void)
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN; /* enable GPIOB clock */
    GPIOB->MODER = (GPIOB->MODER & ~0x000000c0) | 0x00000080; /*PB3 alt func*/
    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 */
}
```

[5] Adapted from the GDB scripts of the Orbuculum project.

SAM D5x series[5]

```
void trace_init_SAMD5x(void)
{
    /* enable peripheral clock on GCLK_CM4_TRACE */
    GCLK->PCHTRL[47] = GCLK_PCHCTRL_GEN(0) | GCLK_PCHCTRL_CHEN;
    /* set PB30 to SWO */
    PORT->Group[1].PMUX[15].bit.PMUXE = PORT_PMUX_PMUXE(7);
    /* enable PMUX for PB30 */
    PORT->Group[1].PINCFG[30].bit.PMUXEN = 1;
}
```

LPC13xx series

```
void trace_init_LPC13xx(void)
{
    LPC_SYSCTL->TRACECLKDIV = 1;
    LPC_IOCON->PIO0_9 = 0x83;     /* func 3, no pull-up/down */
}
```

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;    /* func 2, no pull-up/down, digital */
}
```

LPC5411x series

```
void trace_init_LPC15xx(void)
{
    LPC_SYSCTL->TRACECLKDIV = 0;
    LPC_SYSCTL->SYSAHBCLKCTRLSET = 1 << 13;
    LPC_IOCON->PIO0_15 = 0x82;    /* func 2, no pull-up/down, digital */
}
```

LPC546xx series

```
void trace_init_LPC15xx(void)
{
    LPC_SYSCTL->TRACECLKDIV = 0;
    LPC_SYSCTL->SYSAHBCLKCTRLSET = 1 << 13;
    LPC_IOCON->PIO0_10 = 0x306;   /* func 6, digital, filter off */
}
```

SWO Tracing on the Cortex M0/M0+

The ARM Cortex M0 and M0+ architectures lack support for SWO tracing. While you still have the option for tracing via a UART or semihosting, if you want to use a uniform debugging environment for all ARM Cortex microcontrollers, it may be worthwhile to emulate SWO tracing on Cortex M0/M0+.

• *Emulating asynchronous mode*

When using a ctxLink or another debug probe that supports asynchronous mode, the first step in emulating SWO is to wire the TxD pin of the UART to TRACESWO on the debug connector. The function to transmit the trace messages must be adapted to add a header byte in front of each packet (as explained in section TRACESWO Protocol on page 10). In a nutshell, an SWO packet can have a payload 1, 2, or 4 bytes, so there is a header byte for every sequence of payload. Obviously, it must also store the data (header bytes plus payload) in the UART FIFO instead of in the ITM FIFO.

The function ARM_USART_Send that is used in the snippet below is appropriate for the Keil implementation of CMSIS (and perhaps others); you may need to replace it with an equivalent function when using another UART driver library.

```
void trace(int channel, const unsigned char *data, unsigned size)
{
    if (TRACESWO_TER & (1 << channel)) {      /* if channel is enabled */
        while (size >= 4) {
            uint8_t header = (channel << 3) | 3;
            ARM_USART_Send(&header, 1);
            ARM_USART_Send(data, 4);
            data += 4;
            size -= 4;
        }
        if (size >= 2) {
            uint8_t header = (channel << 3) | 2;
            ARM_USART_Send(&header, 1);
            ARM_USART_Send(data, 2);
            data += 2;
            size -= 2;
        }
        if (size >= 1) {
            uint8_t header = (channel << 3) | 1;
            ARM_USART_Send(&header, 1);
            ARM_USART_Send(data, 1);
        }
    }
```

```
}
```

In this implementation, the global variable `TRACESWO_TER` takes over the role of the "Trace Enable Register" (`TER`) of the ITM. It must be declared as a 32-bit integer, and I recommend that it is initialized to zero. This way, when running the firmware outside a debugger, all traces drop out immediately, but when running under GDB (or a trace viewer that uses the `gdbserver`), the debugger can set this variable to a non-zero value and enable the trace channels. The BMTrace trace viewer (page 85) and BMDebug front-end (page 62) check for a variable with the name "`TRACESWO_TER`" and configure it automatically when enabling or disabling channels from the user interface.

• *Emulating Manchester mode*

At the time of writing, the native Black Magic Probe only supports Manchester mode for SWO tracing. The obvious recourse is to emulate Manchester mode via bit-banging, but that is slow, and to keep within the timing constraints, the bit-banging routine must run with interrupts disabled. The combination of the two: slow code that runs with interrupts disabled, carries a risk that interrupts are not responded to quickly enough, or even that they are missed altogether.

There is yet a way to implement hardware-supported SWO emulation on a Cortex M0/M0+, if you have a spare SPI interface on your microcontroller. The trick is to expand each bit that is transmitted to a two-bit sequence: a `1` to "`10`" and a `0` to "`01`", and then transmit these through over the `MOSI` line. This expansion can be efficiently done per 4 bits with a 16-byte lookup table. This same lookup table inverts the bit order: the SPI protocol transmits the most-significant bit first, whereas the SWO protocol (with Manchester encoding) transmits the least-significant bit first.

```
static const uint8_t manchester_lookup[16] = {
    0x55, /* 0000 -> 0101 0101 */
    0x95, /* 0001 -> 1001 0101 */
    0x65, /* 0010 -> 0110 0101 */
    0xa5, /* 0011 -> 1010 0101 */
    0x59, /* 0100 -> 0101 1001 */
    0x99, /* 0101 -> 1001 1001 */
    0x69, /* 0110 -> 0110 1001 */
    0xa9, /* 0111 -> 1010 1001 */
    0x56, /* 1000 -> 0101 0110 */
    0x96, /* 1001 -> 1001 0110 */
    0x66, /* 1010 -> 0110 0110 */
    0xa6, /* 1011 -> 1010 0110 */
    0x5a, /* 1100 -> 0101 1010 */
```

```
    0x9a, /* 1101 -> 1001 1010 */
    0x6a, /* 1110 -> 0110 1010 */
    0xaa, /* 1111 -> 1010 1010 */
};

#define M_EXPAND(buffer, byte) \
    ( (buffer)[0] = manchester_lookup[(byte) & 0x0f],        \
      (buffer)[1] = manchester_lookup[(uint8_t)(byte) >> 4] )
```

Apart from the bit expansion, the routine to emulate Manchester encoding is similar to the one that emulates asynchronous encoding for SWO tracing, on page 82, so it is not repeated here. A separate implementation (source code file) is provided among the example files with this book.

Monitoring Trace Data

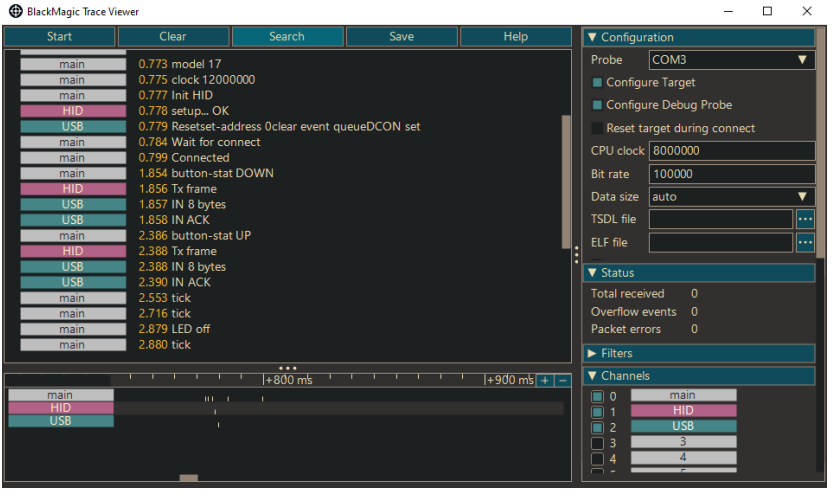
As of version 1.7 of the firmware of the Black Magic Probe, you can redirect the SWO trace data to the virtual UART. The upside is that you only need a serial terminal to view the trace data, and there are many to choose from. However, there are downsides too: channel information is not preserved, and filtering of enabled channels happens in the Black Magic Probe, instead of in the target. Furthermore, and probably a minor point, the Black Magic Probe has only a single UART interface, so you cannot use both the UART and trace redirection at the same time. See the `traceswo` command on page 51 for more information.

To elaborate on the limitations of the transmission of SWO trace data over the UART, there are two practical uses for the channel information. The first is to separate the messages in different lists, or mark them in different colors. The second is to reduce the performance impact of tracing, by disabling the channels that are not relevant in the context of a particular debug session. Trace data that is *not* transmitted implicitly has minimal overhead. When you let the Black Magic Probe transmit the SWO trace data over the UART, it allows you to enable or disable channels; however, it is the Black Magic Probe that filters out the disabled channels. The target still transmits all trace data for *all* channels to the Black Magic Probe. So you are still subject to the full performance penalty of the SWO transmission —and especially so in the case of Manchester encoding.

The alternative is to capture the SWO trace data directly. This requires a special tool or viewer. An advanced set of tools is the Orbuculum project, which was mentioned earlier. 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 postprocessing 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. Orbuculum runs on Linux, macOS and Microsoft Windows.

A stand-alone graphical SWO trace viewer for the Black Magic Probe is the BlackMagic Trace Viewer —or BMTrace. 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 microcontrollers that it supports. Another distinctive feature of BMTrace is that it supports the Common Trace Format, see page 92.



As described earlier, SWO tracing can use either modes Manchester or Asynchronous, but most variants of the Black Magic Probe support only one of these (and not both). If BMTrace detects that the selected debug probe is a native Black Magic Probe, it sets Manchester mode; likewise, if it detects the ctxLink probe, it sets Asynchronous mode. For other Black Magic Probe variants, you must select the mode in the configuration of BMTrace.

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 microcontroller 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 79. 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 and does not connect to Black Magic Probe's `gdbserver` (it connects only to the separate USB endpoint for SWO tracing). 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 color 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 92), the names of the channels are overruled by the "stream" names that are defined in the metadata file for the traces.

Apart from filtering on channels, BMTrace allows filtering incoming messages on keywords in the text. If no filters are set, all messages are shown; if one or more filters are set, only the messages that match any of these filters are shown. Note that content filtering is done by the viewer, unlike filtering on channels —so it does not reduce tracing overhead on the target device.

Text matching in a filter is case-sensitive. However, it supports the following wildcards:

- **?** matches a single arbitrary character;
- ***** matches any number (including zero) of arbitrary characters;
- **/** matches punctuation or whitespace, as well as the beginning or the end of a string. This wildcard is useful for matching a word delimiter.

If the filter text starts with a "¬", the filter is inverted: the message is *not* shown if it contains the keyword (behind the *not sign*). The tilde ("~") may be used as an alternative to the *not sign* ("¬"), since not every keyboard layout provides a key combination for this symbol. Each filter can be enabled or disabled, for quickly toggling them on or off.

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 *if* the incoming trace data is in the Common Trace Format *and* timestamps are present in the CTF stream, BMTrace shows these embedded timestamps instead. These

timestamps are generated on the target, and they are generally more accurate.

Real Time Transfer (RTT)

Real Time Transfer (RTT) is a bidirectional communication protocol developed by SEGGER Microcontroller. The communication runs over the debugging interface —SWD in the case ARM Cortex, so it requires no extra pins. It *does* require support code in the firmware, though.

RTT works by having the firmware store its output into a queue in RAM. The debug probe then reads this queue at a regular interval, and forwards it to the host. A basic function for any embedded debug protocol is to be able to access all the memory of the target device. In the case of the ARM CoreSight architecture, the debug subsystem runs independently from the microcontroller core. RTT builds on this to be able to read the queue without affecting normal code execution.

From the perspective of the firmware, storing a character in a queue (or "ring buffer" as the RTT documentation calls it), is very fast. This minimizes the slowdown that run-time tracing has on the code —until the queue is full. If that happens, RTT offers a choice between stalling until the debug probe empties the queue, and simply dropping all data that no longer fits. Neither option is attractive. In practice, trace messages often come in bursts. One can take advantage of this by defining the queue big enough to hold a burst of messages without overflowing (after which the probe reads and empties the queue at its own pace).

On the target device, code must be added to declare the data structure for the queue, as well as functions to push data into the queue. The canonical implementation in C is provided by SEGGER, at no cost and with a very liberal license. It only takes a few lines to add RTT to your code.

```
#include "SEGGER_RTT.h"

int main(void)
{
    SEGGER_RTT_Init();

    /* ... */

    SEGGER_RTT_WriteString(0, "Hello World\n");

    /* ... */
}
```

At the same time, this snippet only scratches the surface of RTT's functionality. RTT allows for multiple channels, which can be both for input, as for output. The first channel (numbered zero) is predefined, for both input and output —although you can override the default queue sizes.

The data structure with the channel definitions consists of a header with a signature, followed by a list of fields for each possible channel. In RTT, this data structure is called the "control block." You can print out this structure with a `monitor` command.

```
(gdb) monitor rtt cblock
cbaddr 0x20000728
ch ena i/o buffer@      size   head   tail flag
 0   y out 0x200000b0   1024    977    977    2
 1   y out 0x00000000      0      0      0    0
 2   n out 0x00000000      0      0      0    0
 3   y in  0x000000a0     16      8      0    0
 4   n in  0x00000000      0      0      0    0
 5   n in  0x00000000      0      0      0    0
```

The above list represents a default configuration. By default, there is a maximum of three output channels and three input channels, but only the first two output channels and the first single input channels are enabled. Furthermore, although output channel 1 is enabled, there is no buffer attached to it, so it is effectively non-functional until it is explicitly initialized in the firmware. The maximum number of output and input channels is fixed at compile time, but queues (buffers) must be assigned at run time (with the exception of the first output and the first input channel). Also note that the definition blocks for the input channels always follow those of the output channels, and while the channels are numbered sequentially, in the RTT API input channel numbering restarts at zero.

The signature (or "ident string") is not displayed by the above command, but if you dump the memory at the returned address for the "`cbaddr`" structure, it will start with the signature strings —the default is "`SEGGER RTT`". The debug probe uses the signature to scan memory (of the target) for the RTT control block. In fact, before the above command outputs the table with the channel configurations, RTT must first have been enabled (the Black Magic Probe only scans for the RTT control block after RTT is enabled).

There are two caveats with the scan for the RTT control block. Firstly, if the signature has been changed from the default in the configuration file for the firmware, the signature must be set before enabling RTT:

```
(gdb) monitor rtt ident secret_trace
```

Secondly, the probe scans the memory range that is recorded in the *driver* for the target microcontroller. However, manufactures often create a microcontroller in several variants, which differ only in the amount of Flash ROM and SRAM. The Black Magic Probe uses a single driver for these variants, and records the memory ranges of the largest variant. The upshot is that when you are using a smaller variant, the memory range that Black Magic Probe has recorded for it (and which you can see with an "`info mem`" GDB command) is too large. If you then enable RTT, and the signature is not found in the valid memory range, the debug probe will continue to search in non-existing RAM —which may then cause a hard fault exception, and hang the firmware. To avoid this, you can explicitly set the memory address range for the RTT scan:

```
(gdb) monitor rtt ram 0x20000000 0x20004000
```

See also the section Real Time Transfer on page 52 for more GDB `monitor` commands, related to RTT.

At the time of writing, only the Jeff Probe supports RTT in its officially released firmware. The Black Magic Probe includes RTT support in firmware version 1.9, but only in source form. Hence, you will need to compile the BMP firmware yourself (with `ENABLE_RTT=1` on the `make` command line). The technology behind RTT is patented,[6] which may be why the black-magic project decided to distribute the implementation as source only. Please see the black-magic project for instructions to build the firmware; see chapter Further Information on page 182 for a link.

The Black Magic Probe forwards the RTT data to its secondary virtual serial interface. To view the output, only a serial terminal is needed. In the serial terminal, you can set any baud rate (and data & stop bits) for opening the connection, because the virtual interface ignores these settings.

When RTT is active, the secondary serial interface is shared with the real TTL-level UART on the Black Magic Probe. If this serial UART is connected to the target, and activated, data coming on the UART are mixed with the RTT data. Data that you transmit on the serial interface, will be directed to the RTT input queue, if RTT is active.

A claimed advantage of RTT, is that you can leave it in production code. Code for semihosting and SWO tracing must be removed from the production builds, or be implemented in such a way that they detect whether a debugger or trace viewer is attached (and quickly exit if otherwise). UART tracing is likely to slow down the code noticeably. RTT has none of these

[6] US patent US9384106B2, *Real time terminal for debugging embedded computing systems.*

disadvantages: it has low overhead for the microcontroller and it is, by default, implemented as non-blocking.

There is, however, a caveat: if the microcontroller enters "sleep mode" by executing a WFI or WFE instruction, RTT will often stop functioning correctly —see topic RTT capture in the Troubleshooting chapter for an explanation. For devices that, in normal operation, use sleep mode for power saving, you should disable this when tracing. This might be done by having a "release" build and a "debug build," or with run-time detection of whether a debugger is attached.

So on the one hand: yes, you can leave RTT in production builds without causing harm; but on the other hand, this does still *not* mean that you can run the same code, irrespective of whether tracing is being *captured* (with a debug probe).

Tracing with Command List on Breakpoints

Setting and using breakpoints is covered in section Breakpoints and watchpoints (page 43). 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 "drops through."

For example, consider a command list with only 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
```

On setting a breakpoint (on hypothetical line 121), GDB responds that breakpoint number 3 was set. When adding a command list, we will therefore have to repeat this identifier.

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, a `print` command to show the values of specific variables, or a `backtrace` command to show the call stack that lead to the breakpoint being reached.

As drop-through breakpoints are very convenient for impromptu tracing of a variable or function parameter, GDB offers the "`dprintf`" command specifically for this purpose. The `dprintf` command sets a breakpoint and attaches a formatted print function to it.

```
(gdb) dprintf 121, "Iteration %d, LED state %d", i, led_state
Dprintf 3 at 0x3ce: file blinky.c, line 121.
```

The first parameter of the `dprintf` command is the breakpoint location. It can be a line number, a function name or a filename : line-number specification, as described in the section Breakpoints and watchpoints, page 43. The second parameter is a format string, which uses the same syntax as that of the C/C++ "`printf`" function.

On hitting the breakpoint, GDB will evaluate any parameters behind the format string, and display them accordingly within the format string. After printing the text, it will immediately continue running.

The Common Trace Format

As explained in the chapter on Run-Time Tracing (page 74), 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" (or more precisely "*latency*") are:

- The need to format the data into a trace message 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, and any I/O overhead in accessing it.

When it comes to avoiding the probe effect, there is a prevalent fixation on the last point, the speed of the transfer interface. Yet, it is obvious that no matter how well you've optimized `sprintf`, skipping it entirely will always be quicker; like it is obvious that transmitting a few bytes is quicker than transmitting many (under equal conditions). Possibly, higher transfer speeds were the easiest goal to achieve in the early days, and perhaps as a corollary to the *Law of the Hammer*,[2] the reflex is to search for a bigger hammer if the current one won't do anymore.

Both the other two points (avoiding message formatting on the *target* and minimizing the amount of data that must be transferred), are addressed by the Common Trace Format (CTF). 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 microcontroller, and it also skips transferring text strings when it can instead reference these strings in the metadata file. The CTF specification is maintained by the Diagnostic and Monitoring workgroup (DiaMon) of the Linux Foundation.

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

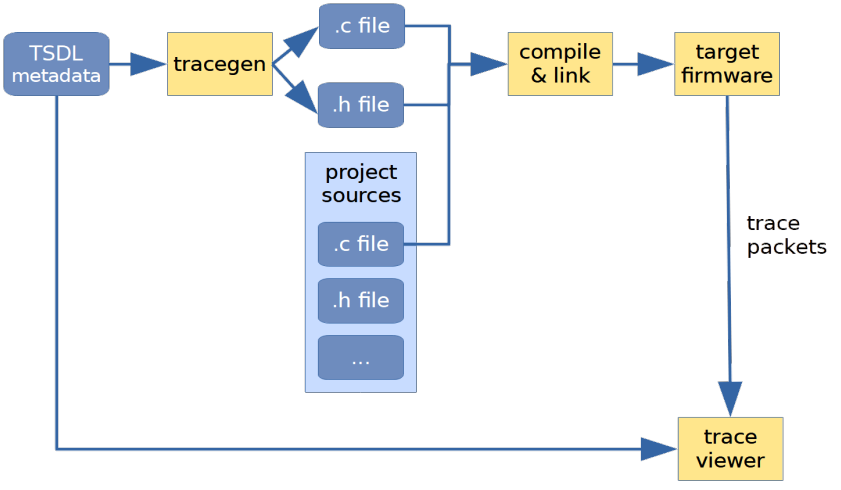
[1] J. Gait; *A probe effect in concurrent programs*; Software: Practice and Experience; March 1986.

[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]

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.

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.

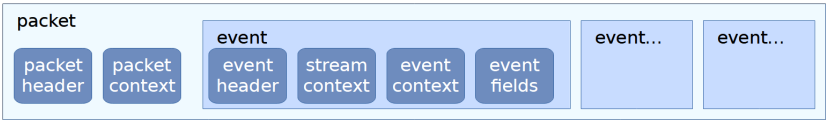
Two tools exist that generate OS-independent C code for CTF: `barectf` by the same authors as CTF, and tracegen (which is a companion tool to this book). Both tools use the metadata to generate individual C functions to build a binary CTF "packet" for each particular trace event. The generated file is 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 reference to "tracegen" could also be `barectf`. 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. The "trace viewer" in the diagram may be either BMTrace (the BlackMagic Trace Viewer, see page 85) or another CTF compatible viewer, like Trace Compass.

Binary Packet Format

The Common Trace Format sends trace messages in packets. Each 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 book focuses 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 contains data that applies to all events in the stream, whereas the event context holds 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. The full specification of this declaration language is on the DiaMon site, see the chapter Further Information on page 182 for a link.

A minimal example for a specification file is below. It defines a single event called "`peltier-plate`," with a 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
```

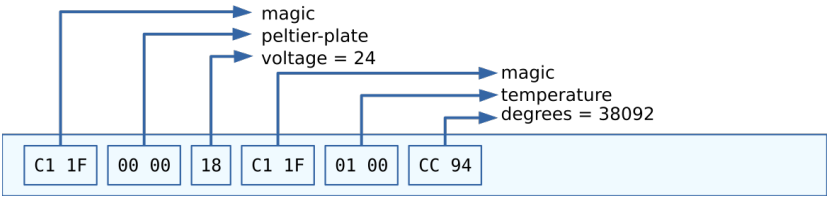
Merely a single byte does it take, to send a descriptive parametrized event: it does not get much more compact than that. However, this is an exceptional case. When there is more than one event, an event header is needed so that the various events can be distinguished. This leads to the need for a packet 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.

```
trace {
    major = 1;
    minor = 8;
    packet.header := struct {
        uint16_t magic;
    };
};

stream {
    event.header := struct {
        uint16_t event.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 flexible `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 byte stream `C1 1F 01 00 CC 94` would be displayed as:

```
temperature: degrees = 37.199
```

TSDL knows the standard C types `char`, `int`, `short`, `long`, `float` and `double`, along with their `unsigned` variants. It also inherits the annoyance of C that the size and byte-order ("endianness") of these types are implementation-defined. The tracegen utility uses the following defaults:

`char`	8-bits, signed
`int`	32-bits, little-endian
`short int`	16-bits, little-endian
`long int`	32-bits, little-endian
`float`	single precision IEEE-754, 32-bits
`double`	double precision IEEE-754, 64-bits
`enum`	"`int`"-size by default (typically 32-bits).

The tracegen utility also predefines the most common "fixed width" integer types typically found in `stdint.h`, like `int16_t` and `uint32_t`. The `bool` type is also predefined (as an 8-bit unsigned integer, which should only hold the values 0 or 1). All predefined types can be overruled (with a `typealias`), if so desired.

An enumeration is an integer type, where labels are attached to specific values. The advantage of specifying a parameter type as an enumaration, is that the trace viewer can show the label, instead of a magic value.

