

The Step Command

A Closer Look

Submitted by: Kripa Varma

Contents

1	Introduction	4
2	Background	4
3	Tracing the “step” command	6
4	GDB Internals	7
4.1	The Symbol Side	8
4.2	The Target Side	8
4.3	Tracing “Step” through arm-none-eabi-gdb.....	9
5	pyOCD Internals	11
5.1	Interface.....	11
5.2	Transport.....	11
5.3	Target.....	12
5.4	Flash	12
5.5	Board.....	12
5.6	GDBServer	12
5.7	Tracing “Step” through pyOCD.....	12
6	USB Frame Formats	16
6.1	Token Packet format	16
6.2	Data Packet format	16
6.3	Handshake Packet	16
6.4	PID Field Encoding.....	16
6.5	Device Configuration capture.....	17
6.6	Step Command Capture	18
7	CMSIS-DAP Interface Firmware.....	19
7.1	USB Device Drivers	20
7.2	USB Function Drivers.....	20
7.3	USB Application Layer	20
7.4	Tracing “Step” through CMSIS_DAP Interface firmware	21
8	Serial Wire Debug Protocol	21
9	DP, AP and Debug Registers	23
9.1	SW-DP registers	23

9.1.1	AP SELECT Register	23
9.2	AHB-AP register summary.....	24
9.3	Debug registers	25
9.3.1	Debug Halting Control and Status Register	25
9.3.2	Debug Fault Status Register bit assignments	26
10	Procedure	30
11	References	31

1 Introduction

While debugging a remote target using GDB, what exactly is happening behind the scenes when a “step” command is entered at (gdb) command line? This study is an attempt at uncovering the elaborate dance performed in the backstage among various software components involved in the process, using Ubuntu running arm-none-eabi-gdb and pyOCD at the host side and mbed LPC1768 module at the target side. Each software component that is involved in the path traced by the “step” command is analyzed, and the flow of control within each component is studied.

[Section 2](#) provides some background information about the study setup.

[Section 3](#) gives a high level view of all the interactions taking place between various components during the execution of “step” command.

[Sections 4](#) through [7](#) briefly outlines the architecture of each of the relevant components and the control flow within the component as it carries forward the “step” command.

[Section 8](#) discusses the Serial Wire Debug Protocol (SW-DP) of ARM Debug Interface v5, that is used for accessing the DP and AP registers.

[Section 9](#) discusses the fields of various registers that comprises Debug Access Port of ARM Debug Interface v5.

[Section 10](#), concludes this report with a detailed account of the step-by-step process taken to perform this study.

2 Background

The setup for this study comprises of following resources and tools :

- 1) Mbed board with :
 - Target : LPC1768 (ARM Cortex M3)
 - Interface CPU : LPC1114 (ARM Cortex M0)
 - Interface Firmware : cmsis_dap
- 2) Ubuntu 14.04 Host PC
- 3) Beagle USB 5000 v2 USB Protocol Analyzer
- 4) arm-none-eabi toolchain
- 5) gdb
- 6) pyOCD
- 7) cscope

The figure below shows the setup used for this study. All relevant software components that were studied are shown separately and explicitly.