```
enum RequestTarget : uint8_t {
  DEVICE,
  INTERFACE,
  ENDPOINT,
  OTHER,
};
```

Like the C/C^{++} enumerations, the first label gets the value zero by default, and the value is incremented for each subsequent label. This can be overruled by setting a label to a value explicitly. Unlike C/C^{++}, the integer size of the enumeration data type can be explicitly set —in the above example, it is set to 8-bit unsigned.

For zero-terminated text strings, you use the special type "`string`" in TSDL. The tracegen utility converts it to "`const char*`" in the trace support files that it generates, see page 80. The default encoding for strings is UTF-8; though you can set the encoding to ASCII, this is rarely useful —UTF-8 is fully compatible with plain ASCII, and for extended ASCII you would need to know which codepage to use for the upper 128 characters (you cannot set the codepage in TSDL).

For general-purpose trace message output, you could therefore create a definition like the one below:

```
event debug_message {
    fields := struct {
        string msg;
    };
};
```

This creates a function that transmits a message in a zero-terminated string, and that message can be anything. If you are used to "`printf`"-style tracing (or methods derived from that), it might be tempting to implement that as the only event, and call it from everywhere with preformatted strings. However, this does away with the principal advantage of CTF: efficiency, due to a compact encoding. In other words, if you are using CTF in this way, it might not be worth the trouble to use CTF at all.

General structure

At the global level are the declarations of the trace context, the streams, the events, plus any type definitions. The `trace` context includes the packet header; each `stream` declaration includes an event header; and each `event` may contain a declaration of one or more fields. In the generated C code, each event declaration is translated to a function, and the fields are translated to parameters.

Every declaration starts with a keyword, like `trace`, `stream` or `event`. The definition of the attributes of the declaration then follow between curly braces: the declaration *block*. For the `stream` and `event` keywords, a name may optionally be present; this is the name of the stream or the event (this is a tracegen extension).

TSDL supports comments in C^{++}-style syntax; either between "`/* ... */`" tokens or following "`//`" up to the end of the line. Formally, each metadata file should start with the comment "`/* CTF 1.8 */`", but neither Babeltrace nor `tracegen` identify this special comment.

Trace context

The trace context is a block at the global level, that is introduced with the `trace` keyword. It may contain the following fields inside its declaration block:

`major`	Major version of the CTF specification.
`minor`	Minor version of the CTF specification.

`version`	The version in "major.minor" format; the current version is 1.8. This is a tracegen extension, and it can be used as an alternative to the `major` and `minor` fields.
`uuid`	The UUID (in string format) may be used to ensure that a received packet matches the metadata file (the UUID should then also be included in the packet header).
`byte-order`	Either `le` or `be` for Little-Endian and Big-Endian respectively. Little-Endian is the default.
`packet.header`	The declaration of the packet header, see the next section.

See the snippet on page 95 for an example of a `trace` context specification.

Packet header

The packet header is nested inside the `trace` context (the `packet.header` structure). It contains the *type definitions* for one or more of the following fields (in any order):

`magic`	Must be declared as a 1-byte, 2-byte, or 4-byte unsigned integer. The purpose of this field 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 value 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.

See the snippet on page 95 for an example; the `packet.header` specification is inside the `trace` context block.

Stream Declaration

A trace specification may contain one or more streams. Since we need to include a unique ID for each event, and this ID is declared in the *event header*, which in turn is declared in a stream... there is typically at least one stream declaration.

The stream declaration is a block at the global level, that is introduced with the `stream` keyword. It may contain the following fields:

`id`	The unique numeric identifier for the stream. Both tracegen and `barectf` can auto-number streams, so this field is optional.
`name`	A unique name for the stream. This is a tracegen extension.
`event.header`	The declaration of the event header, see below.

If there is only a single stream, it needs neither an ID nor a name. However, in practice, the `event.header` field should always be present.

When using tracegen, the name of the stream may also appear between the keyword `stream` and the opening brace "`{`." For example, the definition:

```
stream PIDcontroller {
    /* other fields */
};
```

is equivalent to the regular form:

```
stream {
    name = PIDcontroller;
    /* other fields */
};
```

Event header

The event header is nested inside the `stream` declaration (see the previous section, specifically the `event.header` structure). It contains the *type definitions* for one or more of the following fields:

`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 Timestamps further down on this page).

Event Declaration

Event declarations are converted to C functions that send trace packets by the utilities tracegen and `barectf`. An event declaration includes the event name and parameters, plus optionally the stream it is part of and attributes.

`id`	The unique numeric identifier for the event. Both tracegen and `barectf` can auto-number streams, so this field is optional.

`name`	A unique name for the event.
`stream.id`	The numeric ID of the stream that the event belongs to. Redundant if the trace information uses only a single stream. When using tracegen, you may also specify the stream *name* rather than the numeric ID. This field may also be called "`stream_id`" for compatibility with other CTF implementations.
`fields`	The declaration of a structure for the parameter names and types (if any).
`attribute`	Optional GCC attributes that will be included on the C function that tracegen generates.

The event name and (if relevant) the stream name, may be put between the `event` keyword and the opening brace "`{`" of the declaration. If both a stream name and an event name are present, the two should be separated with a double colon ("::").

Timestamps

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

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

Scaling up: multiple streams, many events

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

The tracegen utility extends TSDL by allowing a stream to have a name, so that an event can identify its stream by its name rather than a numeric constant. It also supports automatic numbering of streams and events (`barectf` also supports auto-numbering). For brevity in the TSDL file, the names of a stream and of an event can be placed immediately following the `stream` or `event` keywords, see the snippet below for examples. In the case of an event, the name of the stream may be prefixed to the event name, with a double colon between the two names.

Below is the example from page 95, extended with a stream name and using 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 event.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`" belonging 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` field in the `packet.header` structure 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/BMDebug utilities. When using a different trace viewer, the basic TSDL syntax (as specified on the site of the DiaMon workgroup) should be used.

Another feature that is unique to tracegen is its support for including files. It enables you to split off parts of a larger metadata files into separate files. One example of this is to keep type definitions that you use in multiple projects in a separate file.

```
include types.tsdl    /* for fixed_point declaration */

trace {
    version = 1.8;
    packet.header := struct {
        uint16_t magic;
    };
```

```
};

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

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

A point of attention is that the `trace` section is *not* taken into account in included files. When the section is present in an included file, it must still be syntactically correct, but its contents are not stored. The main file should contain a `trace` section with the appropriate settings for the project.

Even with a trace encoding format that has low overhead and a compact encoding, and even when coupled to a fast transfer protocol, run-time tracing will eventually slow down your code to a crawl, given enough of these trace messages. When a project has many modules and there are many trace messages in these modules, you will want to selectively filter out traces from run to run, and focus on an appropriate sub-set for the task at hand.

While a trace viewer often offers filtering capabilities, this wouldn't help with performance —all messages are still transmitted from the target to the host. The tracegen utility optionally generates code to filter trace messages on the target device, and to do so dynamically. This way, the trace messages that are filtered out, are not transmitted from the device —and their corresponding CTF packets are not even assembled.

The messages can be filtered both on stream ID and on a severity level. When using SWO Tracing (see page 76), filtering on stream ID amounts to the same thing as enabling and disabling SWO channels, but the CTF filter works equally well with RS232/UART tracing or the Real Time Transfer (RTT) protocol.

The severity levels are below, from low to high. When setting a severity level, the messages with a lower level are filtered out. For example, when setting the level to "warning," notices, informational and debug messages are no longer transferred. The default level is info.

Severity levels

`debug`	*lowest*
`info`	*default level*
`notice`	
`warning`	
`error`	
`critical`	*highest*

Stream IDs are implicitly assigned when the streams are declared. Severity levels need to be explicitly set on each event. The following snippet declares three events: the first does not declare a severity level, so it gets the default level ("info"); the second is declared as an error and the third will only show up when debug messages are requested.

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

event cooler::rangefault {
    severity = error;
    fields := struct {
        uint16_t value_read;
        uint16_t limit;
    };
};

event cooler::calibration {
    severity = debug;
    fields := struct {
        uint16_t adc_offset;
        uint32_t multdiv;
    };
};
```

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 (prototypes) 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 102 is saved in a file with the name "`peltier.tsdl`," you can run:

```
tracegen -s=swo peltier.tsdl
```

The two files that are created, are named trace_peltier.c and trace_peltier.h. These contain the implementation and declaration of two functions (because there are two events defined in the TSDL snippet):

```
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: in the C function name, the "-" is replaced by a "_."

The "`-s=swo`" option to tracegen makes it generate code for SWO tracing. When you would use the Common Trace Format for tracing over an RS232 line (or TTL-level UART), 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 type "`fixed_point`" 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=swo -t peltier.tsdl
```

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

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

Filtering on streams and severity levels was discussed at the end of the previous section. This functionality must be enabled with tracegen. The applicable command line options are "`f=stream`" and "`-f=level`". Both can be set at the same time. Thus, the command below will allow you to filter on both stream IDs and on severity levels, from within a debugger.

```
tracegen -f=stream -f=level peltier.tsdl
```

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.

The tracegen utility has a few more options. To see a summary, type:

```
tracegen -?
```

When integrating tracegen with Make, note that the output files have the base name of the input file, but with "trace_" prefixed to it. That is, for an input file "peltier.tsdl," the generated output files are "trace_peltier.c" and "trace_peltier.h." An inference rule to match this could look like:

```
trace_%.c : %.tsdl
        tracegen -s=swo -i:stdint.h $<
```

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

To start with the latter, the `trace_timestamp` function returns a device-specific 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 101. If the event header does not include a timestamp, there is no need to implement this function (as it will not be used).

The above example for `trace_xmit` assumes that you ran tracegen with the "`-s=swo`" option on the command line. Without the "`-s=swo`" option, function `trace_xmit` lacks the `stream_id` parameter (the stream ID would instead be embedded in the packet header).

The task of the `trace_xmit` function is to transmit the data over a kind of port or interface. For SWO tracing, this would be an adaption of the trace function on page 77, see the following snippet.

```
void trace_xmit(unsigned 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[stream_id].u32 == 0UL)
```

```
                    {}    /* null statement */
                ITM->PORT[stream_id].u32 = value;
                value = shift = 0;
            }
        }
        /* transmit last collected characters */
        if (shift > 0) {
            while (ITM->PORT[stream_id].u32 == 0UL)
                {}
            ITM->PORT[stream_id].u32 = value;
        }
    }
}
```

For the Real Time Transfer (RTT), the implementation is much simpler, because the signature for the trace transmit function is compatible with the RTT function `SEGGER_RTT_Write`. You might be able to forego writing an interface function altogether, and simply ask tracegen to generate calls to `SEGGER_RTT_Write`.

```
tracegen -s=swo -fx=SEGGER_RTT_Write peltier.tsdl
```

The crucial option in the above command is "`-fx`" (with the name of the substitute function), but we also need to add "`-s=swo`" so that the stream ID is transmitted as the first parameter. In RTT, this functions as the channel index. This would then require that your firmware sets up an output channel for every CTF stream. The alternative is to transmit all trace output on a single, fixed stream, and create a simplistic wrapper function:

```
#define TRACE_CHANNEL 0   /* channel of your choosing */

void trace_xmit(const unsigned char *data, unsigned size)
{
    SEGGER_RTT_Write(TRACE_CHANNEL, data, size);
}
```

To match the above wrapper function, you would run tracegen *without* the "`-s=swo`" option, so that the stream IDs declared in the TSDL metadata are part of the packet data (instead of being passed as a separate parameter).

The previous section briefly covered filtering on stream IDs and a severity level. When filtering is enabled in tracegen, the utility generates the global variables `trace_stream_mask` and `trace_severity_level`, plus a quick "opt-out" test in every generated trace message. The initial settings are that all streams are enabled, and that the severity level is set to "info" —meaning that debug messages are hidden by default. You can set these variables at run time through the debugger or the trace viewer.

GDB allows you to read and set variables, as covered in the section Examining Variables and Memory (page 45). For the BMDebug front-end, the more convenient route is to use its built-in commands for filtering CTF traces, which apply to both its SWO tracing view and its serial monitor. See page 69 for the commands.

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 (*Instrumentation Trace Macrocell*), see SWO Tracing on page 76. 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 *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 conditions that you *know* must be true... provided that the code was called with the correct input parameters, but of which you *know* that these were checked by the caller. The answer to the question of why on earth you would test what you already know to be true, 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 preconditions, postconditions 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 74, 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 sends 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 an assertion should not have a side effect. Changing a variable inside the expression of an assertion is right out of the question, but C functions

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

with side effects, like `strtok()` should be avoided inside an assertion as well. 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 assertion 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. In embedded software, it is common to re-implement the `assert` macro so that it is 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, within 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* of the error location, instead of the filename and line number. This approach reduces overhead to a minimum. Implementations of assertions typically have two parts, a conditionally defined macro and a function that is called when the assertion fails.

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

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 to catch unintentional assignments in the condition; the standard implementation of `assert` stays silent when you write "`assert(var = 1)`," 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.

The core functionality of the `assert` functionality is implemented in the function `assert_fail()`. There is only a single implementation of this function, whilst macros are expanded inline at every invocation. Therefore, it saves code space to let the `assert_fail()` function determine the

address of the assertion failure, rather than 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 `assert_fail()`, plus three support functions. The first thing `assert_fail()` 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 it can be overridden. 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. The intended 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()` would stop all fans and motors and shut heating off.

While on the subject, `__BKPT()` is a CMSIS macro. Other microcontroller 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.

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 62) 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 and 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 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:

```
-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:

```
__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 covered in the previous section on the `assert` macro.

The entry and exit functions themselves should *not* be instrumented, to avoid unbounded recursion. This also applies to any function called from the entry and exit functions —including *inline* functions. To avoid a function from being instrumented, use the "`no_instrument_function`" attribute on it, as was done in the preceding snippet.

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, see:

```
  -finstrument-functions-exclude-function-list
```

and

```
  -finstrument-functions-exclude-file-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 92). 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 following snippet. Note that this is not a complete TSDL file (e.g. it lacks the definitions of

the packet and event headers), but just the part that declares the events —see the appendix Code Profiling, specifically the section on Calltree Analysis (page 120) for a full TSDL file for function tracing.

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

event enter {
    attribute = "no_instrument_function";
    fields := struct {
        code_address symbol;
    };
};

event exit {
    attribute = "no_instrument_function";
    fields := struct {
        code_address symbol;
    };
};
```

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. The events in the above TSDL snippet have a declaration for an `attribute`, which is set to "`no_instrument_function`." This attribute is copied to the generated C functions. As an alternative, you can add the option `-no-instr` 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
```

As covered in section Integrating Tracing in your Source Code (page 107), the C functions generated from the TSDL file call `trace_xmit`, a function that must be implemented by you. That section also gives an example

implementation. When tracing function entry and exit, the `trace_xmit` function must also have the `no_instrument_function` attribute.

Code Profiling

Code profiling is a technique that you use to analyze where the program spends its time; it identifies the "hot spots" or "cycle-eaters" in the code. With this knowledge, you can decide where to optimize the code —and whether to optimize it at all.

Performance optimization is not just about making the code run faster. Increasing the efficiency of the code may allow you to reduce the clock frequency of the microcontroller. Since the power consumption of the microcontroller correlates with the clock frequency, you indirectly reduce power consumption. This may be especially relevant for battery-powered devices.

The two most common methods for profiling are by instrumenting the code and by sampling. Section Tracing Function Entry and Exit (page 113) briefly touched on instrumenting entry and exit of functions. This allows you to profile at a *function* level: it allows you to measure the time spent in each function. It does not tell you which loop or computation inside the function is the hot spot. There are tools to instrument source code such that you get timings on a *source line* level. The drawback is that due to the overhead of instrumentation, the code runs significantly slower, which can affect the sequence in which tasks and interrupts run —and this may then give rise to the probe effect.[1]

Sampling-based profiling works by sampling the *Program Counter* (PC) at a regular interval, and then look up which line in the source code corresponds with the address. No instrumentation is needed, and the code runs at its original speed (no slowdown, and therefore no probe effect). On the other hand, the sampling frequency is typically in the range of a few kHz, whereas a microcontroller runs at several MHz. A lot of code can run between two samples. However, if a profiling session runs long enough, it results in a statistical distribution of where time is spent.

Sampling on ARM Cortex

The ARM CoreSight architecture implements PC sampling in its ITM (*Instrumentation Trace Macrocell*) and DWT (*Data Watchpoint & Trace*) units. This means that the sampling is separate from the microcontroller's arithmetic and logic unit. Thus, sampling is truly non-intrusive; it does not even incur the overhead of interrupt processing.

[1] See also page 92.

The sampled values are transferred to the debug probe via the TRACESWO pin and interface. The configuration for profiling therefore has a lot in common with that for SWO Tracing, see page 76. In particular, the device-specific configuration for SWO tracing must be performed for profiling as well —see the various snippets starting on page 79 for details.

Note that, as is the case with SWO tracing, the ARM Cortex M0 and M0+ architectures lack support for profiling.

The generic initialization for profiling is below. This snippet may be compared with the one on page 79; there are only few differences in the set-up for profiling versus tracing.

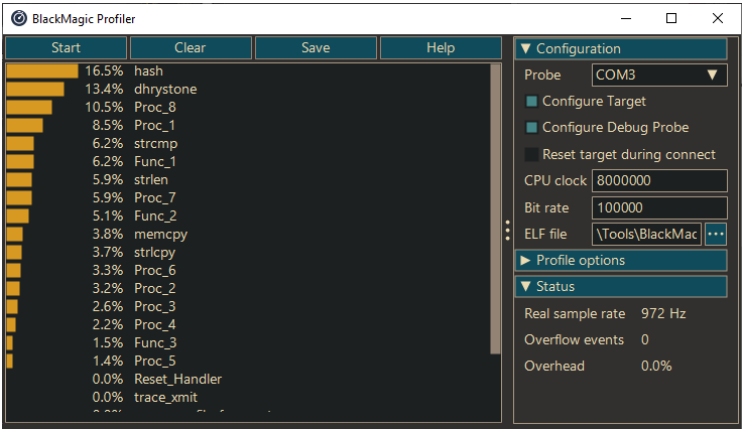
```
void trace_init(int protocol, uint32_t bitrate, uint32_t samplerate)
{
    uint32_t clockfreq =  (protocol == 1) ?  2 * bitrate : bitrate;
    uint32_t divider = CPU_CLOCK_FREQ / (1024 * samplerate);
    divider = min((divider > 0 ? divider - 1 : 0), 15); /* clamp to 0..15 */

    CoreDebug->DEMCR = CoreDebug_DEMCR_TRCENA_Msk;

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

    ITM->LAR = 0xC5ACCE55;    /* unlock access to ITM registers */
    ITM->TCR = (1 << 16) | ITM_TCR_DWTENA_Msk | ITM_TCR_ITMENA_Msk;
    ITM->TPR = 0;             /* privileged access is off */
    DWT->CTRL = DWT_CTRL_PCSAMPLENA_Msk | DWT_CTRL_CYCTAP_Msk |
                DWT_CTRL_CYCCNTENA_Msk | ((divider & 0xf) << 1);
}
```

The above snippet limits the sampling frequency to the microcontroller clock divided by 1024. This relates to the earlier remark that a lot of code can run between two samples (the ARM Cortex architecture executes most instructions in a single cycle). You can increase the maximum sampling frequency by a factor 16 when clearing the CYCTAP bit in the DWT_CTRL register. However, in practice, the limiting factor is the maximum SWO bitrate that the Black Magic Probe supports. The sampling data has to be transferred from the microcontroller to the debug probe, after all.

Once the set-up is done, the microcontroller starts streaming the sampled PC addresses to the debug probe, over the TRACESWO pin. Each packet is five bytes long: a header byte (with a fixed value of 0x17) followed by

a 32-bit address, transmitted low-byte first (see also section TRACESWO Protocol on page 10).

The BlackMagic Profiler (BMProfile) is a small utility that shows the profiling results in a bar-graph. On start-up, it initially shows a list of functions, sorted on sample counts. When you click on a particular function, it shows you the source code for that function, with the bar graph (and percentages of samples) for the source lines. Thus, the utility allows for both *function* level and *source line* level profiling. Clicking in the source view returns to the function list.



Like the BMTrace utility, BMProfile can optionally perform the configuration of the target microcontroller, through the debug interface. It does not use GDB, but relies on the *Remote Serial Protocol* (RSP) to communicate with the target microcontroller. For several families of microcontrollers, BMProfile can do the device-specific initialization for tracing & profiling as well. These steps are optional: if the setting "Configure Target" is deactivated in the right column of the user-interface, no device-specific initialization will be done.

Sampled addresses that fall outside the range of the ELF file, or that cannot be attributed to a function, are collected as "overhead" in the bottom panel of the right column. Some microcontrollers have "drivers" or support routines in ROM that firmware code can use, and time spent in these would thus be collected as "overhead." Similarly, functions linked in from the run-time library sometimes are neither included in the DWARF information, nor in the ELF symbol table. When the PC is sampled inside these functions, it is also collected as overhead.

For further analysis, you can save the collected sample data in CSV format (comma-separated values).

Calltree Analysis

The screen capture of the BMProfile utility, on the previous page, shows that the function on top is `hash` with 16.6% of the MCU time. The total time spent in a routine, like function `hash`, is the time that a single run takes, multiplied by the number of times the routine runs. Thus, the question is, when a function comes out high in the profiling graph, is it because it runs slowly, or because it is called so very often. And this is a question that a sampling profiler doesn't answer, on its own.

A first step in deeper understanding of the code and where it is spending its cycles, is to make a calltree of the program. If you are using C, the GNU cflow program creates calltrees by means of a static analysis of the source code. A "reverse calltree" generates for each function a list of call sequences that lead to running that function. This quickly tells you by who each function is called. The cflow program creates a reverse calltree with the `--reverse` command line option:

```
cflow --reverse dhrystone.c hash.c
```

When using C++, the situation is less straightforward: cflow does not support C++ and commercial alternatives, such as Understand by SciTools and CppDepend by CoderGears, are fairly expensive. Another caveat is that the static call tree of a program is not necessarily the same as the run-time call tree. Functions passed as parameters, call-back functions and invocations of virtual functions are absent from the static calltree.

The alternative is to create a run-time calltree, by running the code while tracing the function entries and exits. This is the topic of section Tracing Function Entry and Exit (page 113). That section covered the concept and included a snippet of a TSDL file that might be used for function tracing. For completeness, a full (and slightly adapted) TSDL file for tracing functions follows below.

```
trace {
    major = 1;
    minor = 8;
    packet.header := struct {
        uint16_t magic;
    };
};

stream function_profile {
    id = 31;
    event.header := struct {
        uint16_t id;
    };
};
```

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

event function_profile::enter {
    attribute = "no_instrument_function";
    fields := struct {
        code_address symbol;
    };
};

event function_profile::exit {
    attribute = "no_instrument_function";
    fields := struct {
        code_address symbol;
    };
};
```

This TSDL file creates a stream "function_profile" at channel 31 —the channel reserved for profiling and system tracing. The goal now is to run the same code while capturing traces with BMTrace (page 85). As you will observe, the code will run significantly slower, but since the goal is to generate a calltree, accurate timings are not as relevant. What is relevant, though, is that all code paths that were executed during the profiling session, are also executed during the function tracing phase.

After saving the captured trace messages to a CSV file, you can run the calltree utility on it to generate a familiar calltree. The calltree utility is another software utility that comes with this book. Like the cflow utility, calltree can create both normal calltrees and reverse trees.

```
calltree -r testrun.csv
```

There are trade-offs between a static calltree and a run-time calltree. As stated, a static calltree misses function calls that are computed at run-time: specifically functions called via function pointers and virtual class members. A run-time calltree includes those functions, but on the other hand it lacks calls to library functions —as those are not instrumented.

Firmware Programming

As show in chapter Debugging Code on page 32, GDB downloads the code in the microcontroller as part of the debugging process. This opens the way for using the Black Magic Probe for small-scale production programming as well.

Using GDB

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 (on 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. On Linux, you may use the BMScan utility to automatically fill in the device name for the gdb-server 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 microcontroller series from NXP regarding the `compare-sections` command on page 38.

Downloading BIN files

GDB only supports the ELF file format for downloading to Flash memory. It may on occasion be useful to be able to write a flat binary file ("BIN file") to an address in Flash memory.

One use case is to restore an image of the firmware previously extracted from the the microcontroller. Imagine, for example, that a development tool that was used to build the firmware, doesn't run on a contemprary operating system, and no update or suitable alternative is available;[1] replicating the firmware from an image onto another (identical) microcontroller may be your only recourse.

Creating the image is accomplished with the GDB `dump` command. The `dump` command writes the contents of a memory range to a file. And while there is a `restore` command as a counterpart of `dump`, `restore` has the limitation that it can only write to SRAM —not to Flash. To download to Flash memory, you must use the `load` command. However, the `load` command requires an ELF file format —and while the `dump` command supports various output formats, it does *not* write ELF.

Your options are to either use a different program than GDB to download the image (an example of such utility is covered in the next section), or to convert the BIN file written by the `dump` command to a pseudo-ELF file. You can do the latter with the `objcopy` utility. This utility is part of the binutils package, as well as of the GNU ARM Embedded Toolchain ("arm-none-eabi").

```
arm-none-eabi-objcopy -I binary -O elf32-littlearm image.bin image.elf
```

The above snippet wraps the flat binary file `image.bin` inside a skeleton of an ELF file with a "`.data`" section for the payload. The generic `objcopy` utility may not support the `elf32-littlearm` target, but for downloading to Flash, the generic `elf32-little` target is fine.

The BIN file format does not record an address where the data should be located in the microcontroller's Flash memory. If you do not specify a base address, `objcopy` stores the data at address 0. This is incorrect for microcontrollers from the STM32 series, or RA series (Renesas). To set a different target address, add an option to the `objcopy` command.

```
arm-none-eabi-objcopy -I binary -O elf32-littlearm --change-section-address
   .data=0x08000000 image.bin image.elf
```

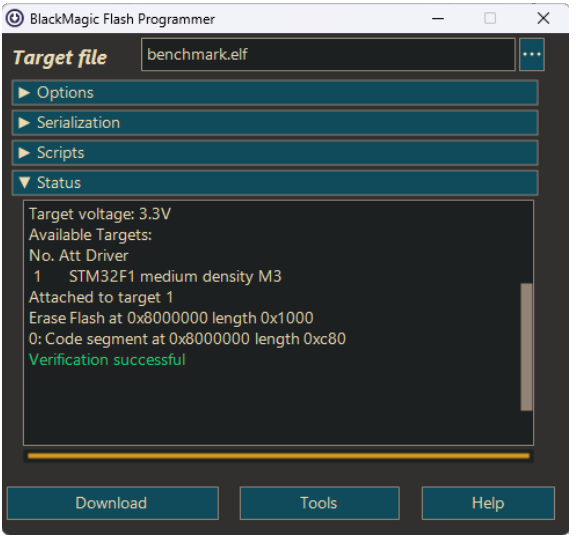
[1] Admittedly, this happened on a proprietary platfom that is completely unrelated to the ARM architecture and its toolchain.

The command should be typed on a single line; it is wrapped over two lines here above, because it would otherwise not fit between the margins.

The `objcopy` utility allows you to rename the "`.data`" section to something else; a "`.text`" section seems more appropriate for an executable image. This might be useful if you also want to be able to debug the code (assembly language only, though, since all symbolic information was lost). This is beyond the scope of this chapter, however.

Using the BlackMagic Flash Programmer

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



The BMFlash utility automatically scans for the Black Magic Probe on start-up, and connects to it. It also has built-in handling of the idiosyncrasies of the LPC microcontroller series from NXP (see, for example, section Verify Firmware Integrity on page 38).

File Formats

BMFlash supports downloading of files in ELF, Intel HEX and BIN (flat binary) formats. The ELF format is recommended, because it provides

the most detailed information on the embedded data. BMFlash runs a few checks on the target file before downloading it to the microcontroller, and some of these checks are only available on ELF files.

The BIN file format lacks information on *the address* that the data must be written to. When selecting a BIN file as the target file, a text field appears below the path/filename of the target file, in which you can enter the base address. The address may be in decimal or hexadecimal; for the hexadecimal format, prefix the value with "`0x`" (as in `0x08000000`). If you leave the address field empty, the data is downloaded to the base address of the Flash memory of the microcontroller —which is usually the correct address to download firmware code too.

Note that while ELF files and HEX files will always be written to the address that is recorded in the file, BIN files can be written to any valid address in Flash.

Code Read Protection

One of the options that BMFlash provides, is to enable Code Read Protection (also called Read-out Protection) during or after the download. You can enable this option when you wish to protect your firmware from being copied out of your product and disassembled and/or examined.[2]

Microcontrollers from diverse brands use different techniques for code protection. Currently, BMFlash supports the STM32 family form STMicroelectronics, and the LPC family from NXP.

The STM32 series of microcontrollers implements code protection via one of its option bytes. The option byte needs to be set after downloading the code, and it is then picked up after a power-cycle. The BMFlash utility lets you choose between levels 1 and 3, but for the STM32, only levels 1 and 2 are valid. Code protection level 1 can be undone (see also section Miscellaneous tools further down in this chapter), but level 2 secures the microcontroller's Flash memory permanently.

The LPC series of microcontrollers uses a magic value at a fixed location to enable code protection. Three levels are available. In principle, all protection levels are reversible, but level 3 requires that the firmware does so from within —so if the firmware does not have a "self-erase" function, level 3 cannot be reverted. In any case, all three levels disable the SWD interface, so reverting levels 1 and 2 of Code Read Protection requires

[2] Code Read Protection deters the casual hacker, but not a determined one. The protection schemes of both the STM32 and LPC families have been defeated.

you to use either ISP (in-circuit programming via the UART0 peripheral of the microcontroller, with the $\overline{\text{BOOT}}$ pin held low), or implement self-erase functionality in the firmware.

There is actually also a fourth magic value, but "CRP" level 4 only disables the $\overline{\text{BOOT}}$ pin on start-up, with the side effect that the microcontroller won't enter ISP mode. It does not provide any kind of read protection, however. The BMFlash utility does not offer level 4, for this reason. That does not stop you for setting level 4 —all it takes is to have the correct magic word at the predefined address in the firmware.

The BMFlash utility is only able to set the appropriate magic value for the selected CRP level, if the dedicated address is reserved in the target file. The magic value is a 32-bit number, and its address is `0x000002FC` for ARM Cortex microcontrollers and `0x000001FC` for microcrontrollers using the older ARM7TDMI architecture. When building the firmware, and code protection is required, the compiler should make sure that it does not store anything else at this address. The GNU toolchain handles this with a linker script that places the vector table at the start, and all code sections *behind* the reserved spot for the CRP value.

It is common, though, that a project uses two linker scripts: one used during development, and another for the release version with CRP set. The development build does not reserve a spot for the CRP magic value. The rationale is: you need to be able to build a version *without* code protection during development, because you cannot debug protected code, but no magic value for CRP level 0 ("no CRP") has been defined. The unintended side-effect is that if you build for release, but *without* CRP (for a final test), the generated firmare is different from the version you subsequently build *with* CRP. Not only because of the magic value, but mostly because the code is moved to different addresses, due to the space reserved in the linker script.

With the BMFlash utility, a simpler workflow is possible, using a single linker script. BMFlash defines the magic value `0xBC00B657` for CRP "level 0." Thus, the set of magic values becomes:

Magic	Level
0xBC00B657	No protection
0x12345678	CRP1
0x87654321	CRP2
0x43218765	CRP3
0x4E697370	$\overline{\text{BOOT}}$ pin on start-up disabled

In the linker script, you now reserve a 32-bit word at the dedicated address, and set this to the provisionary signature for "level 0": `0xBC00B657`.

The relevant part for the linker script follows below, with the important lines in bold.

```
SECTIONS
{
  .text {
    . = 0;
    *(.isr_vector_table)      /* vector table must start at address 0 */
    *(.after_vectors*)

    . = 0x000002FC;           /* location for CRP magic word */
    LONG(0xBC00B657)          /* magic for "no protection" */

    *(.text*)                 /* start-up code & read-only data */
    . = ALIGN(4);
    *(.rodata .rodata.*)      /* read-only data */
  } > flash

  /* other sections ... */
}
```

When firmware is built with this linker script, code is unprotected by default. So it allows you to debug the release build of the firmware with GDB and the Black Magic Probe —since the debug symbols are never downloaded to the target microcontroller, there is no downside to always build the release version with full debug info. Then, when the time comes for production programming, select the appropriate level of protection in the options of BMFlash, and the utility will replace the provisionary magic word `0xBC00B657` by the appropriate value on the flight.

BMFlash gives a notice in the log viewport when it sets CRP. If you downloaded firmware with CRP accidentally (for example, because the option was still set from an earlier run), you can undo this by wiping the Flash memory immediately. When a CRP mode is set, the SW-AP port will be disabled after a reset or power cycle; however, as long as you do not reset the target, you keep access to the Flash memory.

Serialization

BMFlash supports serialization of the firmware, meaning that each device that the firmware gets downloaded to, gets a unique serial number in firmware. It does this by patching a serial number into the code during the download (without changing the original file). After each successful download, the serial number is incremented.

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 and an offset in bytes. The offset is a hexadecimal value. The section name typically ("`.text`" or "`.rodata`") is only relevant for ELF files. For BIN and HEX files, leave the section field empty. The offset is relative to the section, if it is present; otherwise it is an absolute offset from the beginning of the file.
Match	In this mode, the BMFlash utility searches for a signature or byte pattern in the target file, and replaces it with another byte pattern ("prefix") plus a serial number. The "match" string can be an ASCII string, like "$serial$." It can also contain binary values, which you specify with `\`ddd or `\`xhh 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 to single-byte characters. If a backslash must be matched, it must be doubled in the match field. The "prefix" string follows the same syntax as the "match" string. It is optional; if not present, the serial number is written from the start of the signature found in the target file. If you want to store the serial number behind the signature in the target file (without modifying the signature), the prefix string should be set equal to the match string.

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.

The configuration settings are stored in a file that has the same name as the target file, but with the extension "`.bmcfg`" appended to it. By default,

the settings for serialization are also stored in this file. However, one of the options in the Serialization section is to configure a separate file that contains only the serialization options, plus the up-to-date serial number. This allows you to use a shared file for the serialization options and serial number —for example, for the case that you have production firmware in multiple variants.

You have a few options to implement serialization. First is to store the serial number at a fixed address in a segment that resides in Flash memory. If you control the start-up code for the microcontroller, you can, for example, reserve a field for the serial number right behind the interrupt vector table. Since this table is at a fixed address and has a known size, you can choose "Address" mode for serialization, and enter the segment name and the offset.

Alternatively, you can declare a "static const" array or string anywhere in your source code and initialized it to a sequence of characters that is unique in the program. Then, you can choose "Match" mode and let it search for this signature (the sequence of characters). The signature needs to be at least as long as the width of the serial number (otherwise BMFlash may patch over data or instructions that follow the array). As an aside, BMFlash checks the complete target file for the signature, and gives a warning when it is found multiple times.

Log file

The BMFlash utility can optionally add a row to a log file for each successful download. To activate it, set a check-mark in the "Keep log of downloads" option in the "Options" tab. The log file is in CSV format (comma-separated values). The filename is the same as that of the target file (the file that is downloaded to the target), but with the extension "`.log`" appended.

Each row starts with the date and time of the download. It is followed by three fields identifying the target file: the file date & time, the size in bytes, and a POSIX checksum. This checksum is actually a CRC32 of the contents of the file plus the file size. After that, there is an RCS identification string (only for ELF files, and only if one was present), and finally the serial number patched into the code during the download (if serialization is enabled).

The POSIX checksum enables you to distinguish which version of the target file was downloaded. It is calculated over the original target file, before patching a serial number in the code. You can verify whether a file matches the number in the log file, by running `cksum` on the ELF file.

The utility `cksum` is a core utility of Unix and Linux distributions; it has also been ported to Windows as part of the GnuWin32 project. A self-contained re-implementation of the `cksum` utility (with minimal features) is also provided with this book.

The RCS identification string is easier to use as a unique identifier of the code that was downloaded. It works in conjunction with a version-control system and a placeholder for the identification string in the source code. On each commit, the version-control creates a unique stamp and patches that into the placeholder in the source code. With the stamp, you can then look up the matching commit in version-control history.

RCS (*Revision Control System*) is legacy version-control software, but the format for the identification strings lives on. A typical string that you would add to the main source file of your embedded application is:

```
const char __id[] = "$Revision$";
```

Whereas RCS used a handful of keywords, BMFlash only supports `$Id$` and `$Revision$` (which may be abbreviated to `$Rev$`). You must furthermore enable keyword expansion in your version-control software, on all source files. In Apache Subversion, add the "`svn:keywords`" property to the source files and add at least the "`Id`" and/or "`Revision`" keywords to the list. For `git`, you can add the following lines to the `.gitattributes` file:

```
*.c ident
*.h ident
```

Note that `git` only supports the `$Id$` keyword (not `$Revision$`).

These are not the only solutions (in fact, the above solutions have their shortcomings). When using Subversion, a more reliable scheme is to use the `SvnRev` utility as part of the build. The code to add to the main source file changes to the snippet below:

```
#include "svnrev.h"
const char __id[] = SVNREV_RCS;
```

Enabling keyword expansion is not required when using `SvnRev`. For `git`, Mercurial and Bazaar, an alternative is Autorevision, but which runs only on Linux. See Further Information on page 182 for links to the various utilities.

As an aside, if you want to check whether a file contains RCS identification strings, you can use the `ident` utility. This utility is part of the RCS package (and of the GnuWin32 project); a self-contained re-implementation is also provided with this book.

Preprocessing and Postprocessing

The "Scripts" tab has fields for two filenames: a script that runs *before* the download and a script that runs *after* the download. Both scripts are optional.

The scripts receive the firmware filename and serial number (plus other fields, if relevant) as variables. A script can perform various actions, on the firmware or on the target, and it can also read/write files on the host workstation or launch other utilities. See chapter Tcl Primer on page 153 for details and examples.

By default, the postprocessing script only runs on a successful download. However, below the field for the filename, there is a setting to also run the script after a failed download. The status of the download is passed onto the script (in a variable), so that the script can check how to proceed.

Miscellaneous tools

The Tools button on the bottom row of the utility provides a few additional commands:

- Re-scan for Black Magic Probes on the USB bus (e.g. for the case that you launched the utility without first connecting a Black Magic Probe).
- Verify whether the code in the microcontroller matches the selected target file (without downloading it).
- Erase the option bytes (on microcontrollers that use option bytes), see the notes below.
- Erase the full Flash memory of the target, see also the notes below.
- Check that a microcontroller is blank, meaning that its Flash memory is fully erased.
- Dump the contents of Flash memory to a flat binary file (BIN format).

The STM32Fxx microcontrollers use "option bytes" for code protection. Erasing these clears the protection, and by clearing protection, it also erases all Flash memory. The STM32Fxx microcontrollers need a power-cycle, before the change in option bytes is picked up. If the target is powered from the Black Magic Probe, this is handled automatically by the BMFlash utility. When the target device is self-powered, you should power-cycle it after clearing the option bytes.

Also see the section Reset Code Protection at page 35 for more information.

A full erase of the Flash memory is needed to clear the code protection of some other microcontroller families. As stated in the section Code Read

Protection on page 125, enabling "code read protection" on the LPC microcontrollers (from NXP) disables the SW-AP port after a reset or power cycle, so you want to erase the Flash memory after accidentally downloading code with the CRP option set, you must do so immediately, without leaving the BMFlash utility and without power-cycling (or resetting) the target device.

The Dump Flash to File option in the Tools menu always dumps the full Flash memory, but then trims it to the part that actually contains data. So if a microcontroller has 64 KiB of Flash memory, but the firmware fills only the bottom 40 KiB, the resulting BIN file will be 40 KiB is size. However, if the firmware stores configuration data in the last sector of Flash memory, the BIN file will include this as well (and will then be 64 KiB in size). The unused (blank) Flash sectors are stored as blocks of bytes the value `0xFF`.[3]

[3] The technical detail is: "programming" Flash only writes zero bits. Erasing Flash sets all bits to ones. Thus, when storing new data in Flash memory, it must first be erased (to remove all zeros) and then be re-programmed. The take-away is that blank Flash memory has all bit set, so byte values `0xFF`.

Updating Black Magic Probe Firmware

At the time of writing, the current version of the Black Magic Probe hardware is version 2.3. The predecessor, "hardware version 2.1," is now sold out, but still widely in use. The firmware is in continuous development, however, for both versions 2.3 and 2.1 (and for other platforms as well). Support for more microcontrollers, as well as new features, are regularly being committed to the GitHub project. There is therefore good reason to update the firmware of the Black Magic Probe to either the latest "stable" firmware release (version 1.9.1 at the time of writing), or an up-to-date "development version."

You can build the latest firmware yourself (see section Building from Source on page 135), but you often do not need to. The stable releases are available on the GitHub project, and pre-compiled builds of the development release are also updated each day. See chapter Further Information on page 182. The exception is when you want to enable features that are disabled in the official build.

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 20. As noted in that section, both the DFU interfaces for *normal mode* and *DFU mode* must be installed.

The next step is to install a utility to perform the DFU procedure. At the time of writing, a dedicated tool, bmputil, is in the early stages of development —but it may have progressed by the time you are reading this. The prevalent general purpose utility for DFU is dfu-util. Links to both bmputil and dfu-util are in section Further Information, page 182. In Linux, it can be conveniently installed through the package manager of your distribution; for example:

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

The remainder of this section uses dfu-util for the options, both because bmputil does not yet have a stable release, and because dfu-util works for the native Black Magic Probe as well as compatible probes.

The options on dfu-util for updating the firmware are (native Black Magic Probe):

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

For ctxLink, use the command below (note the address set with the `-s` option):

```
dfu-util -a 0 -s 0x08000000 -D blackmagic-ctxlink.bin
```

Note that you need to use the specific "ctxLink firmware" for the ctxLink; the native Black Magic Probe firmware will not run on the ctxLink.

Likewise, the Jeff Probe requires a different address; and similar to the ctxLink probe, you must get the firmware specifically for the Jeff Probe from the manufacturer's GitHub project page. The dfu-util command options for the Jeff Probe are:

```
dfu-util -d 1d50:6018,:6017 -s 0x00002000:leave -D blackmagic.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 page 22).