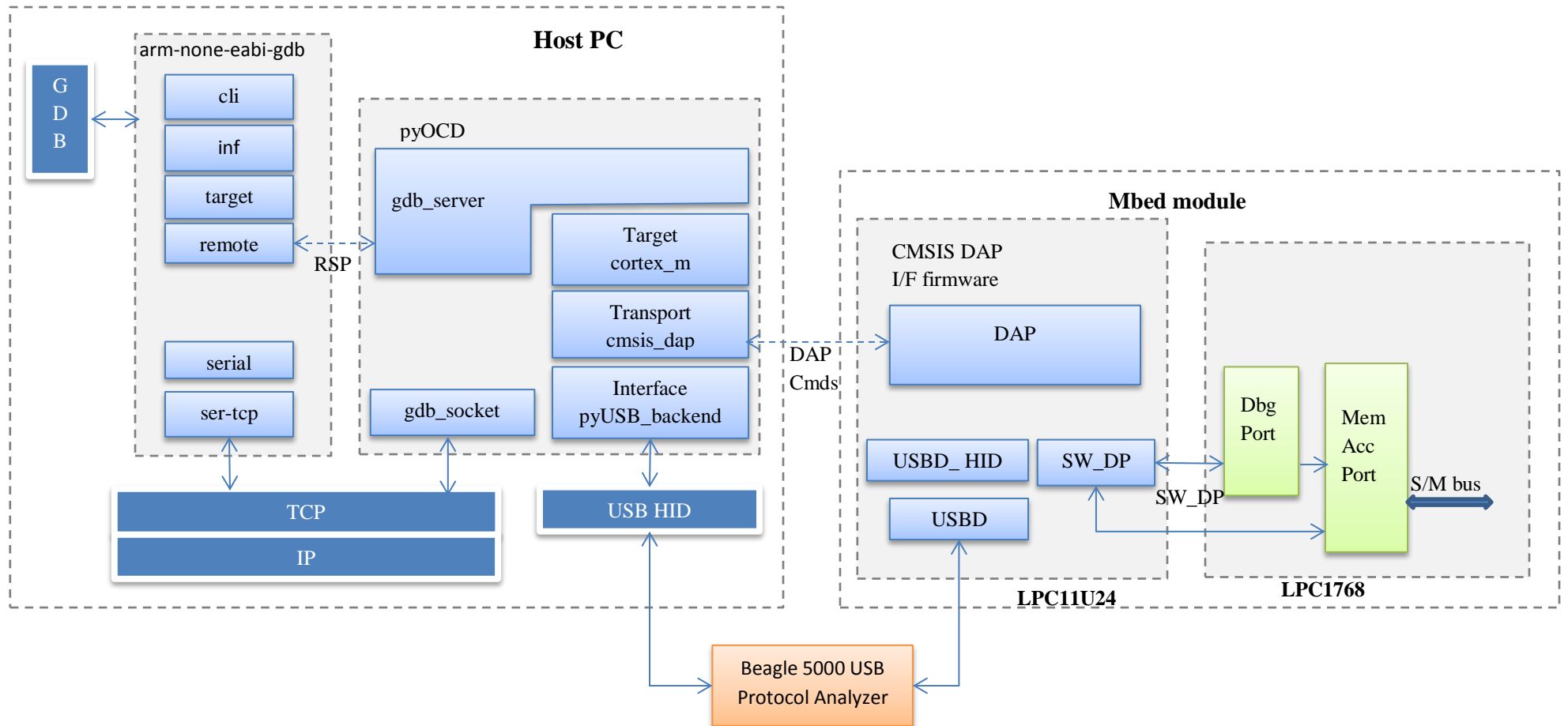


Figure 2-1 : Debugger setup

3 Tracing the “step” command

When target side hits a breakpoint and halts, and the user asks to step to the next line in the program, the target side is asked to execute only a single instruction of the program and then stop it again. This is done by setting the **C_STEP** bit to 1, and then clearing the **C_HALT** bit to 0, in register [DHCSR](#). The core acknowledges completion of the step and re-halt by setting the **S_HALT** bit of the Debug Halting Control and Status Register. The steps involved in achieving this:

- 1) GDB client prepares the RSP packet for the step command in the format :
\$s#cs // where cs is the check sum
- 2) This command is sent over the TCP connection (established using a “target remote” command) to the pyOCD gdb server.
- 3) pyOCD translates it into a request to set **C_STEP** field of **DHCSR** register of ARM Cortex M3 target, along with a series of other register read/writes.
- 4) Each register read/write request is carried to the target over **CMSIS_DAP** transport which translates the register access to DAP Transfer Commands. The generic format of a DAP command is:
CmdId(8), DAPIdx(8), count(8), { { req1 (8), data1 (32) }, { req2 (8), data2 (32) }, ... }
For instance, a single m/y or register write for eg: will be
 - a. **Transfer Command:** write the AP [TAR register](#) (Transfer Address Register) with the address of **DHCSR** (0xE00EDF0)
 - b. **Transfer Command:** write the AP [DRW register](#) (Data Read/Write register) to write the value at **DHCSR**.The DAP Command for setting **C_STEP** of **DHCSR** is:
0x5, 0x0, 0x2, 0x5, 0xE00EDF0, 0xD, 0xA05F000B
- 5) The DAP Commands are then sent over the USB HID Connection to the mbed Interface CPU, as Interrupt Out Transfer.
Sync(8), PID(8) { DAP Transfer Command }, CRC16, EOP
- 6) The command is decoded and translated to Serial Wire Debug Protocol (SW_DP) by the **CMSIS_DAP** interface firmware. The bits are sent over LPC1114's SWD_IO/PIO0_15 connected to LPC1768's TMS/SWDIO/Pin_3.
- 7) The SW_DP requests are decoded by DAP (Debug Access Port) of target LPC1768, then sends an ACK /WAIT/FAULT response and performs the corresponding Memory/Register access on the System Bus. The DAP is an implementation of ARM Debug Interface (ADI) which consists of Debug Port (DP) and Access Port. The mbed module under study implements a Serial Wire Debug Port.
- 8) The SW_DP response is communicated back to the cmsis_dap host, along with result of register/memory access if any, as a HID Interrupt In Transfer.
CmdId(8), count(8), resp_Code(8), [data[32]] // data is present if the DAP Transfer was a Read Tfr.

Upon being informed that the program has stopped, GDB asks for the program counter (PC) register and then compares it with the range of addresses that the symbol side says is associated with the current line. If the PC is outside that range, then GDB leaves the program stopped, figures out the new source line, and reports that to the user. If the PC is still in the range of the current line, then GDB steps by another instruction and checks again, repeating until the PC gets to a different line.