You can check which firmware version you have with the GDB `monitor` command (after connecting it as an "`extended-remote`" target, see also page 27).

```
(gdb) monitor version
Black Magic Probe v1.8.2, Hardware Version 3
Copyright (C) 2022 Black Magic Debug Project
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

The BMScan utility also shows the firmware version number (see page 27 for an example of the output).

The development release of the firmware uses a GitHub hash instead of (or in addition to) a version number. In the snippet below, it is the hexadecimal value "`0740d92a`."

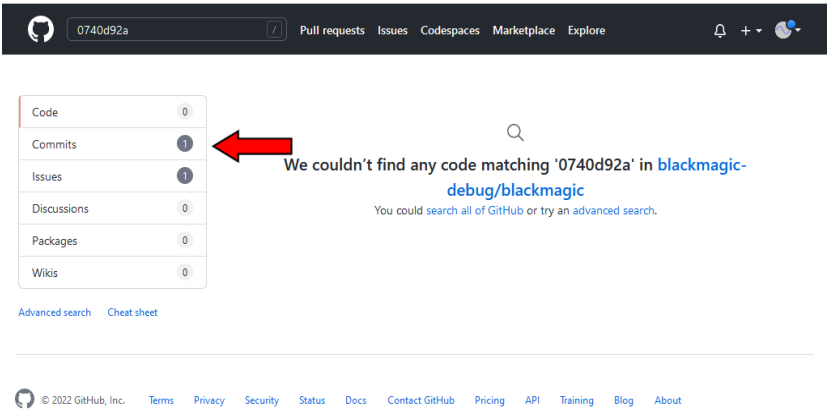
```
(gdb) monitor version
Black Magic Probe 0740d92a, Hardware Version 3
Copyright (C) 2022 Black Magic Debug Project
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

The format of the version number changes from time to time. In more recent development releases, it looks like the example below, and in this case the GitHub hash is the hexadecimal string at the end, after the "g".

```
Black Magic Probe v1.8.0-1138-g0740d92a, Hardware Version 6
Copyright (C) 2022 Black Magic Debug Project
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

A drawback of a hash is that these numbers 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. Note that you should enter only the hash number, not the complete version string (and without a "g" prefix, if any).

When GitHub returns the search results, the page will tell you that it could not find any *code* that matches the hash (`0740d92a` in the above example). In the left column, however, is a list with the topics Code, Commits, Issues, and a few others. If you click on "Commits" (see arrow in the picture) you will get a summary of the relevant commit, plus the date of that commit. You can then look through the list of commits, to see what changes have been applied to the firmware since the release of the version that you have.



Building from Source

A few tools are required to build the Black Magic Probe firmware. First of all, a GCC version for ARM Cortex M architecture, but you are likely to already have it. Other tools, like Python, git, autotools and make are typically also already installed on Linux. If not, you can get them from the package manager of your Linux distribution. If you are running on Microsoft Windows, see further down in this section for setting up a build environment.

Once you have the prerequisites, you have the option of downloading the source code of a release (version 1.9.1 at the time of writing) from the GitHub page, or getting the latest *mainline* source with `git`.

```
$ git clone https://github.com/blackmagic-debug/blackmagic.git
```

In the directory where the firmware source is downloaded or extracted, you can now run `make` to build it. For the tool suite prefix, the Makefile uses "`arm-none-eabi-`" by default. This is usually the correct one, and it can be overruled with a command line option if not. See the Black Magic

Probe project website for other options —specifically if you want to build firmware for derivatives of the original Black Magic Probe.

When `make` finishes, the firmware file, "`blackmagic.bin`" is found in the "`src`" directory inside the firmware source directory.

The default options build firmware for the native Black Magic Probe hardware, but excluding support for SEGGER's Real Time Transfer. To build firmware that includes RTT support, add the relevant option to `make` (and possibly do a "`make clean`" before):

```
$ make ENABLE_RTT=1
```

The firmware can be built for other probes than the native hardware. For example, a common budget option has been to convert an ST-LINK/V2 debug probe to a Black Magic Probe, by way of building the firmware for the "`stlink`" platform. A particular build platform is "hosted," —a version of the firmware that runs on the workstation, instead of on dedicated hardware. The resulting executable is now called BMA or BMDA (which stands for "Black Magic (Debug) App"), but the platform is still referred to as `hosted`. To build firmware for an alternative platform, set it on the `make` command:

```
$ make PROBE_HOST=hosted
```

Microsoft Windows Build Environment

To build the firmware on Microsoft Windows, the first step is to set up a Linux-like environment. The two most popular ways are Cygwin and msys2. Sid Price has documented the steps for Cygwin, see Further Information on page 182 for a link. The steps for msys2 (my preference) follow below.

After downloading the msys2 installer and running it, run the ucrt64.exe program in the installation directory. This opens a console window ("terminal" in Unix terminology). The initial setup is bare-bones; the essential development tools must now be installed with the msys2 package manager.

```
$ pacman -S base-devel
$ pacman -S autotools
$ pacman -S git
```

The three packages in the above snippet can be combined on a single `pacman` command, by the way. For clarity, I am installing one package at a time.

A caveat with the msys2 console (called "`mintty`") is that it supports *copy & paste* from the clipboard with the legacy key combinations Ctrl+Insert and Shift+Insert, but *not* with Ctrl+C and Ctrl+V. Instead, Ctrl+V inserts a character with ASCII code 22 on the command line, and the trouble is: `mintty` does not show characters with ASCII values below 32. Thus, if you use Ctrl+V to paste a command line into `mintty`, nothing will apparently happen —but if you then paste the command with the proper key combination and press enter, the command will fail because there is this non-visible character "Ctrl+V" in front of it.

To build firmware for native hardware (or any other supported platform except "hosted"), Python is also needed (the Makefile uses it to configure the `libopencm3` library, which is used by all platforms except "hosted").

```
$ pacman -S python3
```

A final tool that is needed is the GCC compiler for the ARM Cortex M architecture. As stated earlier, if you develop for ARM Cortex microcontrollers, you likely already have it. In the msys2 environment, you can then add the location of this compiler to the search path, but using a command in the Unix syntax. For example, if you installed the GCC compiler in C:\gcc-embedded, the command look like this:

```
$ export PATH=/c//gcc-embedded/bin:$PATH
```

As an alternative, you can also install the compiler in the msys2 environment with the package manager (which then also installs `newlib`):

```
$ pacman -S mingw-w64-ucrt-x86_64-arm-none-eabi-gcc
```

To build the "hosted" platform, a few different packages are needed. This platform runs on a desktop PC, which (on Microsoft Windows) implies the x86/AMD64 instruction set. Hence, a different compiler back-end and support libraries are needed.

```
$ pacman -S pkg-config
$ pacman -S mingw-w64-ucrt-x86_64-gcc
$ pacman -S mingw-w64-ucrt-x86_64-libftdi
```

Once the msys2 environment is set up, you can proceed downloading the Black Magic Probe firmware source code and build it, as covered in the first part of this section (page 135).

Troubleshooting

The first step in troubleshooting is to check whether the Black Magic Probe has power. When the Black Magic Probe is connected to a USB port, the green and orange LEDs (labelled "PWR" and "ACT" respectively) should be on. The ACT LED may be bright or dim, depending on the firmware version. On the ctxLink, only the ACT LED must be on; the green LED indicates the connection mode, not "power."

After having verified that the Black Magic Probe is powered-on, the next steps depend on the issue that you are having.

Check whether the system detects the probe

The step to check whether the system can find the Black Magic Probe is covered in section Checking the Set-up, page 27. To summarize, the BMScan utility shows the interfaces of all probes that it can find. For troubleshooting, it is of course recommended connecting only a single debug probe at a time; for the ctxLink, we also recommend to first get the device set up with a USB connection.

```
d:\Tools> bmscan

Black Magic Probe [Version: 1.8.2, Hardware Version 3, Serial: 7BB180B4]
  gdbserver port: COM9
  TTL UART port:  COM10
  SWO interface:  {9A83C3B4-0B99-499E-B010-901D6C2826B8}
```

If this fails, the first step is to check that the connectors of the USB cable fit correctly (deep enough) into the Black Magic Probe and the host (workstation). The Black Magic Probe does not come with a USB cable; if you grabbed a random one from your storage, verify that the cable is not a "charge-only" cable.

The BMScan utility also does a quick test on whether it can open the ports that it detects, and on the gdbserver port, whether it responds to a version request. If it indicates "[no access]" after a port (like in the example below), it has detected the port, but at the same time failed to open it.

```
$ bmscan

Black Magic Probe [Serial 7BB180B4]
  gdbserver port: /dev/ttyACM0 [no access]
  TTL UART port:  /dev/ttyACM1 [no access]
  SWO interface:  1-2:1.5
```

There are multiple reasons why a serial port cannot be opened. An obvious one is that the port is already open. On Linux, another common cause is access rights: a non-root user must be included in the `dialout` group to access the ports, see page 23 for more information.

Check whether the probe detects the target

As is common for open-source project, several derivatives of the Black Magic Probe have emerged. The ctxLink probe is one of these, and one that offers additional features. Most derivatives, however, are optimized on cost —by cutting down on features or quality. Some of these probes do not have sufficient Flash memory to contain the current Black Magic Probe firmware, others are simple (FTDI MPSSE-based) protocol interfaces that do not contain any embedded firmware at all. To make these low-cost probes work, the firmware of the Black Magic Debug project can also be built as a desktop application. Thus, the "firmware" runs on a workstation or laptop, and it communicates with the debug probe via one of several low-level serial protocols. It was originally called the "hosted" set-up, but the project now prefers to refer to it as BMA or BMDA, which stands for the "Black Magic (Debug) App."

The relevance, in regard to this chapter, is that the native Black Magic Probe also supports a low-level serial protocol, and (more importantly), the desktop-build of the Black Magic Probe firmware offers additional diagnostics. For the purpose of troubleshooting, the advice is therefore to run the Black Magic code on the desktop, while it is connected to the Black Magic Probe hardware (and while the Black Magic Probe is also connected to a target device).

```
$ blackmagic -t
```

The "`-t`" option displays the information that the utility gathers about the debug probe and the target. When the target is powered from the Black Magic Probe, you should also add the "`-p`" option. The utility has more options that may be relevant. Use the following command to list them all:

```
$ blackmagic -h
```

The output of the "`blackmagic -t`" command, for the case that no problems are detected, is similar to the snippet below:

```
$ blackmagic -t

BMP hosted
 for ST-Link V2/3, CMSIS_DAP, JLINK and LIBFTDI/MPSSE
Running in Test Mode
Target voltage: 3.3V Volt
Speed set to  3.2727 MHz for SWD
DPIDR 0x0bb11477 (v1 MINDP rev0)
RESET_SEQ failed
AP   0: IDR=04770021 CFG=00000000 BASE=e00ff003 CSW=03000040 (AHB-AP var2 rev0
Halt via DHCSR: success 01030003 after 1ms
ROM: Table BASE=0xe00ff000 SYSMEM=0x00000001, designer 43b Partno 471
0 0xe000e000: Generic IP component - Cortex-M0 SCS (System Control Space)
 (PIDR = 0x04000bb008  DEVTYPE = 0x00 ARCHID = 0x0000)-> cortexm_probe
CPUID 0x410cc200 (M0 var 0 rev 0)
1 0xe0001000: Generic IP component - Cortex-M0 DWT (Data Watchpoint and Trace)
 (PIDR = 0x04000bb00a  DEVTYPE = 0x00 ARCHID = 0x0000)
2 0xe0002000: Generic IP component - Cortex-M0 BPU (Breakpoint Unit) (PIDR =
 0x04000bb00b  DEVTYPE = 0x00 ARCHID = 0x0000)
ROM: Table END
***  1      LPC11xx M0
RAM   Start: 0x10000000 length = 0x2000
Flash Start: 0x00000000 length = 0x20000 blocksize 0x1000
```

One of the first things to look at is the target voltage; it is 3.3 V in this example. The Black Magic Probe uses the voltage on the VREF pin for its voltage level shifters on the debug pins. When the voltage on the VREF pin is zero, for example because you did not wire the VREF pin to the target, then the Black Magic Probe won't work.

If the target runs on 3.3 V, you can power it from the Black Magic Probe by adding "`-p`" option on the command line of the `blackmagic` program (in addition to "`-t`").

The following snippet shows a case where the debug probe cannot find a target microcontroller:

```
$ blackmagic -t

BMP hosted
 for ST-Link V2/3, CMSIS_DAP, JLINK and LIBFTDI/MPSSE
Running in Test Mode
Target voltage: 3.3V Volt
Speed set to  3.2727 MHz for SWD
Exception: SWDP invalid ACK
Trying old JTAG to SWD sequence
Exception: SWDP invalid ACK
```

```
No usable DP found
Can not attach to target 1
```

You will also get this response when the target voltage is 0 V, but that is not the case here. You should double-check the wiring between the Black Magic Probe and the target device. You may want to try to repeat the command again with the "-c" option (for connecting under reset). Other explanations are:

- ◇ The target microcontroller has redefined the SWCLK and/or SWDIO pins, and hence you need to reset to bootloader mode (see page 30).
- ◇ The SWD interface has been disabled altogether —e.g. because code-read protection is active on an NXP LPC-series microcontroller. See section Reset Code Protection on page 35 for details.

Another case that may occur is that the target microcontroller does not yet appear in the tables of the Black Magic Probe. New microcontrollers are introduced on the market at a quick pace, and software support for them is often a bit lagging behind. The output for an unsupported microcontroller is similar to the snippet below (this is a contrived example, the particular microcontroller with these designer & part numbers and ID code *is*, in fact, supported by the Black Magic Probe):

```
$ blackmagic -t

BMP hosted
 for ST-Link V2/3, CMSIS_DAP, JLINK and LIBFTDI/MPSSE
Running in Test Mode
Target voltage: 3.3V Volt
Speed set to  3.2727 MHz for SWD
DPIDR 0x0bb11477 (v1 MINDP rev0)
RESET_SEQ failed
AP   0: IDR=04770021 CFG=00000000 BASE=e00ff003 CSW=03000040 (AHB-AP var2 rev0
Halt via DHCSR: success 00030003 after 2ms
ROM: Table BASE=0xe00ff000 SYSMEM=0x00000001, designer 43b Partno 471
0 0xe000e000: Generic IP component - Cortex-M0 SCS (System Control Space)
 (PIDR = 0x04000bb008  DEVTYPE = 0x00 ARCHID = 0x0000)-> cortexm_probe
CPUID 0x410cc200 (M0 var 0 rev 0)
LPC11xx: Unknown IDCODE 0x2998802b
LPC8xx: Unknown IDCODE 0xfffdff88
1 0xe0001000: Generic IP component - Cortex-M0 DWT (Data Watchpoint and Trace)
 (PIDR = 0x04000bb00a  DEVTYPE = 0x00 ARCHID = 0x0000)
2 0xe0002000: Generic IP component - Cortex-M0 BPU (Breakpoint Unit) (PIDR =
 0x04000bb00b  DEVTYPE = 0x00 ARCHID = 0x0000)
ROM: Table END
*** 1 Unknown ARM Cortex-M Designer 43b Partno 471 M0
```

For the team maintaining the Black Magic Probe firmware (on GitHub), there are several important values in this dump. The microcontroller is detected as a Cortex-M0 architecture. The numeric ID for the designer is `43b` (hexadecimal) and the ID for the part is `471`. These values are not conclusive: the value `43b` is the code for ARM Ltd., for example —but the part is from NXP.

The information on the architecture, the designer and part IDs is sufficient to probe deeper. As can be seen from the output, the Black Magic Probe tries to match microcontrollers in the `LPC1100` and `LPC800` series. Both attempts fail, but the `IDCODE` values are important. In this particular case, the microcontroller being tested was an `LPC11U14`, and many microcontrollers from that series are already supported. It may be sufficient to add the number `0x2998802b` to a table or list of known codes that are matched against.[1]

When running the Windows build of the `blackmagic` utility, you may need to set the "`-d`" option with the COM port of the Black Magic Probe, in addition to the other options.

```
$ blackmagic -t -d com9
```

Whether or not this option is needed depends on the build options for the utility —more specifically, it depends on which debug probes the utility is built to support. When the `blackmagic` utility is configured (during compilation) to only support the Black Magic Probe (or 100% compatibles like the ctxLink), this option is not needed; when the utility is configured to support additional debug probes (e.g. ST-Link V2 or V3, or CMSIS-DAP), you will need to set the COM port for the Black Magic Probe.

How to get the hosted blackmagic utility?

When you come to the point that you want to run the blackmagic program for troubleshooting, the first hurdle in doing so is... that you don't have it. The hosted build is not part of the official releases.

If you are running Linux, you can get a daily build from GitHub (below the "Actions" tab). An executable build of the "hosted" firmware is part of the daily builds.

On Microsoft Windows, you have a few options, one of which is to build it yourself —see section Building from Source on page 135. If you don't want

[1] As stated earlier: this is a contrived example. The given code (`0x2998802b`) is already in the list for the `LPC11**` series. I used a sabotaged build of the firmware to get this output.

to set up a build environment just for the purpose of troubleshooting, for your convenience, an executable `blackmagic` utility for Microsoft Windows can be downloaded from the GitHub project for this book —again, see Further Information on page 182 for a link. We tend to follow the official releases of the Black Magic project, so this version is a *release* build rather than a daily build.

Target scan hangs

When the "`monitor swdp_scan`" command appears to hang, this may indicate a problem in the communication between the Black Magic Probe and the target. A typical situation is:

```
(gdb) monitor swdp_scan
Target voltage: 3.3V
```

Neither does the list with available targets appear, nor is there an error message. The "`(gdb)`" prompt does not appear either —until you break the connection between your workstation and the Black Magic Probe.

The target voltage is measured by the Black Magic Probe itself; this first part of a scan does not involve communication with the target. The second phase of a scan, however, is to query the target using the SW-DP protocol. More pointedly, the target scan is typically the first command that involves communication with the target.

The *hang-up* is the result of the target responding, but responding in a way that confuses the debug probe or GDB. This can be caused by the wiring between the Black Magic Probe and the target: a wonky connection, noise picked up by long wires, reflections (transmission line effects), . . . So the first step is to check connections and whether there is a noise source (like, for example, a noisy power supply that powers the target).

If the above step does not (reliably) fix the problem, an alternative is to reduce the speed of the SW-DP protocol, using the "`monitor frequency`" command. On start-up, the `SWCLK` clock runs at approximately 3.5 MHz. You can lower this with the following command:

```
(gdb) monitor frequency 2M
Max SWJ freq 00225510
```

This sets the `SWCLK` frequency to roughly 2 MHz —roughly, because the slowdown is done by inserting "wait states" in the toggling of the `SWCLK` pin, and those wait states are in discrete amounts. The value that the command returns, `00225510` in the above example, is in hexadecimal in firmware 1.8.

You need to issue the "`monitor frequency`" command *before* running "`monitor swdp_scan`," of course.

GDB crashes on "attach"

If you get either of the following errors on attaching to the target, this relates to a bug in GDB versions 11.1 up to (at the time of writing) 13.1.

```
(gdb) attach 1
../../gdb/remote.c:7979: internal-error: ptid_t remote_target::select_thread_for_ambi
 guous_stop_reply(const target_waitstatus*): Assertion `first_resumed_thread != nullptr'
 failed.
A problem internal to GDB has been detected,
further debugging may prove unreliable.
Quit this debugging session? (y or n) [answered Y; input not from terminal]

This is a bug, please report it.  For instructions, see:
<https://www.gnu.org/software/gdb/bugs/>.
```

or

```
(gdb) attach 1
Attaching to program: blinky, Remote target
../../gdb/thread.c:72: internal-error: thread_info* inferior_thread(): Assertion
 `current_thread_ != nullptr' failed.
A problem internal to GDB has been detected,
further debugging may prove unreliable.
```

The GDB bug affects communication with `gdbserver` stubs in general, so it is not specific to the Black Magic Probe or any particular microcontroller. A patch has been available for some time, but at the time of writing, there is no confirmation that this patch will be included in any future release of GDB.

While waiting for a fix, there are two workarounds:

- revert to GDB version 10;
- update the Black Magic Probe firmware to 1.8.2 (or later), which contains a bypass for the GDB issue.

Attach regularly fails

One reason why GDB may have difficulty to attach to a microcontroller, is that the firmware that is currently running in it, puts it in low-power mode (or *sleep* mode). The solution then is to connect under reset, see page 51 for the command description.

Failure to erase Flash memory

When the target microcontroller is from the STM32F series, the following error on a `load` command is likely due to "**r**ea**d**out **p**rotection" (RDP).

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

For more information, see Reset Code Protection on page 35.

Spying on the communication

GDB communicates with the Black Magic Probe via the *Remote Serial Protocol* (RSP). This is largely a text-based protocol. GDB lets you view the commands and responses of this protocol with the following command:

```
(gdb) set debug remote 1
```

The strings of RSP are intermixed with the other console output of GDB. They are easy to recognize, though: each string is prefixed with "Sending packet:" or "Packet received:." In RSP, binary data, and monitor commands and their replies, are transmitted in encoded form. The strings that GDB prints in its console are the data as it transmits it to the Black Magic Probe; that is, in encoded form.

For example, the snippet below shows the output of a `monitor` command. The `monitor` command itself is translated to the `$qRcmd` command; its parameter (the string "`tpwr enable`") is encoded as two hexadecimal digits per character. The reply starts with the letter "`O`" and then a string that is encoded in the same way. The decoded message ("Enabling target power") is printed next.

```
(gdb) set debug remote 1
(gdb) monitor tpwr enable
Sending packet: $qRcmd,7470777220656e61626c65#07...Ack
Packet received: O456e61626c696e6720746172676574207 06f7765720a
Enabling target power
Packet received: OK
```

The Remote Serial Protocol is described in detail in the GDB manual, "Debugging with GDB." See Further Information on page 182 for a reference.

How to Reset the Black Magic Probe

Tracing and debugging may leave the Black Magic Probe or the USB drivers in a state that they cannot recover from. For example, when resetting a target just when it was transmitting data over TRACESWO, the TRACESWO capture on the Black Magic Probe may get stuck in an error state. We have also experienced that Microsoft Windows keeps flagging the virtual COM port used by gdbserver as "in use" even though GDB has already ended.

A reset of the Black Magic Probe fixes these issues. However, the Black Magic Probe neither has a "Reset" button, nor a command for that purpose.[2] Yet, you can reset the Black Magic Probe without physically removing and re-inserting it, with the help of DFU. DFU is the protocol for updating the firmware of the Black Magic Probe itself. It is covered in more detail in chapter Updating Black Magic Probe Firmware (page 133). These are the commands that you need to give to reset the Black Magic Probe and have the operating system re-enumerate the device (for ctxLink, the address should be `0x08000000` instead of `0x08002000`):

```
dfu-util -d 1d50:6018,:6017 -e
sleep 0.5
dfu-util -d 1d50:6018,:6017 -s 0x08002000:leave
```

If you type the commands by hand, there is of course no need to insert a `sleep` between the two calls to dfu-util —your typing will be more than sufficient delay. However, if you put these commands in a shell script or batch file, a delay between the two commands is required.

TRACESWO Capture

If a trace monitor, such as the BlackMagic Trace Viewer (page 85), stays fully silent —no trace messages & no error messages, the first things to check is whether there is a good voltage on the VREF pin of the Black Magic Probe. The reason is that the TRACESWO pin goes through a voltage level shifter on the Black Magic Probe (like the other debug pins), and that the "target side" of this level shifter is powered through the VREF pin. If you power the target through the Black Magic Probe (see the `monitor tpwr` command,

[2] There is a `monitor reset` command; however, it resets the target micro-controller, not the Black Magic Probe.

page 50), this step is moot. If a trace monitor successfully attaches to the Black Magic Probe, this is also an indication that there is an adequate voltage at the VREF pin.

If you received TRACESWO output on an earlier launch, and it stops on a relaunch even though nothing else has changed, you may want to try resetting the Black Magic Prove, as covered in the preceding section.

Once you have established that the VREF pin is powered, you can subsequently check, with a logic analyzer or an oscilloscope, whether trace data is received at all on the TRACESWO line. Absence of data means that on the TRACESWO line means that SWO tracing has not been configured (or not configured correctly) in the target. Depending on the trace monitor, you may need to configure tracing from inside your source code, or you may need to run a particular script from GDB or some other tool.

If there is data on the TRACESWO line, check whether it is in the correct format and with a bit-rate that is in range with the capabilities of the debug probe. For instance, the Black Magic Probe hardware supports Manchester mode, whereas ctxLink only supports asynchronous mode. For Manchester encoding on the Black Magic Probe, the bit-rate is limited to approximately 200 kb/s; asynchronous encoding typically supports higher bit-rates.

If the trace monitor shows an error message along the lines of "access denied" or "failure opening device," this may indicate either a missing driver (on Microsoft Windows), or a missing `udev rule` (on Linux). See chapter Setting up the Black Magic Probe, and specifically the relevant section on either Microsoft Windows (page 20) or Linux (page 22), for details.

Another test that you can do is to redirect the TRACESWO data to the virtual UART interface of the Black Magic Probe. You can view it using a serial terminal. Use the following command —see also page 51:

```
(gdb) monitor traceswo decode
```

If data now shows up, it means that the target is configured correctly, and that the Black Magic Probe correctly receives the TRACESWO data. You can then focus on receiving the data on the dedicated interface.

RTT capture

The CoreSight architecture is designed such that the debug interface runs independently of the processor core. When the microcontroller drops into sleep mode, a debug probe can still access it, because the clock of the Debug Access Port (DAP) still runs.

Real-Time Trace, however, is implemented by having the debug probe poll a region in SRAM. SRAM is outside the microcontroller core, and whether a clock on the bus (AHB) is still running in sleep mode, is implementation-defined. For example, in case of the STM32 series of microcontrollers, the bits 0 and 2 must be set in a clock control register:

```
RCC_AHB1ENR |= SRAMEN | DMA1EN; /* keep SRAM + DMA1 enabled while sleeping */
```

Other microcontrollers may not provide a straightforward fix or workaround. In its knowledge base, SEGGER itself has this simple recommendation (emphasis is theirs):[3]

> **Solution**: When using RTT, make sure that low-power modes are **not used**.

TTL-Level UART

The Black Magic Probe has a TTL-level UART for general purpose communication with a device. This UART can be used together with the SWD interface or on its own.

A feature of the interface is that the RxD and TxD lines run through level shifters (just like the pins on the Cortex Debug Header). Thus, you can use the UART to interface with microcontrollers running at 5 V as well as at 3.3 V, 2.5 V… down to 1.2 V. The implication is, however, that there needs to be a voltage on the secondary side of the level shifters. This is why the UART connector (4-pin 1.25 mm pitch "PicoBlade") has a VCC pin —VREF would be a more accurate name. The logic voltage of the device should be connected to this pin.

The VCC pin on the UART connector is the same as the VREF pin on the debug header. When the VREF pin is powered from the Black Magic Probe (the `monitor tpwr` command), so is the VCC pin. If the attached target is running on 3.3 V, the wire to VCC is optional if you instead power the secondary side of the level shifters through the Black Magic Probe.

GDB on Microsoft Windows

GDB may optionally use an "index cache" to increase performance on debugging large executables. It stores this cache in the "home" directory of the workstation. To find the home directory, it uses the HOME environment variable. In Microsoft Windows, this variable is not set by default. Therefore, on launching GDB, you may be greeted with the warning:

[3] Source: https://wiki.segger.com/RTT#Low-power_modes

```
warning: Couldn't determine a path for the index cache directory.
```

This warning may be safely ignored; for executable files of the scale that fit in a microcontroller, you are unlikely to notice any reduced performance.

Alternatively, you can set the `HOME` environment variable before launching GDB:

```
SET HOME=%USERPROFILE%
```

or in PowerShell:

```
$Env:HOME = $Env:USERPROFILE
```

To keep the variable permanently set (instead of having to re-type it each time that you open a console to run GDB), you can add the variable to the list of static environment variables, in the System Properties dialog, TAB Advanced. After clicking on the button Environment Variables (near the bottom of the dialog), you will be presented with a new dialog with two lists of environment variables: one for the variables for the current user and one for the system variables (valid for all users). If you are the only user of the workstation, it makes no difference which one you take.

Microcontroller Driver Support

Microcontrollers frequently need some configuration to set up specific GDB functions or SWO tracing. For example, the section Flash Memory Remap on page 34 addressed a step that is needed before you can download code into the microcontrollers of the LPC families from NXP. In that section, we also recommended defining a command for that step in the `.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 microcontroller 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 on the (roaming) "Application Data" folder. This directory also holds the "INI" files that several utilities create (for storing their settings). 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 microcontroller 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 microcontroller's Flash memory map conforms to the ELF file layout —see also page 34.

`partid` To read the microcontroller's "device id" or "chip id."

`flashsize` To get the total Flash memory size of the microcontroller —this is exclusive to the STM32 microcontroller series at the moment.

`swo_device` For the MCU-specific configuration for SWO tracing.

`swo_trace` For the configuration for SWO tracing that is common to all ARM Cortex microcontrollers.

`swo_channels` To set the mask for enabled channels (for SWO tracing); this script is common to all ARM Cortex microcontrollers.

`swo_profile` To configure statistical profiling through the *Instrumentation Trace Macrocell* (ITM) and *Data Watchpoint & Trace* (DWT) units; this script is also common to all ARM Cortex microcontrollers.

`swo_close` To disable SWO tracing and profiling (i.e. reset ITM and DWT); this script is common to all ARM Cortex microcontrollers.

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

The only operations allowed on the registers (within a script) are assignment with operators `=`, `|=` and `&=`. These function in the same way as in GDB (and the C language). Numbers can be in decimal or hexadecimal notation, a "~" may prefix a value (or register) to denote the bitwise inversion of the value.

The operand on the left of the assignment operator is always considered an address. A literal number or a parameter on the right of the operator is considered a value. Thus, the literal or the value of the parameter is stored at the address at the left. The C *dereference operator* "`*`" can be used to interpret the value as a pointer, and read the value that the literal points to. A register does not need a dereference operator; it is always considered to be an address that needs to be dereferenced.

A parameter is specified as "`$0`" to "`$9`". The number of parameters and their values depends on the script. For the `swo_channels` script, for example, the single parameter `$0` is the bit mask of enabled channels. See the "`bmscript`" file that is provided with the utilities for details on the parameters for each script. When a parameter appears on the right side of an assignment, you may add both a shift value and a literal value behind it. The value of the parameter is then shifted left by the shift value and the literal value is merged with a binary "or" operation.

For a script that has a result (to be used by BMDebug, BMProfile, BMTrace or BMFlash), the result must be assigned to the special parameter "`$`" (no value following the dollar symbol).

```
define DBGMCU_IDCODE [ stm32f0* ] = {int}0x40015800

define partid [ stm32f0* ]
    set $ = DBGMCU_IDCODE
end
```

The assignment to the "`$`" parameter should also be the last statement in the script. At the moment, this is relevant only to the "`partid`" script (the other scripts do not return a value).

Tcl Primer

Two of the utilities that accompany this book, can be scripted to extend their functionality. The scripting language chosen for this project is Tcl.

The Tcl programming language started from a need for a general purpose scripting language, to enable users to extend the functionality of their applications with custom routines. The name reflects this: it stands for "Tool command language." The Tcl interpreter presented here, is a reduced version of standard Tcl.[1] It is based on ParTcl,[2] by Serge Zaitsev, but with various modifications. Both Serge Zaitsev's original implementation and my modified version are available on GitHub.

This chapter starts with a tour over the concepts and the "idiom" of the language. After that, separate sections on specific elements of the language give more details on these elements.

Syntax

The overall syntax of Tcl resembles that of Unix shell scripts. Instructions are lists of words that are separated by spaces. Strings between double quotes are also considered a "word" in the Tcl syntax —or more accurately, it is the other way around: words are strings, even if not between quotes. In interpreting it, the first word in a list is the *command*, with the other words as its arguments.

Double quotes are needed when a word contains a space or any other special characters. Tcl also offers another means of grouping words or "quoting" words: curly braces. Strings enveloped by curly braces can be nested, creating strings inside other strings —or lists of strings. A difference between the two forms of quoting is that in a double-quoted string, variables are substituted by their contents, but this does not happen in a brace-enveloped group.

```
set age 54
puts "I am $age years old"
puts {I am $age years old}
```

[1] The language has grown since it humble beginnings, and it is now increasingly used to create applications and utilities —rather than serving as an auxiliary embedded component of said applications.

[2] Tcl is commonly pronounced as "tickle"; the name ParTcl is a pun on this convention.

In the above example, if it were run, the first `puts` command prints "I am 54 years old," but the second prints the argument verbatim.

The above snippet has three instructions. The way Tcl goes through each, is in two stages. First, it collects the words for an instruction into a list, and then it evaluates (or interprets) that list, before proceeding with the next instruction. These phases are called *parsing* and *execution* respectively. Tcl moves from parsing to execution when it sees either a line end (*newline*) or a semicolon (";"). So when putting several commands on a single line, a semicolon is needed to separate them. There is an exception for newlines and semicolons inside quoted strings and brace-enveloped groups: these are not considered execution points.

Another key concept of Tcl is substitution. As the snippet above shows, a variable name prefixed with a "$" is replaced by its contents (except inside curly braces). Moreover, text between square brackets is replaced by the result of interpreting that text as a command.

```
set age 54
puts "It's [expr 65 - $age] more years until retirement"
```

The section "`expr 65 - $age`" is first extracted and interpreted. That is, the command `expr` is executed with the argument "`65 - $age`." The result of this simple calculation is then inserted at that position. Here it is a numeric result, but the same principle applies to commands that return strings.

There is no *assignment* operator in Tcl; the `expr` command only evaluates expressions and returns the result, and the `set` command sets a variable. The correct way to change the value of a variable, is like in the following:

```
set a 5
set a [expr $a * 2]
```

Control structures follow the syntax of commands. The following snippet swaps variables "a" and "b" if the former is greater than the latter:

```
if {$a > $b} {
    set tmp $a
    set a $b
    set b $tmp
}
```

The `if` command is a built-in command, and implemented such that it always evaluates the first argument as an expression. It is therefore not necessary to write it as "`if [expr {$a > $b}]`" (though this is still allowed). Also note in the sequence of `set` commands, that you should only use the "$" prefix when referring to the value of the variable, not when referring to the variable name.

The body of the `if` statement is a brace-enveloped group. The Tcl interpreter passes the entire content to the `if` command, as-is. The `if` command then decides (based on the condition in its first argument) whether to evaluate it, or whether to ignore it.

The rule for when to move from parsing phase to execution, is important for the coding style, notably the placement of braces. As written above, a newline or a semicolon mark an execution point, unless these appear inside a string or a group. The first line of the `if` snippet ends with a "`{`." Therefore, a brace-enveloped group has started and the newline that follows the "`{`" is *not* an execution point. Instead, Tcl reads up to the closing brace, and only then executes the `if` command.

The upshot is that we are thus not free to choose brace placement, as we are in C, Javascript, and many other programming languages. Also, white space between words and/or grouped blocks is often significant: there must be a space (or TAB) after a command name, for example; writing `if{$a < $b}` is incorrect (no space between `if` and the opening brace).

```
proc factorial x {
    if {$x == 1} { return 1 }
    return [expr {$x * [factorial [expr $x-1]]}]
}

factorial 4
```

The `proc` command adds a user procedure to the list of commands. It takes three arguments: the name for the new command, the parameter list, and the body for the user procedure. The parameter list is a series of names between curly braces (but if there is only one name in the parameter list, the curly braces may be omitted).

Variables must be set before being used, and variables that are inside a procedure are local to that procedure. When a procedure must keep global state, it must explicitly declare a variable as global.

```
proc random {} {
    # middle-square method to generate 4-digit numbers
    global random_seed
    set random_seed [expr {($random_seed ** 2 / 100 + 1234) % 10000}]
}

puts [random]
puts [random]
puts [random]
```

The `global` command marks variables that come behind it as global. The variable may already exist at global scope, but otherwise the `global` command creates it with an initially empty value.

A comment starts with a "`#`" and runs up to the end of the line, and must be placed on a line of its own, or following a semicolon.

Also note that the `random` procedure lacks a `return` command. If an explicit `return` command is lacking, the return value of a procedure is the result of the last command. Therefore, for simple cases like these, no `return` command is needed.

A final remark on the general syntax of Tcl is, that Tcl is case-sensitive. The built-in commands are all in lower case, and it would be an error to use `SET` instead of `set`. For your own user procedures and variables, you are free to choose the name, but you have to stick to it throughout the script.

Flow Control Structures

Tcl has various built-in control structures. The `if` was already briefly introduced, as a command that takes a condition and a brace-enveloped group of commands. It can, however, take a variable number of arguments. The complete syntax is:

```
if { condition } then {
    body
} elseif { condition } then {
    body
} else {
    body
}
```

A condition is *true* in Tcl if it evaluates to a non-zero value, when interpreted as a numerical expression.

The literal words `then`, `elseif` and `else` are all optional. You may insert them for clarity, or omit them for brevity. In practice, the `then` is traditionally omitted, but the `elseif` and `else` are put in. There may be any number of `elseif` blocks.

The `switch` command selects a body to execute, based on pattern matching. There are two syntaxes for the command; below is the most common one:

```
switch value {
    pattern {
        body
    }
    pattern {
        body
    }
    default {
        body
    }
}
```

The patterns can use wildcard characters "*," "?" and ranges between square brackets. The value is matched to each of the patterns, and on the first match, the relevant body is executed. All other bodies are skipped. If none of the patterns match, the `default` body executes. The `default` pseudo-pattern is optional; if it is present, it must be the last.

If a "–" follows the pattern (instead of a list of commands in curly braces), that body is a "fall-through" to the next body. This allows you to have a single instruction body for several patterns. The body (in curly braces) is set in the last of the patterns, and the patterns above it have a "–" behind the pattern.

For an example of the `switch` command, including fall-through cases, see the example script that concludes this chapter, page 176.

The basic command for loops is (`while` loop):

```
while { condition } {
    body
}
```

The loop keeps running the commands in its body as long as the condition is true. There are, however, a few other instructions that break out of loops. The `break` command causes a jump out of the innermost enclosing loop, and proceeds running at the command below the loop. All commands inside the loop body that follow the `break` are skipped. The `continue` command is similar to `break` (in that it skips all remaining commands in the loop body), but it jumps back to the loop condition. If the condition is still true, the loop will then continue.

The `break` command is also similar to `return`. In a way, `return` breaks out of procedures, quite like how `break` breaks out of loops. A final command that breaks out of the entire script is `exit` —it aborts running. Like `return`, `exit` may specify a return code.

As it is common for a loop to have a fixed number of iterations, there is a special construct for it:

```
for { setup } { condition } { post } {
    body
}
```

The instruction in "setup" is only evaluated once, before entering the loop. The condition has the same function as in the `while` loop: the `body` is only evaluated (i.e. executed) when the condition is true. After the body runs, the `for` command first evaluates the "post" word, before proceeding to the condition —to evaluate whether the loop must run another iteration. A typical use case is:

```
set total 0
for {set count 1} {$count <= 10} {incr count} {
    set total [expr $total + $count]
}
```

This loop runs ten times: the count variable starts at 1 and is incremented by 1 after every iteration.

The `break` and `continue` instructions can be used for the same purpose in a `for` loop as in the `while` loop, with the caveat that `continue` jumps to the "post" argument of the loop, rather than directly to the condition.

The last control structure is `foreach`, which loops over all items in a list. On every iteration, the variable in the first argument of the list is set to the consecutive item from the list.