4 GDB Internals

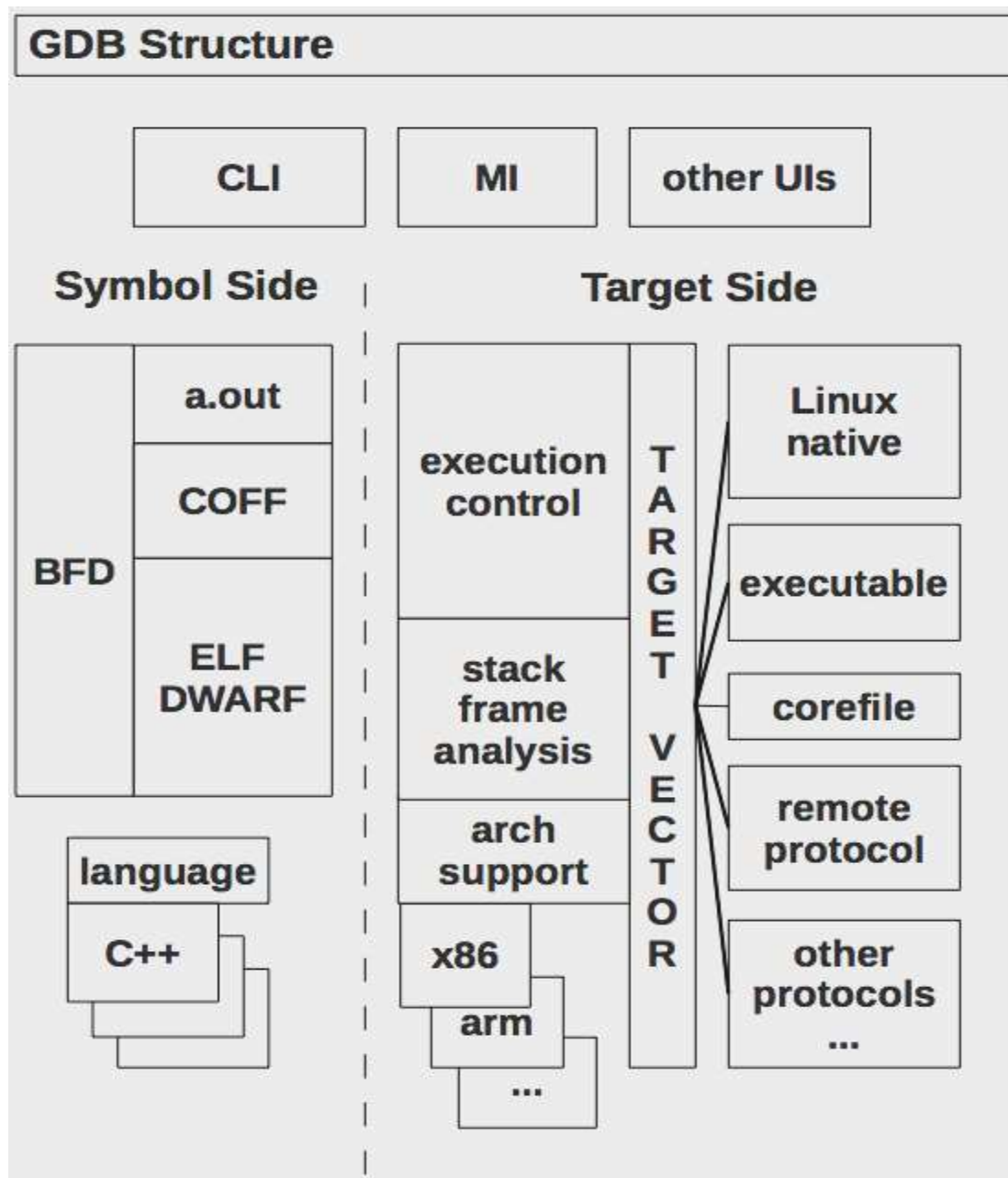


Figure 4-1 GDB Internal blocks

The **Command Line Interface** (`gdb/gdb/cli`) handles the character-by-character interaction with the user using `readline()` library function. The command returned by `readline()` is parsed and compared against command tables. Multiple levels of command tables are maintained. The top level list is `cmdList`. To add commands to first level `add_com()` is called. Separate lists are defined for sub-commands of various top level commands. To add a sub-command, `add_cmd()` is called. Yet another function is `add_prefix_cmd()`.

The **Machine Interface** (`gdb/gdb/mi`) is fundamentally a command-line interface, but is much more elaborate and both commands and results have additional syntax that makes everything explicit. Programs that use `gdb` as part

of their development suite, and provide a debugging GUI, use GDB MI to interface with GDB. For eg: eclipse CDT (C/C++ Development Tooling) plugins.

4.1 The Symbol Side

The symbolic side of gdb can be thought of as “everything you can do in gdb without having a live program running”. For instance, you can look at the types of variables, evaluate expressions, get address of symbols, get symbol at a given address etc.

The symbol side of GDB is mainly responsible for **reading the executable** file, extracting any symbolic information it finds, and building it into a symbol table. Each local variable, each named type, each value of an enum—all of these are separate symbols. The usual symbol file is the file containing the program which GDB is debugging. Symbol files are initially opened by code in ``symfile.c'` using the BFD library.

GDB uses **Binary File Description** (gdb/bfd) library, a part of the binutils package, to read binary files. BFD allows applications to use the same routines to operate on object files whatever the object file format. A new object file format can be supported simply by creating a new BFD back end and adding it to the library.