```
set words [list the quick brown fox]
foreach w $words {
    puts $w
}
```

Numbers

Although the basic type in Tcl is a string, when arithmetic needs to be performed, these words are interpreted as numbers. Tcl supports three number bases:

- Decimal: a series of digits (between `0` and `9`), *not* starting with a `0`.
- Octal: a series of digits between `0` and `7`, prefixed with a `0`.
- Hexadecimal: a series of digits between `0`...`9` and between `A`...`F`, prefixed with `0x`. Hexadecimal numbers are *not* case-sensitive, so you may use `a`...`f` instead of upper case letters.

Lists and Strings

Lists were mentioned a few times, like how instructions are a list of words —the first word is the command and the successive words are its arguments. A list is not an explicit data structure in Tcl. Rather, lists are strings that are formatted in a particular way. More concretely, a list contains words that are separated by a space. A "word" in Tcl is a sequence of letters and/or digits, or a group of words enclosed in curly braces (or on occasion, enclosed by double quotes or square brackets).

There is, in essence, no difference between a string and a list. However, the distinction is made because Tcl offers a separate set of commands for list manipulation and for string manipulation.

Variables and Arrays

Simple variables have already been used in the snippets presented so far. A variable has a name and a value. Tcl attributes no type to the value; it can contain text or a number —or a list. Tcl imposes few restrictions on the variable name; a name like "`has-completed?`" would be invalid in most programming languages, but is perfectly valid in Tcl. Even spaces are permitted if you wrap the name in curly braces, like in "`{top level}`." When using the value of such a variable, put the "`$`" before the opening brace: "`${top level}`."

Yet, such special variable names are not recommended when the variables might also be used in expressions of the `expr` command. The infix expression evaluator has its own syntax, and variables with characters that conflict with operators, may confuse the evaluation.

When the variable name is prefixed with a "`$`," it is substituted by its value. From this follows that a variable must exist before it can be used.

Variables are created automatically when you set them, either with `set` or with another command that sets a variable. Variables set inside a use procedure are created as local variables; these cease to exist once the procedure ends. To access a global variable from within a user procedure, the variable needs to be declared inside the procedure with the `global` command. One or more variable names follow the `global` keyword. The `global` command creates a reference to each of the specified variables, but if the variable does not yet exist at the global level, it first creates it (with empty content). Once the global variable exists, it is not re-created or re-initialized by the `global` command.

A variable name can have an index appended. The index is a positive number between parentheses. The lowest valid index is zero. This allows a variable to have multiple values, each at a unique index.

```
for {set i 0} {$i < 10} {incr i} {
    set series($i) [expr 2 * $i]
}
```

Only single-dimension arrays are supported. In the full Tcl language, indices may be any text,[3] and their implementation is actually that of an associative *map*. In ParTcl, however, arrays need to be indexed with numbers.

Non-text data is often easier to process as an array. The `array` command enables conversion of text and binary data to an array of values. See the section Binary data (on page 164) for details.

Expressions

The `expr` command evaluates arithmetic expressions, like addition and multiplication. There are also relational and logical operators, and operators for bit twiddling. ParTcl does all arithmetic in integers; it does not support floating point. The relational operators ("==", "`!=`", "<" etc.) can compare strings (case-sensitive), but for matching with wild-cards or case-insensitive comparison, you need to use the `string` command instead.

There is no *assignment* operator; in Tcl you need to use the `set` command to assign a value to a variable. A few code snippets on preceding pages have already illustrated this —see for example the body of the `for` loop in the snippet on page 158.

The operator table (and precedence levels) of ParTcl are below:

`- + ! ~ ()`	unary operators: negate, unary plus (a no-operation operator), logic not, binary invert, and sub-expressions between parentheses
`**`	exponentiation
`* / %`	multiply, divide, remainder after division
`+ -`	addition, subtraction
`<< >>`	binary shift left & shift right
`< <= > >=`	smaller than, smaller than or equal, greater than, greater than or equal
`== !=`	equal, not equal
`&`	binary and

[3] Using this property, multi-dimensional arrays are simulated by a convention of joining indices with a "." separator.

^	binary exclusive or
|	binary or
&&	logic and
||	logic or
? :	conditional selection (ternary operator)

ParTcl (like Tcl) uses "floored" integer division. For positive numerators and denominators, floored division gives the same results as the (more common) truncated division: "14 / 3" is truncated to 4 (and with remainder 2). The difference is with negative results: "-14 / 3" with *floored* division gives -5 with remainder 1. Floored division is defined such that the remainder is always a positive value.

User procedures

The `proc` command takes three parameters. The first is the name for the user procedure. It is followed by a parameter list, which is a Tcl list of parameter names. These parameters are local variables inside the procedure. When creating a procedure without any parameters, you must explicitly declare an empty parameter list with `{}`. See also the examples on page 155.

The final parameter of the `proc` command it the *body*, which is a list of commands that will be evaluated when the user procedure is called.

```
proc name { parameters } {
    body
}
```

The user procedure itself results in a value. This can be explicitly given with the `return` command, which sets the "outcome" for the procedure and exits it. Alternatively (and quite common in Tcl code) is to use a `set` command just before the end of the procedure's body, since the result of the procedure is the value of the last command that ran. If a variable already has the correct value, you can skip the second argument of the `set` command, as the snippet below illustrates.

```
# Greatest Common Divisor, by means of Euclid's algorithm
proc gcd {p q} {
    while {$q != 0} {
        set r [expr {$p % $q}]
        set p $q
        set q $r
    }
    set p
}
```

On the last line of the body, there is no need to say "`set p $p`" —when called with a single argument, `set` returns the value of the variable *without* changing it.

Note, though that the "last command that ran" may not be the last command in the body. If, in the above snippet, the final "`set p`" were omitted, the last statement that ran, would be the evaluation of "`$q != 0`" in the `while` loop (via an implicit `expr` command) —and for the case that this expression evaluates to zero.

Parameters in the parameter list may specify a default value. This allows you to have optional arguments, when calling the user procedure. All parameters with a default value must be at the end of the parameter list; that is, when a parameter has a default value, any parameters that follow it must also specify a default value. To set a default value, enclose the parameter name and its default value in curly braces. In the example below, the user procedure `pow` takes either one or two arguments, and if no argument is passed for `exp`, this parameter is set to 2.

```
proc pow {base {exp 2}} {
    expr $base ** $exp
}
```

The special parameter name "`args`" collects all arguments beyond the fixed arguments. Thus, the user procedure accepts a variable argument list.

```
proc sum args {
    set total 0
    foreach v $args {
      incr total $v
    }
    set total
}
```

The `args` parameter must appear last in the parameter list —if used at all. It may be the only parameter, as in the above example. A variable argument list may be used in combination with parameters that have default values. The `args` parameter itself may not specify a default value.

Procedure parameters and variables set inside a user procedures are local to the frame of the procedure. A user procedure can access variables in frames lower down in the call chain with the `upvar` command. This command is mostly used to implement "pass-by-reference" arguments, where a procedure can modify a variable that is passed as a parameter.

```
proc decr {name {count 1}} {
    upvar $name var
    incr var [expr {-$count}]
}
```

The `decr` procedure is declared to take a name and (optionally) a count. It then uses the `upvar` command to create a local variable ("`var`") as a reference to a variable with the given name and which is one level lower than the frame for the `decr` command itself. Any operation on `var` in fact reads or writes the variable that it references.

```
set counter 10
decr counter
```

On the call to `decr`, the name "`counter`" is passed in. Thus, inside the `decr` procedure, `$name` equals to "`counter`," and `upvar` binds this name to local variable `var`. However, when changing `var`, it is actually `counter` that is modified.

This example illustrates the most common use of `upvar`, but it is more flexible. The `upvar` command may refer to a variable two (or more) levels up, and it may refer to an absolute frame level. To do this, the level may be specified as the first parameter after the command:

```
upvar level name variable...
```

When the level is a number, it is a *relative* from the current level; when it starts with a "`#`", the value behind it is taken as the *absolute* level. The usual case is `#0`, meaning the global level.

The last point is that pairs of reference names and local variable names may be declared on a single `upvar` command. This allows you to create multiple references at once (on the same level). The reference variables must exist on the targeted level —unlike the `global` command, the `upvar` command does not *create* variables at a lower level.

Comments

The "`#`" character starts a comment, which runs up to the end of the line. However, a comment may only appear at an "execution point," which is either after a newline or after a semicolon. In practice, it means that you can place a comment on a line of its own, or alternatively —if you want to add a trailing comment behind a command, place a semicolon in front of the `#`.

Exception handling

A run-time exceptions occurs when attempting to perform an operation that cannot proceed, such as opening a file that does not exist, or using a variable that was never set. With Tcl, you would normally be able to avoid such errors —for example, by first checking the existence of a file before opening it, with the "`file exists`" command. However, it is often more convenient to catch the exception and handle it when it occurs.

```
if [catch {set fd [open $filename]} errmsg] {
    puts "Error: $errmsg"
} else {
    puts [read $fd]
}
```

The `catch` command evaluates its first argument, which is the command "`set fd [open $filename]`". This in turn evaluates the nested command "`[open $filename]`" first. If variable `filename` does not hold a name of a file that can be opened, the `open` command throws an exception. The exception cascades through the `set` command, and would eventually abort the script —but it the `catch` command stores the error message in the `errmsg` variable (the second argument to `catch`) and clears the error.

The result of the `catch` command indicates whether an exception occurred. It returns one of the following values:

- `0` Normal return.
- `1` Error or exception occurred, `$errorInfo` holds the message.
- `2` Abort due to a `return` or `exit` statement.
- `3` Abort due to a `break` statement.
- `4` Abort due to a `continue` statement.

In practice, `catch` will not return values 2, 3 or 4 for well-written code, because these cases have already been handled by procedure and loop commands. However, if you use `break` outside a loop, `catch` may indeed... well, *catch* that.

In your own code, you may throw an exception with the `error` command. This command takes a message as an argument, which is the message that `catch` will subsequently store in the variable.

Binary data

Tcl offers two ways to extract values from binary data: the `array` and `binary` commands. If you have a chunk of binary data in a variable called "`blob`," you can convert it to a byte array with:

```
set blob 12345
array slice data $blob
```

The "data" variable will have as many entries as the length of the contents of "blob." Each entry in data has the value of the respective byte in blob. In this example, data has five entries, from data(0) to data(4); where data(0) is set to 49, data(1) to 50, and so forth up to 53 for data(4). In other words, the first character of blob is "1," which has ASCII code 49, and thus 49 is stored in the first array element.

The array slice command can also chop up the blob in chunks of 16, 24, 32 or 64 bits. The "word size" (1 to 8) can be optionally appended at the end of the command. When the word size is not 1, the default is that the data is sliced in Little-Endian order (low byte first). Optionally, the argument "be" can be appended behind the word size, for Big-Endian byte order.

When the binary data has a mix of fields with different sizes, the binary command is suitable. The command takes a "format" argument that allows you to specify the type, size and count of each subsequent field. There are two sub-commands: format is to pack Tcl values into a binary blob, and scan is to unpack a binary blob into Tcl variables.

The snippet below illustrates the binary scan command, to interpret a blob as a series of bytes, and set variable data to a list of individual byte values. After the command, data is set to {49 50 51 52 53}.

```
set blob 12345
binary scan $blob cu* data
```

In this example, array slice and binary scan perform essentially the same function —the only difference is that binary scan creates a list, rather than an array. However, the format string ("cu*" in the above example) offers a lot of flexibility. For example, the "Read Input Registers" frame from the MODBUS-RTU protocol can be decoded with the following format pattern:

```
# suppose blob contains the byte sequence 01 04 00 00 00 02 71 cb
binary scan $blob cucSSsu address function register number crc
```

The device address, function code, register start address, register count and CRC that are extraced from the blob, are all stored in separate variables. The format string has a field for each of the five variables that follow. The first letter gives the size and byte order of the field.

- c 8-bit integer
- s 16-bit integer
- i 32-bit integer
- w 64-bit integer

When this letter is upper case, the field is set in Big-Endian; otherwise it is set in Little-Endian. The upper case "C" is undefined, because byte-order is irrelevant for a single-byte field.

The letter "u" may follow the leader letter, to indicate that the integer has an unsigned value (the default is signed). After that, a number may follow for the count of these integers to store in the respective variable. If there is a "*" instead of a number, it means that it runs up to the end of the data in the binary blob.

Referring tp the MODBUS-RTU example, `address` is an 8-bit unsigned integer ("cu"), whereas `function` is an 8-bit *signed* integer ("c"); `register` and `number` are both 16-bit signed integers in Big-Endian ("S"); and `crc` is a 16-bit unsigned integer in Little-Endian ("su").

Built-in commands

Several of the built-in commands have subcommands —for example, the `file` and `string` commands. In the following table, these subcommands are listed separately.

A few other command accept switches. Switches are optional parameters that change the operation of the command. An example is the `puts` command. It normally ends the argument that it prints with a newline; however, when the switch `-nonewline` is added to the command, `puts` prints the argument without newline.

Switches must be placed after the command, but before the first normal argument. If a command takes a subcommand, the switches must be placed after the subcommand. If a switch appears at an incorrect position, or if a switch is not recognized as valid, it is taken to be a normal argument.

append *var word*	Append contents to a variable (concatenate strings).
array length *var*	Return the number of elements in the array.
array size *var*	Same as `array length`.
array slice *var word*	Slice the word into bytes or multi-byte fields, and store the values (when interpreted as binary data) into array elements. See page 164.
array split *var word*	Split the string on a separator and store the elements in an array. If no *sep*arator is given, the string is split on whitespace.
array split *var word sep*	

binary format *fmt arg ...*	Return a string with the binary representation of the arguments, and according to the type specifications in the *fmt* parameter.
binary scan *word fmt var ...*	Extract values from binary data in *word*, according to the type specifications in the *fmt* parameter, and store these in the variables listed at the tail of the command. See page 165.
break	Abort a loop, jumps to the first instruction below the loop.
catch *body* **catch** *body var*	Evaluate the body, catch any error; return 0 if the body evaluated normally, and 1 if an exception occurred. The error message is stored in variable *var* (if given). See page 164.
clock seconds	Return the number of seconds since the Unix Epoch (00:00:00 January 1st, 1970).
clock format *time format*	Format time and date according to the *format* string. The *time* parameter is the number of seconds since the start of the Unix Epoch.
close *file*	Close the file indicated by the file handle.
concat *word ...*	Join multiple lists into a single list.
continue	Skip the remainder of the loop body, jumps back to the start of the loop.
eof *file*	Return 1 if the file (specified by handle) is at its end.
error *msg*	Set an exception or error, which aborts execution (but which can be caught with **catch**). See page 164.
exec *word ...*	Run the parameter as an executable in the shell, with any additional words as the arguments to the program. The command returns the console output of the program.
exit **exit** *word*	End the script with an optional return code. Note that this command aborts the script, but not the application.
expr *expression*	Interpret the infix expression that follows. It supports only integer arithmetic. See page 160.
file dirname *path*	Return the directory part of the path.
file exists *path*	Return whether the path exists (0 or 1).
file extension *path*	Return the file extension of the path.
file isdirectory *path*	Return whether the path refers to a directory (0 or 1).
file isfile *path*	Return whether the path refers to a regular file (0 or 1).
file rootname *path*	Return the part of the path *without* extension.
file size *path*	Return the size of the file.
file tail *path*	Return the base name of the path (*without* directory).
firmware length	Return the size in bytes of the firmware.
firmware read *address size*	Read binary data from the firmware.
firmware write *address data*	Write binary data to the firmware. This command is only available in the BMFlash utility; see Scripting the BMFlash Programmer on page 171 for an example.
flush *file*	Flush buffered data to the file.

for *setup cond post body*	Evaluate *setup*, then run *body* in a loop as long as *cond* stays true. At the end of every iteration, *post* is evaluated. See page 158.
foreach *var list body*	Run a loop over all elements in *list*. Each time that *body* is evaluated, *var* is set to the next element from *list*. See page 158.
format *string word ...*	Format a string with placeholders, similar to `sprintf` in C. Currently "%c," "%d," "%i," "%x" and "%s" are supported, plus optional padding and alignment modifiers (for example "%04x" or "%-20s").
gets *file*	Read a single line from the file; the trailing newline is stripped.
global *var ...*	Mark any variable following it as a global variable. Multiple names may follow, separated by spaces.
if *cond* **then** *body* **elseif** *cond* **then** *body* **else** *body*	Conditional execution of *body*. The keywords **then**, **elseif** and **else** are optional. See page 156 for details.
incr *var value*	Increment a variable by *value*. If the *value* parameter is omitted, *var* is incremented by 1.
info exists *var* **info tclversion**	Return 1 if the variable exists, and 0 otherwise. Return the version of the Tcl interpreter.
join *list* **join** *list separator*	Create a string from a list, by concatenating elements, with a separator chosen by the user. If the *separator* parameter is omitted, a space is used for separation.
lappend *var word ...*	Append values to a variable (where the variable is presumed to contain a list).
lindex *list index*	Return a specified element from the list (parameter *index* must contain a valid element number, between 0 and the list length minus 1).
linsert *list index word ...*	Insert elements in a list in front of the value of *index*. The first list element has index 0.
list *word ...*	Create a list from the values that follow it.
llength *list*	Return the number of elements in a list.
lrange *list first last*	Return the elements *first* to *last* (inclusive) of *list* as a new list. Parameter *last* may be set to "end" (instead of a number) to indicate the end of the list.
lreplace *list first last ...*	Delete the elements *first* to *last* (inclusive) from *list* and insert a set of elements at that position. If there are no new elements behind parameter *last*, it deletes the elements between *first* and *last*.
lsearch *list pattern*	Find the index of the first element in the list that matches the pattern. The *pattern* argument may contain wildcard characters "*" and "?", or character sets or ranges between square brackets. However, if the switch `-exact` is set, wildcards are disabled and all characters must match.

lsort *list*	Sort the elements of a list, returning a sorted list. The default is an alphabetic sort in increasing order; switches `-integer` and `-decreasing` toggle these settings.
open *path mode*	Open a file and return a file handle (this file handle must be passed to other file commands, like **close** or **puts**).
proc *name* args body	Create a new (user-defined) command. See page 161.
puts *word*	Print the argument to the stdout.
puts *file word*	Print the argument to the *file* (handle). The command ends the output with a newline, unless the option `-nonewline` is set.
read *file*	Read the complete file as a string.
read *file count*	Read a maximum of *count* bytes from the file and return it as a string. If the switch `-nonewline` is set, a final newline character (if any) is stripped from the returned data.
return	Jump out of the current command ("proc"), with an optional explicit return value.
return *word*	
scan *word format var ...*	Parse a string and stores extracted values into variables. This command currently supports "%c," "%d," "%i" and "%x" placeholders, plus optional "width" modifiers (for example "%2x").
seek *file position*	Set the read/write position of the file.
seek *file position whence*	The *whence* parameter can be set to `current` or to `end` to move the file position relative to these markers.
serial cache *number*	Cache part of the received data that follows *number*, so that it is prepended to `$serialrecv` on the next run. The bytes before *number* are dropped. If the *number* is omitted, *all* data is cached.
serial gobble *number*	Gobbles up (removes) the specified number of bytes from `$serialrecv`. If the number is not given, *all* data is gobbled.
serial transmit *text*	Transmits the text in the parameter. The text is transmitted as-is. No line ending is automatically appended; if you wish to add carriage-return or newline characters, these must be in the *text* string. This command is only available in the BMSerial terminal utility; see BMSerial Terminal, page 176 for an example using the **serial** command.
set *var word*	Assign a value to the variable, and return this value. If no *word* parameter is present, the current value of the variable is returned.
source *path*	Read the file and evaluates it as a Tcl script. It returns the value of the last command in the file, or that of a **return**.
split *word*	Create a list from a string, by splitting the string on a separator chosen by the user. If no *separator* is given, the string is split on whitespace.
split *word separator*	

string compare *word word*	Compare two strings, returns an order ranking value (0 if both strings are equal).
string equal *word word*	Test strings for equality (returns 1 if equal, 0 otherwise).
string first *word sub skip*	Finds the first occurrence of *sub* in *word*, skipping the first *skip* characters in *word*.
string index *word value*	Return the character in *word* at the given index.
string last *word sub skip*	Finds the last occurrence of *sub* in *word*, skipping the last *skip* characters from the end of *word*.
string length *word*	Return the length (in characters) of the string.
string match *pattern word*	Return 1 if the pattern matches the word, and 0 otherwise. The pattern may use "*" and "?" wildcards, plus character sets or ranges between square brackets.
string range *word first last*	Return a string that has the range of characters between *first* and *last* (inclusive). Parameter *last* may be set to "end" to indicate the end of the string.
string tolower *word*	Returns the *word* string in lower case.
string toupper *word*	Returns the *word* string in upper case.
string trim *word charset*	Removes characters in *charset* from the start and end of the string. The *charset* parameter defaults to whitespace.
string trimleft *word chars*	Like **trim**, but only trim the start of the string.
string trimright *word chars*	Like **trim**, but only trim the end of the string.
subst *word*	Perform command and variable substitution in the parameter.
switch *word* **{** *pattern body* *pattern body* **default** *body* **}**	Control flow structure, executing a block selected from matching one out of several patterns. The patterns may use "*" and "?" wildcards, plus sets or ranges between square brackets. However, the `-exact` switch disables wildcards. The **default** clause is optional (it is taken if none of the patterns match). See page 156.
syscmd *command*	Transmits the command to the debug probe using the GDB RSP protocol. This command is only available in the BMFlash utility; see Scripting the BMFlash Programmer.
tell file	return the current position of the file.
unset *var ...*	Clear variables (remove the given variables completely).
upvar *name var ...* **upvar** *level name var ...*	Create a "reference variable" to a variable at a lower scope, so that setting the local variable changes the value of the referenced variable. See page 162.
wait *value* **wait** *var value* **wait** *var value body*	A delay for the time specified in milliseconds. Wait until the variable ("*var*") changes, but time out after the number of milliseconds in the *value* parameter. On a timeout, the optional *body* is evaluated. This command is only available in the BMFlash and BMSerial utilities.
while *cond body*	Run a loop as long as the condition is true. If the condition is already false on start, the body is never evaluated.

Further Reading

This primer is brief for a reason: so much fine information on Tcl is already available in on-line tutorials and books. John Ousterhout, creator of Tcl, wrote a very readable, and comprehensive book on it. A draft of the first edition is freely available in PDF format. See chapter Further Information on page 182 for a reference.

Old books are fine, because, as stated earlier, ParTcl actually draws back to the roots of Tcl: as a light-weight extension language for applications. It is closer (in syntax and semantics) to Tcl versions before 7.0 than to the current 8.6 release.

Keep in mind that the expression parser in ParTcl is integer-only. There is no floating point arithmetic, and operators that function on lists are also unavailable. Another limitation of ParTcl is that arrays in ParTcl must be indexed with a number (equal to, or greater than zero); non-numeric array indices are not supported.

Application-Specific Extensions

Applications generally supplement the standard Tcl commands with their domain-specific commands and variables.

Scripting the BMFlash Programmer

The BMFlash utility is covered in the chapter Firmware Programming, and specifically section Using the BlackMagic Flash Programmer on page 124. Scripts can be selected to run both before and after a download (the "Preprocess" and "Postprocess" fields).

Before running a script, the following predefined variables are set:

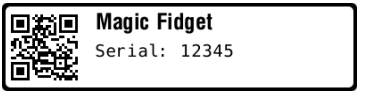
`$cksum`	The checksum of the ELF file (as reported by the `cksum` utility).
`$ident`	The RCS identification string (as reported by the `ident` utility), or an empty string if no indentification string was found.
`$filename`	The full name of the ELF file (or HEX/BIN file) that was downloaded.
`$serial`	The serial number set in the file, or an empty string if serialization is not active.
`$status`	For the postprocess script, the status of the download; set to 1 if the download was successful, or 0 on failure. Note that you must explicitly configure the postprocess script to also run after failed downloads.

Extra commands for BMFlash are `firmware`, `syscmd` and `wait`, as already listed in section Built-in commands (page 166).

To start with an example, below is a script that prints a label with the serial number on a Zebra label printer (or compatible). The idea is that when using serialization, the download and the labeling of the PCB are done in a single process —so that labels are not accidentally swapped.