GDB only uses BFD to read files into its own memory. GDB then has two levels of reader functions of its own. The **first level** is for **basic symbols**, or “minimal symbols”, which are just the names that the linker needs to do its work. These are strings with addresses and not much else; we assume that addresses in text sections are functions, addresses in data sections are data, and so forth.

The second level is **detailed symbolic information**, which typically has its own format different from the basic executable file format. For eg., the DWARF debug format is used for holding sections of an ELF file. The a.out format and the COFF format are examples of other executable formats supported by gdb.

Language-specific information is built into GDB for some languages, allowing you to use expressions and operations in your program's native language, and allowing GDB to output values in a manner consistent with the syntax of your program's native language. For eg: in ANSI C, dereferencing a pointer `p` is accomplished by `*p`. Values can also be represented (and displayed) differently. Hex numbers in C appear as ``0x1ae'`. The language you use to build expressions is called the working language. *set language* command can be used to select a working language manually.

4.2 The Target Side

The target side is all about manipulation of program execution and raw data. In a sense, the target side is a complete low-level debugger. One can single step instructions or dump raw memory, without needing any symbols.

There are three classes of targets: processes, core files, and executable files. GDB can work concurrently on up to three active targets, one in each class.

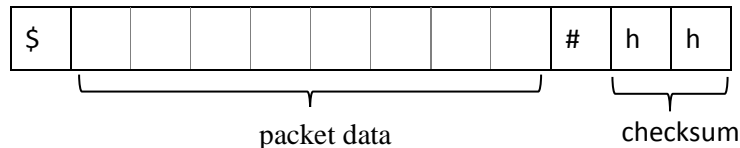
The target architecture object is implemented as the C structure `struct gdbarch *`. The structure, and its methods, are generated using the Bourne shell script ``gdbarch.sh'`. The communication with the target is defined by a set of target operations. These operations are held in a `struct target_ops`. The struct `target_ops` elements are defined and documented in `target.h`. **In the case under study, the step command invokes the target's `to_resume()`.** All variables and return address of a function that is stored to stack is called its **stack frame**. At any moment in a programs execution, the stack consists of a sequence of such frames chained together depending upon current call depth. Contents of stack frame may vary depending on the processor architecture, compiler as well as the OS.

The heart of GDB is its **execution control** loop. The loop is called *wait_for_inferior*, or “wfi” for short, where inferior is the process being debugged. This loop is only entered for commands that cause the program to resume

execution. It waits for control to return from inferior to debugger. When this function actually returns it means the inferior should be left stopped and GDB should read more commands.

GDB implements a protocol called **Remote Serial Protocol** (RSP, *gdb/remote.c*) for connecting to and performing remote debugging through a Remote Stub or GDB Server. The GDB program itself acts as the RSP client and connects with the target acting as the RSP server, listening for a TCP connection. The client issues packets which are requests for information or action. Depending on the nature of the client packet, the server may respond with a packet of its own. **In the case under study the pyOCD/test/gdb_server.py implements the RSP protocol.**

Figure below gives the basic format of a RSP request packet. The response packet always contains a '+' for positive ack or a '-' for a negative ack.



For almost all packets, binary data is represented as two hexadecimal digits per byte of data. The checksum is the unsigned sum of all the characters in the packet data modulo 256. It is represented as a pair of hexadecimal digits.

4.3 Tracing “Step” through arm-none-eabi-gdb

In the case of step command, the RSP packet sent is:

\$s#73

At first the flow of “step” command within gdb, was studied using cscope. Then compiled arm-none-eabi-gdb with debug symbols. Used gdb to attach to arm-none-eabi-gdb while the latter is debugging the remote target. This detailed procedure is explained in [Section 9](#).