```
set template {
  ^XA
  ^FO7,5,0
  ^GB398,100,4,B,1^FS
  ^CF0,25
  ^FO112,17,0^FDMagic Fidget^FS
  ^FO17,17,0^BY1,2,1^BQN,2,3,Q,7^FDQM,20Magic Fidget $serial ^FS
  ^CFA,20
  ^FO112,52,0^FB285,6,0,L,0^FDSerial: $serial ^FS
  ^CF0,20
  ^XZ
}

set filename "label.zpl"
set fhandle [open $filename wt]
if $fhandle {
  puts $fhandle [subst $template]
  close $fhandle
}
exec "lprint $filename"
```

The label is hard-coded in this script, in the "ZPL" language for Zebra printers, but it could equally well have been loaded from a template. The "`$serial`" variable, references in the label template is then replaced with its value —twice, in fact: once as readable text and once as part of a QR code. The adjusted ZPL instructions are written to a file, and in the last step that file is sent to the label printer with the LPrint utility.[4]



A common use of a post-download script is to have the device run a self-test. For this purpose, the `syscmd` command comes handy: it enables you to send debugger commands (running, setting breakpoints, ...) to the target microcontroller, and read back the responses.

[4] "LPrint" is a label printer application for macOS and Linux, by Michael Sweet; see https://www.msweet.org/lprint/lprint.html

The `syscmd` command uses the commands of the *Remote Serial Protocol* (RSP), which are different from GDB commands —the BMFlash talks to the Black Magic Probe directly instead of using GDB. For the RSP commands and syntax, see the GDB manual. The response from the debug probe is stored in variable `$sysreply`. Note that this variable is only updated when the script "waits" for a reply (the debug probe runs asynchronously from the Tcl script; the `syscmd` command returns immediately).

For example, the following resets the firmware, checks the response, and either starts running (with a second command) or prints the error code:

```
syscmd "vRun;selftest"           ;# reset, passing "selftest" as an argument
wait sysreply 1000
if [string match E* $sysreply] {
    puts "Error $sysreply"
} else {
    syscmd "c"                   ;# start running
}
```

The text following the semicolon in the `vRun` RSP command, "`selftest`" in the above example, is passed to the firmware as a "command line" argument. The firmware can query this string with a semihosting call (specifically: `sys_get_cmdline()`). Semihosting is covered in section Semihosting: target to host I/O bridge (page 57). Building on the "`semihosting()`" function developed in that section, a function to request the "command line" is:

```
void sys_get_cmdline(char *buffer, size_t size)
{
  uint32_t params[2] = { (uint32_t)buffer, size };
  semihosting(SYS_GET_CMDLINE, params);
}
```

When the firmware is not running under a debugger, the "command line" is an empty string

Via the Remote Serial Protocol, the Tcl script can capture semihosting operations that the firmware issues, and act on it. A typical example is to print the "console output" of the firmware to the BMFlash log view.

```
syscmd "vRun;selftest"           ;# reset, passing "selftest" as an argument
wait sysreply
if [string match E* $sysreply] {
    puts "Error $sysreply"
} else {
    syscmd "c"                   ;# start running
}
```

```
while 1 {
    wait sysreply
    # semihosting command has the format "F<cmd>,<arg>,<arg>,..."
    if [string match Fwrite,* $sysreply] {
        set tail [string last "," $sysreply]
        set arg [string range $sysreply [expr $tail + 1] end]
        puts [string trim $arg]
        if [string equal $arg completed] {
            exit                ;# completed tests
        }
    }
}
```

The first part of the script is the same as the earlier snippet: it launches the firmware with "`selftest`" as the "command line" argument. In the second part, the script enters a loop in which it waits for responses sent by the firmware. In the *Remote Serial Protocol*, semihosting operations are forwarded in so called *F* packets —indeed these packets start with the letter *F*). The Tcl script checks for "`Fwrite`" and (on a match) prints the last argument of the command: the message text. However, if that text is "`completed`," the script also exits.

The preprocessing script lets you modify the firmware before it is uploaded. For an example of when and how you would use it, suppose that you want to copy-protect your firmare. Consider that the real value of your product may well be in the software —electronics are relatively inexpensive. Electronics circuits are also relatively easy to reverse-engineer: anyone buying your device can screw the lid off and identify the various components and how these are connected. While there exists methods to "harden" the hardware (which may be as easy as potting the electronics in epoxy), but there are drawbacks (think of testing, inspection, repairability, . . .).

So you might want to fortify the software instead. Specifically, you will want to block the option that the firmware of your product ends up in cloned hardware. If the microcontroller supports it, the first step is to enable code protection. However, many of these mechanisms have already been broken. As an example, the CRP layer in the LPC series of microcontrollers (from NXP) is easily broken by power glitching, using an inexpensive *ChipWhisperer device* —read *The Hardware Hacking Handbook*[5] for the gory details. The point is, the hack only needs to succeed

[5] Van Woudenberg, Jasper & Colin O'Flynn; *The Hardware Hacking Handbook*; No Starch Press; 2021; ISBN 978-1593278748.

once, and then the firmware can be extracted and be copied into as many cheap clones as desired.

The next level is then, to ensure that the extracted firmware is somehow linked to the exact product that it was obtained from —a "fingerprint" of the hardware is embedded in the code. So that the extracted firmware won't run (or won't run correctly) in the clone, because of the fingerprint mismatch. What I am aiming at here specifically, and sticking to the example of the LPC microcontroller series, is to store the *unique* ID of the microcontroller into the firmware during the download. When running, the firmware matches the UID of the microcontroller to its stored value, and reports a fault on a mismatch.

```
set salt      0x2980002B  ;# secret "salt" value for the hash
set uid_addr  0x00000300  ;# reserved memory address for the hash

# get the UID (check the response)
syscmd qRcmd,readuid
wait sysreply

if [string match "UID: 0x*" $v] {
  # run a utility to return a hash for the UID (plus secret salt)
  set uid [string range $sysreply 7 end]
  append uid [format "%08x" $salt]
  set hash [exec md5gen -hex $uid]
  # split the hash into 4 dwords
  scan $hash "%8x%8x%8x%8x" md3 md2 md1 md0
  # store the words in the firmware, at a location reserved by the linker
  firmware write $uid_addr [binary format iuiuiuiu $md0 $md1 $md2 $md3]
}
```

From a high level, the script first requests the UID from the attached microcontroller via GDB RSP protocol —the command `syscmd qRcmd` sends a "`monitor`" command to the target. After checking that the response is valid, and obfuscating the UID a bit, it stores it at a fixed address in the firmware; this is the last command in the script: `firmware write`. The address is chosen to come right behind the CRP value (for Code Read Protection) on the LPC microcontroller series. This memory range must be reserved in the linker file.

Obfuscation increases the difficulty and cost of the attempts to crack the software. The goal is that the attackers won't find the UID itself in Flash ROM —because that gives them the location of the key to unlock the software. So instead the script stores the MD5 hash of the UID, after *salting* it with a secret value. In the script, the "salt" is hard-coded, but the objective is that this value is constructed from (peripheral) registers —so that

it isn't stored in Flash ROM either. The value of the salt chosen in this script, is the "Device ID" value for the LPC11U24/401 microcontroller.

In the firmware, you will repeat these steps and compare the results. But you will also need to take measures against reverse engineering. After all, another route that an attacker may choose is to patch the firmware to bypass the validation test. Anti-debugging tricks and measures against static analysis may be the next step in hardening your code, but these topics are well beyond the scope of this book.

BMSerial Terminal

The second utility that supports Tcl scripting is BMSerial: a serial terminal. This utility adds an application-specific command and a predefined variable.

`serial cache` *`number`*	Cache part of the received data that follows *`number`*, so that it is prepended to `$serialrecv` on the next run. The bytes before *`number`* are dropped. If the *`number`* is omitted, *all* data is cached.
`serial gobble` *`number`*	Gobbles up (removes) the specified number of bytes from `$serialrecv`. If the number is not given, *all* data is gobbled.
`serial transmit` *`text`*	Transmits the text in the parameter. The text is transmitted as-is. No line ending is automatically appended; if you wish to add carriage-return or newline characters, these must be in the *`text`* string.
`$serialrecv`	This variable is predefined to hold the new data that was received since the last call.

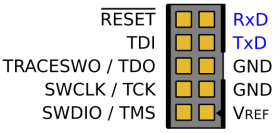
The Tcl scripting in BMSerial serves two purposes: protocol decoding and automatically responding to certain input data. As an illustration, below follows a protocol decoder for the Dynatek multimeter range with RS232-support.

```
# Dynatek multimeter protocol decoder (tested with Dynatek 6080RS)
#
# 1) Turn the multimeter off before inserting the serial cable.
# 2) Press and hold the REC and COMP buttons, then select the function with the
#    rotary button.
# 3) Serial settings: 1200 bps, 7 data bits, 1 stop bit, no parity. The DTR
#    line must be set and the RTS line cleared, for the multimeter to send data.

while {[string length $serialrecv] < 12} {
  wait serialrecv 100 exit ;# A Dynatek frame has 12 bytes
}

set type [string index $serialrecv 0]
set range [string index $serialrecv 2]
set valid 1
```

```
if {![string match \[0-9AB\] $type]} { set valid 0 }
if {![string match \[0-5\] $range]} { set valid 0 }
if {![string equal [string index $serialrecv 11] \r]} { set valid 0 }
set value ""
for {set pos 3} {$pos < 12} {incr pos} {
  set digit [string index $serialrecv $pos]
  if {![string match \[.0-9\] $digit]} {
    if {[string equal $digit O]} { set valid 0 }
    break
  }
  append value $digit
}

set unit ""
switch $type {
  0 { set unit "V/AC" }
  1 { switch $range {
        0 { set unit "Ohm" }
        1 -
        2 -
        3 { set unit "kOhm" }
        4 -
        5 { set unit "MOhm" }
      }
    }
  2 { set unit "V/DC" }
  3 { set unit "mV/DC" }
  4 { set unit "A/AC" }
  5 { set unit "A/DC" }
  6 { set unit "V" }
  7 { set unit "mA/DC" }
  9 { set unit "mA/AC" }
  A { switch $range {
        0 { set unit "nF" }
        default { set unit "uF" }
      }
    }
  B { switch $range {
        0 { set unit "Hz" }
        default { set unit "kHz" }
      }
    }
}
if {$valid} {
  puts "$value $unit"
}
# Gobble up to (and including) the frame delimiter, cache the remainder (if any)
set pos [string first $serialrecv \r]
if {$pos >= 0} {
  serial cache [expr $pos + 1]
}
```

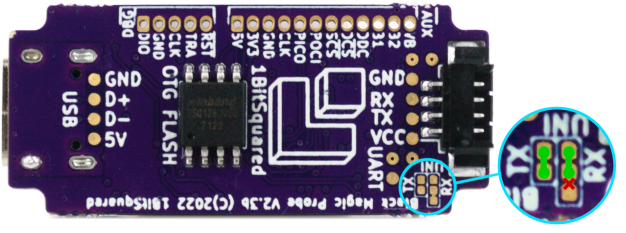
Unified Connector: Debug + UART

Hardware version 2.3 of the Black Magic Probe has the option to link the UART TxD and RxD to pins 7 and 9 of 10-pin Cortex Debug connector. The black-magic project calls it the "unified connector" —or BMDU, which stands for Black Magic Debug Unified.

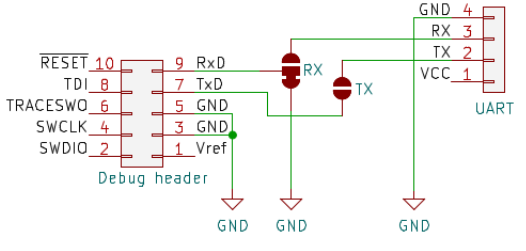


When comparing the pin-out of the unified connector to that of the standard Cortex Debug header on page 26, you will note that RxD replaces a ground pin, and TxD is on an originally unconnected pin.

To make these connections, two pairs of "jumper" pads on the back of the Black Magic Probe must be joined with a dot of solder, and a trace between another pair of jumper pads must be cut. The image below shows the locations of the joints and the cut, in the lower right corner. The operation is reversible, but it requires a soldering iron.



For reference, the wiring between the two connectors and the jumpers is illustrated schematically as well. Note that the pins marked VREF (on the Debug header) and VCC (on the UART connector) are actually connected to each other as well, though this is not drawn.

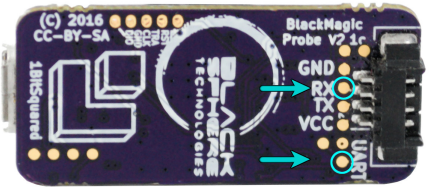


The unified connector will only be functional if the target device uses it too. If the target uses the standard pin-out of the Cortex Debug header,

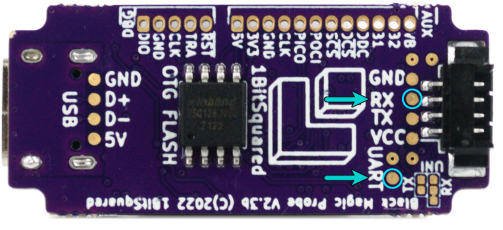
it will link RxD to ground. Serial communication will then not work, neither via the unified connector nor via the 4-pin UART connector (RxD of the UART connector is linked to RxD on the unified connector —which is pulled to ground by a target that uses the standard pin-out). Pin 7 on the Cortex Debug header is officially the "key" pin, intended to avoid incorrect alignment or orientation of the cable connector on the header. Instead of cutting off the pin, the Black Magic Probe simply left it unconnected, and this is actually common practice.

Linking TRACESWO to UART-RxD

Part of why this book exists is for our own reference in how to configure and use the Black Magic Probe and its surrounding utilities. That may already have been conspicuous in the coverage of the ARM Cortex M0/M0+ architectures and the peculiarities of the NXP LPC series of microcontrollers —we happen to use these low-end microcontrollers a lot. Presented here in this chapter, is a small modification that we make to the Black Magic Probes that we own, and which makes an enhancement of the Jeff Probe available on the Black Magic Probe.



The modification is that we add a miniature slide switch (specifically: TE Connectivity product MLL1200S) that optionally connects the TRACESWO pin on the Cortex Debug connector to the RxD pin on the TTL-level UART connector. The switch is glued to the bottom of the PCB and two of its three pins are soldered to the test pads indicated with the light blue circles. The image above is for hardware version 2.1; the one below is for hardware version 2.3.



Linking TRACESWO to RxD enables the Black Magic Probe to receive the asynchronous SWO tracing protocol, although it now receives it via the UART interface instead of through the dedicated raw-data interface. Alternatively, it allows you to use UART tracing (page 75) over the debug connector, so that you only need a single cable between the debug probe and a Cortex M0 microcontroller (which lacks support for SWO tracing). The criterion for a “single cable” is especially relevant for our use of the tag-connect cable, see page 16.

The Jeff Probe can switch the pins in software, which is more convenient that a hardware switch. The `monitor convert_tdio` command maps the

TRACESWO pin to RxD and the TDI pin to TxD, see page 53.

Note that when the TRACESWO and RxD signals are linked, you can no longer use the secondary UART concurrently with SWO tracing if you do, the target device would see the TRACESWO output shorted to TxD. This is the reason for adding a switch in the connection —so to be able to choose between either standard Manchester SWO and a secondary UART, or asynchronous SWO (and forgo the UART).

This "patch" is independent of the modification to come to a unified debug connector, in the preceding chapter. You can do both adjustments at the same time. In principle, the unified connector solves the same problem as this patch. However, our motivation is to be able to use standard (and readily available) cables, and specifically the tag-connect cable.

Further Information

Hardware

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

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

ctxLink: can be obtained from:

Sid Price	http://www.sidprice.com/ctxlink/
Crowd Supply	https://www.crowdsupply.com/sid-price/ctxlink
Mouser	https://www.mouser.com/

Jeff Probe: can be obtained directly from Flirc:
https://flirc.tv/products/flirc-jeffprobe

3D Printed Enclosures for the Black Magic Probe can be found on the Thingiverse and Printables websites. A simple clip that offers some protection for the 10-pin Cortex Debug header (see page 21) is "thing" 2387688 (by Michael McAvoy); a full enclosure with openings for the connectors, LEDs and button is "thing" 2836934 (by Emil Fresk). Both are for version 2.1 of the Black Magic Probe.
https://www.thingiverse.com/
An enclosure with an extra mound around the 10-pin Cortex Debug connector is on Printables with id 52069 for version 2.1, and id 359916 for version 2.3 of the Black Magic Probe
https://www.printables.com/model
Designs for ctxLink enclosures are available on Sid Price's GitHub page:
https://github.com/sidprice/ctxLink_cases

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 page 16.
https://www.tag-connect.com/
freeconnect: open-source a pogo-pin connectors by Rafael Silva, as an alternative to tag-connect cables.
https://perigoso.github.io/freeconnect/

PCBite: PCB holders and needle probes:
https://sensepeek.com/

Software

Black Magic Probe: The project website contains links to downloads, documentation and schematics:

https://black-magic.org/

Active development of the firmware happens on the GitHub project.

https://github.com/blackmagic-debug/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/2020/03/24/

http://www.sidprice.com/2018/05/23/cortex-m-debugging-probe/

bmputil: A tool for flashing Black Magic Probe firmware.

https://github.com/blackmagic-debug/bmputil

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

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

Zadig: A utility for installing the drivers for SWO tracing and firmware update, see chapter Setting up Black Magic Probe, specifically page 20.

https://zadig.akeo.ie/

libusbK: Drivers, support DLLs and development files for generic USB device access.

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

Orbuculum: A set of utilities to process the output ARM Cortex Debug interface (SWO tracing, exception trace, performance profiling, . . .), see also page 76.

https://orbcode.org

gdbgui: Various GDB front-ends were mentioned in chapter Requirements for Front-ends (page 13), 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/

Troll: A source-level debugger supporting the Black Magic Probe, that is independent of GDB.

https://github.com/stoyan-shopov/troll

turbo: A GDB front-end with specific support for the Black Magic Probe, by the author of the Troll debugger.

https://github.com/stoyan-shopov/turbo

Cortex Debug: Visual Studio Code extension that adds debugging functions, with (initial) support for the Black Magic Probe.
https://github.com/Marus/cortex-debug

GnuWin32: The GnuWin32 project has native Windows ports of several GNU utilities, like the core utilities and RCS.
http://gnuwin32.sourceforge.net/

SvnRev & Autorevision: Utilities to extract revision numbers or hashes from version-control repositories.
https://www.compuphase.com/svnrev.htm
https://autorevision.github.io/

Articles, Books, Specifications

Debugging with GDB; Richard Stallman, Roland H. Pesch & Stan Shebs; Free Software Foundation, 2011; ISBN 978-0-9831592-3-0.
The book is also available in PDF and HTML formats on:
https://www.gnu.org/software/gdb/documentation/

Embedded Debugging with the Black Magic Probe (this book) is available on GitHub in PDF format. The tools mentioned in this book (BMDebug, BMFlash, BMProfile, BMScan, BMSerial, BMTrace, `elf-postlink` and tracegen), live there as well.
https://github.com/compuphase/Black-Magic-Probe-Book

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

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

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

Tcl and the Tk Toolkit; John K. Ousterhout; Addison-Wesley, 1994; ISBN 0-201-63337-X. The draft of this book is freely available on:
http://csis.pace.edu/ benjamin/software/book1.pdf

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 92.
https://diamon.org/ctf/

SEGGER RTT: Resources and documentation for SEGGER's RTT protocol, including portable source code to include in your firmware, viewers, loggers and other tools.
https://wiki.segger.com/RTT

SVD repository: A collection of "System View Description" files for various microcontrollers can be found on GitHub:

https://github.com/posborne/cmsis-svd

Index

! * (asterisk), 28, 113
.bmcfg file, 72, 128
.gdbinit file, **28**, 33, 35, 47, 78, 150
! (exclamation mark), 55
~ (filter character), 86
1BitSquared, 1, 182
¬ (not sign), 86

A Access memory, 34
Access point, 24, 25
adafruit, 182
Adapter board, 26
addr2line utility, 113
Address look-up, 113
Addyi (drug), 2
ADI5, 8
Akeo Consulting, 20
Altering execution flow, 43
Apache Subversion, *see* Subversion
Application Data folder, 150
ARM CoreSight, 74, 79, 87, 117, 147
ARM Cortex, 1, 58, 87
  M0/M0+ architecture, 10, 30, 31, 55, 60, 76, 82, 118, 180
Array (Tcl), 160
ASCII, 97
Assembly code, 48, 64, 67
Assertions, **110**, 111
Asterisk, 28, 113, 151
Asynchronous encoding, 10, 51, 71, 78, 80, 85
Attach target, 28
  ~ troubleshooting, 140, 144
Autocompletion, 64, 65
AutoDesk Fusion, 18
Automatic download, 56, 63
Autorevision utility, **130**, 184

B Babeltrace, 96
backtrace (command), **48**, 113
barectf utility, 93, 102, 109
Base (number), 96
Battery, 18, 24
  ~ connector, 15
  ~ status, 24
Bin file format, 123, 124, 128, 132
  convert ~ to ELF, 123
Binary data (Tcl), 164
Bit fields (register), 67
Bit rate, 78
Bit-banging, 83
BKPT (instruction), 58
Black Sphere Technologies, 1
blackmagic utility, 139, 142
blob, 164
bmcfg file extension, 72, 128
BMDA, *see* Hosted set-up
BMDebug (front-end), 33, 47, 56, 59, **62**, 63–69, 72, 109, 113
BMDU, *see* Unified Connector
BMFlash utility, 8, 63, **124**, 124, 127, 171
BMProfile utility, 119
bmputil (utility), 133, 183
BMScan utility, 20, 22, 25, **27**, 122
bmscript file, 150
BMSerial utility, 176
BMTrace utility, 8, 72, **85**, 86, 93, 109, 146
bool type (TSDL), 97
Boot pin, 30
Bootloader (MCU), 17, **30**, 34, 141
Break on exceptions, 55
Break-out board, 16, 26
Breakpoint, 11, 43, 44, 66, 74, 90
  command list, 90
  conditional ~, 44, 45
  disable ~, 44, 66
  drop-through ~, 90, 91
  enable ~, 44, 66
  hardware ~, 12, 39, 66, 91
  ~ ID, 43
  ignore count, 44
  one-time ~, 43
  software ~, 57, 113
Button, *see* Push-button

C Calculator, 46
Call stack, **48**, 91, 113
Calltree, 120, 121
Case-sensitive (search), 65, 86
Case-sensitive (Tcl), 156, 158, 160
CDC class driver, 20–22, *see also* Serial port
Centisecond, 58
cflow (utility), 120
Channel (tracing), 76, 86, 103
Checksum, 129
  vector table, **38**, 63
Chip ID, *see* Part ID
cksum utility, 129, 171
Clone (debug probe), 76
CMSIS, 60, 67, 76, 113

Code instrumentation, 74, 76, 91
  function entry & exit, 113
Code Read Protection, 35, 125, 141
  LPC family, 37, 125, 131, 174, 175
  SAMD family, 37, 54, 55
  STM32 family, 36, 54, 131
CoderGears, 120
COM port, 28
Command file (GDB), 30, *see also* .gdbinit file
Command line
  arguments (semihosting), 173
  autocompletion, 64, 65
  history, 64
Command list, 90
Commands (GDB), 29
  attach, 28
  backtrace, **48**, 113
  compare-sections, 38, **40**, 122
  connect_rst, 30, **51**, 145
  continue, **41**, 90, 91
  define, 29
  directory, **40**, 64
  disassemble, **48**, 67
  display, **46**, 67
  dprintf, 44, 91
  dump, 123
  erase_range, 38, 41, **53**
  file, 33, **40**
  find, 46, 65
  flash-erase, 37, **41**
  help, **39**, 42, 72
  info, **39**, 72, 113
  load, 34, 35, **40**, 55, 56, 63
  monitor, 13, 28, 36, 45, **49**, 134
  print, 46
  reset, **50**, 56
  restore, 123
  run, 38
  start, 38
  target, 27
  trace, 71
  user-defined, 29, 35
Commands (Tcl), 166
Comments (Tcl), 156, 163
Common Trace Format, 70–72, 85, 86, **92**, 94, 95, 109, 114, 184
  filtering, 70, 104, 106, 108
compare-sections (command), 38, **40**, 122
Conditional breakpoint, 44, 45
Conditional compilation, 75
connect_rst (command), 30, **51**, 145
Connector pin-out, 26, 178
Console (GDB), 13, 32, 64
continue (command), **41**, 90, 91
Control block (RTT), 88
CoreDebug, 60
CoreSight architecture, 74, 79, 87, 117, 147
Cortex Debug header, 14, 16, 18, 19, **26**, 26, 76, 178, 180
Cortex M0/M0+ architecture, 10, 31, 55, 60, 76, 82, 118, 180
CppDepend (utility), 120
CRC mismatch, 63
Crowd Supply, 182
CSV file, 70, 72, 119, 121, 129
CTF, *see* Common Trace Format
  ~ packet, 94
ctxLink (debug probe), 1, 7, **15**, 18, 24, 27, 133, 138, 146
Cygwin, 136

D

Daily build, *see* Development release
Dangling-else problem, 111
Data breakpoint, *see* Watchpoint
Data trace, 74
DDD (front-end), 6, 13
Debug Access Port, 1, 30
Debug connector, *see* Cortex Debug header
Debug Port, 9
Debug probe, 1, 6, 7
  ~ effect, 92, 117
Debug symbols, 33, 34, 40, 113
Debugger attached check, 31, 60
Development release (firmware), 133
Device Manager (Microsoft Windows), 20
DFU
  ~ mode, 21, 22
  ~ protocol, 20, 21, 133, 146
dfu-util, 22, 133, 183
DHCP, 25
DHCSR, 31, 60
dialout group, 23, 139
DiaMon, 92, 94
directory (command), **40**, 64
Disable breakpoint, 44, 66
disassemble (command), **48**, 67, *see also* Assembly code
display (command), **46**, 67
  format, 47
Download to target, 34, 55, 56, 63, 122
dprintf (command), 44, 91
Drop-through breakpoint, 90, 91
DTR (serial port), 28
dump (command), 123
Duplicate strings, 111
DUT (device under test), *see* Target
DWARF, 33, 40, 119, *see also* Debug symbols
DWT, 74, 151

E
Eclipse (front-end), 6, 13
Edit-Compile-Debug Cycle, 55
Elektor, 182
ELF file, 38, 63, 123, 124, 128
elf-postlink utility, 38, 63
Emulating SWO tracing, 82
Enable breakpoint, 44, 66
Enclosure, 18, 182
Endianness, 97, 99
Entry (function), 113
Entry point, 63
Environment variables, 28, 149
  HOME, 148
Erase Flash memory, 37, 41
  failure to ~, 145
erase_range (command), 38, 41, **53**
ESD-protection, 19
Ethernet, 94
ETM, 74
Event (CTF), 100
  ~ header, 94, 100
  ~ id, 100
Exception handling (Tcl), 164
Exceptions, 55
Execution point, 48, 64
  altering ~, 43
Exit (function), 113
Expression evaluator (Tcl), 160

F
file (command), 33, **40**
Filter, 86
find (command), 46, 65
Find text (in source code), 65
Fingerprint (software), 175
Firmware download, *see* Download to target
Firmware update, 20, 21, **133**
Fixed-point numbers, 96
Flash memory, 11, 13, 30
  erase troubleshooting, 145
  ~ programming, 30, **122**, 124, *see also* BMFlash
  ~ remap, 34
flash-erase (command), 37, **41**
Flirc, 182
Floored division (Tcl), 161
Frame (call stack), 48
freeconnect (plug-of-nails), 17, 182
Fresk, Emil, 18, 182
Front-end, 6, 13
  BMDebug, 33, 47, 56, 59, **62**, 63–69, 72
  gdbgui, 32, 56
FTDI MPSSE, 139
Function entry & exit, **113**, 117
Function key, 66, 68

G
Gait, J., 92
Galvanic isolation, 19
GDB
  commands, *see* Commands (GDB)
  ~ console, 13, 32, 64
  versions 11 & 12, 144
gdbgui (front-end), 13, 32, 45, 56, 183
gdbserver, 1, 6–8, 20, 22, 27, 49, 86, 146
git, 130, 135
GitHub, 4, 67, 133, 134, 142, 182–184
  ~ hash, 134
Global variables (Tcl), 159
GnuWin32, 129, 130, 184

H
Halfword, 47
Hammer (Law of the ~), 92
Hard reset, 56
HardFault handler, 60, 61
Hardware breakpoint, 12, 39, 66, 91
  number of ~, 12
Header byte (SWO), 11
heapinfo (semihosting), 54
help (command), **39**, 42, 72
Hex file format, 124, 128
High-impedance mode, 9
History (commands), 64
HOME environment variable, 28, **148**
Hosted set-up, 136, 137, 139, 142
HTTP provisioning, 24

I
IAP (In-Application Programming), 37, 125
IDC header, 26
ident utility, 130, 171
Identifier (format), 103, 106
IEEE-754, 97
ihex format, *see* Hex file
Include files (TSDL), 103
Index cache directory, 148
Inference rule (Make), 107
info (command), **39**, 72, 113
Inline function, 114
Inlined function, 57
Instruction trace, 74
Instrumented profiling, 117
Instrumented trace, 74
Instrumenting code, *see* Code instrumentation
Integer types (TSDL), 97
Interrupt Service Routine, 12
Invariants, 110
Isolation, *see* Galvanic isolation
ISP (In-circuit Serial Programming), 37, 125
ITM, 10, 77, 79, 109, 151

J
J-Link (Segger), 7, 85
Jeff Probe (debug probe), **16**, 52, 53, 89, 134, 180
Jitter, 101
JST PH connector, 15, 18
JTAG, 1, 6, 8, 26, 28
~ header, 16, **26**

K
KDbg (front-end), 6, 13
Keil ULINK-ME, 7

L
Label printer, 172
Latency, 92
Law of the Hammer, 92
LED, 21, 27, 28, 50, 138
Level shifters, 26, 50, 140, 146, 148
Li-Po battery, 16, 18, 24
libopencm3, 60, 67
librdimon, 59
libusbK, 21, 22, 183
License, 5
Line number lookup, 113
Link Register, 112
Linker script, 126
Linux, 22, 142
Linux Foundation, 92
Little Endian, 128
load (command), 34, 35, **40**, 55, 56, 63
Local variables (Tcl), 159
Log file, 129
Low-power mode, *see* Sleep mode
LPC microcontrollers, 30, 34, 37, 38, 63, 81, 124, 125, 131, 174, 175
Flash Memory Remap, 34
UID, 175
LTTng, 93

M
MAC address, 25
Machine code, *see* Assembly code
Maguire, Steve, 110, 184
Make (utility), 107
Manchester encoding, **10**, 10, 71, 78, 80, 85
clock derivation, 11
emulation, 83
Maslow, Abraham, 92
Matloff, Norman, 4, 184
McAvoy, Michael, 18, 182
MCU support scripts, 150
MD5 hash, 175
MEMMAP (register), 34, 150
Memory
~ access, 34
display / set ~, 45, 46
~ watch, 69
Merge strings, 111
Metadata file (CTF), 70, 71, 92, *see also* TSDL
micro-USB, 14
Microsoft Windows, 20, 142
monitor (command), 13, 28, 36, 45, **49**, 134
Morse code, 50
Mouser, 182
msys2, 136

N
Needle probes, 17
Nemiver (front-end), 13
Network scan, 25, 27
newlib C library, 54, 59
Nightly build, *see* Development release
Non-intrusive debugging, 74, 92
NRZ encoding, *see* Asynchronous encoding
NTRACE macro, 106
Number base, 96
NXP, *see* LPC microcontrollers

O
O'Flynn, Colin, 174
objcopy (utility), 123
On-the-go programming, 14
One-time breakpoint, 43
OpenOCD, 7
Optimization (performance), *see* Performance optimization
Optimized code, 42, **57**, 57
Option bytes (STM32), 36, 37, 125, 131
Orbuculum, 78, 84, 183
Ousterhout, John K., 171, 184
Overvoltage protection, *see* Galvanic isolation

P
Packet
~ header (CTF), 94, 95, **99**, 103, 107
~ header (ITM), **11**, 77
~ layout (CTF), 94
Packet-based protocol, 94
Parity bit, 9
Part ID, 151
ParTcl, *see* Tcl scripting
Passive listener, 71, 86
PCBite, 17, 182
Performance optimization, 117
Peripheral register, 9, 47
Permission denied, 139
Phase-Locked Loop (PLL), 68
PicoBlade connector, 14, 15, 148
plugdev group, 23
Pogo-pins, 17, 182
POSIX checksum, 129
Post-mortem analysis, 74
Postconditions, 110
Postprocessing, 131, 172
Potting electronics, 174
Power saving (microcontroller), 90, 117, *see also* Sleep mode
Power selection (ctxLink), 16
Power-cycle, 37, 56, 131
Power-down mode, *see* Sleep mode
Preconditions, 110
Preprocessing, 131, 174
Price, Sid, 18, 136, 182, 183
print (command), 46
printf, 59
printf-style debugging, 12, 110
Probe, *see* Debug probe
~ effect, 92, 117
Production code, 75
Profiling, **117**, 151
Protocol
packet-based ~, 94
remote ~, *see* RSP
stream-based ~, 94
Push-button (on board), **14**, 21, 24

Q
QR code, 172

R
RCS identification string, 129, 130, 171
rdimon.specs, 59
RDP (STM32), 36, 37, 125, 131, 145
Read Protection, *see* Code Read Protection
Real Time Transfer (RTT), 16, 52, **87**, 88, 89, 104, 108, 136, 147, 184
Register
debug ~, 9
peripheral ~, 9, 47, 67
view ~, 49, 67
Remote Serial Protocol, *see* RSP
Renesas RA series, 123
reset (command), **50**, 56
Reset (debug probe), 146
restore (command), 123
Reverse calltree, 120
Ribbon cable, 16
Ring buffer, 87
RS232, 6, 28, 94, *see also* UART
RSP, 6–8, 85, 119, 124, **145**, 146, 172, 175
RTOS, 76, 109
RTT, *see* Real Time Transfer
run (command), 38
Run from GDB, 55
Run-time calltree, 120, 121
Run-time tracing, 12, **74**, 74, 110
RZ encoding, *see* Manchester encoding

S
Salzman, Peter, 4, 184
SAM microcontrollers, 80
SAMD microcontrollers, 37
Sampling (profiling), 117
Scan targets, 51
~ troubleshooting, 143
SciTools, 120
Scope (variables), 48
Script, 150, 153, *see also* Tcl scripting
Scripts (MCU support), *see* MCU support scripts
Section (ELF file), 128
Segger, 136, 147, 148, 184
~ J-Link, 7, 85
~ RTT, 16, 52, 87–89, 108, *see also* Real Time Transfer
Self-destruct code, 37, 125
Semihosting, 54, **57**, 57, 60, 69, 76, 110, 113, 173
redirect to UART, 55
Sensepeek, 17, 182
Serial monitor, *see* Serial terminal
Serial number, 127, 129
Serial port, 23, 86, *see also* UART

Serial terminal, 51, 69, 75, 89, 176
  BMSerial, 176
  RTT, 89
  SWO tracing, 51, **84**
Serial Wire Debug, *see* SWD
Serialization, 127, 129
Severity level, 70, 72, 104, 106, 108
Side-effect, 110
Signature (RTT), 52, 88
Silva, Rafael, 182
Sleep mode, 90, 145, 147, 148
Software breakpoint, 57, 113
Software trace, 74
Source files path, 40, 64
sprintf, 92
SSID (Wi-Fi), 24, 25
ST-Link
  ~ clone, 76
ST-LINK (debug probe), 136
Stable release (firmware), 133
Stack
  ~ frame, 48
  ~ pointer, 61
  ~ trace, 44
start (command), 38
Static calltree, 120, 121
Statistical profiling, 117, *see also* Profiling
stderr, 59
stdint.h, 97
Stepping through code, 41
  by instruction, 49, 68
  skip functions, 42
Stimulus ports, 76, 109
STM32 microcontrollers, 30, 35, 37, 80, 123, 131, 151
  option bytes, 36, 37, 125, 131
  RDP, 36, 125, 131
Stop & Stare, 74
Stream (CTF), 70, 99, 101, 102, 104, 106, 108
Stream-based protocol, 94
String (TSDL), 97
Stub (debugger), 6, *see also* gdbserver
Subversion, 130
sudo, 23, 134
SVC (instruction), 58
SVD file, 67, 184, *see also* System View Description
SVDConv utility, 67
SvnRev utility, **130**, 184
SW-DP protocol, 28, 143, *see also* SWDP scan
SWCLK, 8, 26, 50, 56, 143
SWD, 1, 6, 8, 26, 28, 30, 31, 87, 125
SWDIO, 8, 26, 31, 56
SWDP scan, 30
  ~ troubleshooting, 143
SWO Tracing, 51, 69, **76**, 94, 99, 103, 106, 109
  emulation, 82
  protocol, 11, 78
  troubleshooting ~, 146
Symbolic information, *see* Debug symbols
`sys_write0`, 59
SYSMEMREMAP (register), 34
System View Description, 67, 184, *see also* SVD file

T

tag-connect (plug-of-nails), **17**, 17, 180, 182
Target
  attach ~, 28
  GDB command, 27
  ~ list, 51
  ~ power, 26, 146
  scan ~, 51, 143
Tcl scripting, 131, **153**
Temporary breakpoint, *see* One-time breakpoint
Terminal (program), *see* Serial terminal
Text search, *see* Find text
Thingiverse, 182
Thumb mode (ARM), 68, 112
Time stamp, 86
Tool Command Language, *see* Tcl scripting
Torx (screw head), 3
TPIU, 79
trace (command), 71
Trace capture, 20, 21, 23, 78
Trace context (CTF), 98
Trace Viewer, 84, 92, 93, *see also* BMTrace
tracegen utility, 93, 96, 97, 100, 102, 104, **105**, 107, 109, 115
  include files, 103
TRACESWO, 10, *see also* SWO Tracing
  ~ pin, 10, 26, 76, 146, 180
traceswo (command), 78
Tracing, *see* Run-time tracing
Trailing-zero compression, 11, 71, 78
Transfer speed, 77, 92
Tri-state, 9
Troll debugger, 8, 183
Troubleshooting, 138
  connection, 138
  GDB, 144
  SWO Tracing, 146
  target, 139
  UART, 148

TSDL, 92, 93, **94**, 109, 120, 184, *see also* Common Trace Format
  comments, 98
  include files, 103
  types, 97
TTL-level UART, *see* UART
TUI, 6, 13
  ~ on Microsoft Windows, 6
turbo (front-end), 183
Turnaround, 9
typealias, 96, 97
typedef, 96
Types (TSDL), 97

U
UART, 10, 14, 15, 20, 22, 70, 75, 104, 147, 180
  troubleshooting, 148
udev rules, 23, 24, 134, 147
ULINK-ME (Keil), 7
Understand (utility), 120
Unicode, 128
Unified Connector, 178, 181
Unit testing, 110
Unix Epoch, 58, 167
USB, 94
  ~ cable, 138
USB ID, 22
User procedure (Tcl), 155
User-defined command, 29, 35
UTF-8, 97
UUID, 99

V
Value history, 46
Van Woudenberg, Jasper, 174
Variable, 45, 46
  display format, 66
  ~ watch, 46, 66
Variables (Tcl), 159
Vector table checksum, **38**, 63
vector_catch (command), 45
Version-Control software, 130
vFlashErase packet, 35, 145
VID:PID, 7, 22
VirtualBox, 142
Visual Studio, 13
Visual Studio Code, 13, 183
VisualGDB (front-end), 13
Voltage level, 26, 140, 146
VS Code, *see* Visual Studio Code

W
Watch variable, 46, 66
  register, 49
Watchpoint, 12, 43, 44
Watermark, *see* Fingerprint
Weak linkage, 112
WFI / WFE, 147, *see also* Sleep mode
  ~ with RTT, 90, 147
Wi-Fi link, 7, 24
Wiki, 183
Wildcard character, 65, 86, 151
WinGDB (front-end), 13
WinUSB device, 21, 22
WPS, 24

Y
YAML, 93
Yiu, Joseph, 78, 184

Z
Zadig, 20, 21, 183
Zaitsev, Serge, 153
Zebra label printer, 172
Zero-terminated string, 97