The screenshot below shows breakpoint at `net_write_prim()` of *gdb/gdb/ser-tcp.c* where the RSP packet for step command is sent over TCP socket. The contents of `buf` that holds RSP packet is shown.

```

ser-tcp.c
343     'recv' takes 'char *' as second argument, while 'scb->buf' is
344     'unsigned char *'. */
345     return recv (scb->fd, (void *) scb->buf, count, 0);
346 }
347
348 int
349 net_write_prim (struct serial *scb, const void *buf, size_t count)
350 {
> 351     return send (scb->fd, buf, count, 0);
352 }
353
354 int
355 ser_tcp_send_break (struct serial *scb)

multi-thre Thread 0x7f84a In: net_write_prim      Line: 351  PC: 0x47873e
(gdb) x/6cb buf
0x7fff5edb46e0: 36 '$' 115 's' 35 '#' 55 '7' 51 '3' 127 '\177'
(gdb)
  
```

Figure 4-2 GDB RSP Step command view

The figure below shows the path traced by “step” command through the CLI and Target side functions of gdb.

```

Terminal
Inspiron-530: ~/Downloads/pyOCD-master/test
INFO:root:new board id detected: 1010c2a3ff271bf7e9c7412fe86db4043c1c
INFO:root:board allows 10 concurrent packets
INFO:root:DAP SMD MODE initialised
INFO:root:IDCODE: 0x2BA01477
INFO:root:6 hardware breakpoints, 4 literal comparators
INFO:root:CPU core is Cortex-M3
INFO:root:4 hardware watchpoints
INFO:root:GDB server started at port:3333
INFO:root:One client connected!
Inspiron-530: ~/Downloads/pyOCD-master/test$ python gdb_server.py
Welcome to the PyOCD GDB Server Beta Version
INFO:root:new board id detected: 1010c2a3ff271bf7e9c7412fe86db4043c1c
INFO:root:board allows 10 concurrent packets
INFO:root:DAP SMD MODE initialised
INFO:root:IDCODE: 0x2BA01477
INFO:root:6 hardware breakpoints, 4 literal comparators
INFO:root:CPU core is Cortex-M3
INFO:root:4 hardware watchpoints
INFO:root:GDB server started at port:3333
INFO:root:One client connected!
[=====] 100%

Inspiron-530: ~/Downloads/Homework/homework3
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from homework3.elf...done.
(gdb) remote target localhost:3333
Undefined remote command: "target localhost:3333". Try "help remote".
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x0000009a in serial_readable ()
(gdb) load
Loading section .text, size 0x88c8 lma 0x0
Loading section .ARM.exidx, size 0x8 lma 0x88c8
Loading section .data, size 0xb8 lma 0x88d0
Start address 0x9dc, load size 35208
Transfer rate: 219 bytes/sec, 1760 bytes/write.
(gdb) b main
Breakpoint 1 at 0x280: file main.cpp, line 103.
(gdb) c
Continuing.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, main () at main.cpp:103
103 {
(gdb) s
(gdb) c
Continuing.
Program received signal SIGINT, Interrupt.
0x00007fd4089d2110 in __poll_nocancel () at ../sysdeps/unix/syscall-template.S:81
81 ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) b net_write_prin
Breakpoint 2 at 0x47073e: file ser-tcp.c, line 351.
(gdb) c
Continuing.
Breakpoint 2, net_write_prin (scb=0x1b75d30, buf=0x7ffff28153a0, count=7) at ser-tcp.c:351
351 return send (scb->fd, buf, count, 0);
(gdb) bt
#0 net_write_prin (scb=0x1b75d30, buf=0x7ffff28153a0, count=7) at ser-tcp.c:351
#1 0x0000000000476de2 in ser_base_write (scb=0x1b75d30, buf=0x7ffff28153a0, count=7) at ser-base.c:450
#2 0x0000000000489a in serial_write (scb=0x1b75d30, buf=0x7ffff28153a0, count=7) at serial.c:432
#3 0x00000000004854fd in remote_serial_write (str=0x7ffff28153a0 "SHc0db", len=7) at remote.c:7050
#4 0x000000000048584c in putpkt_binary (buf=0x1b7cb80 "Hc0", cnt=3) at remote.c:7181
#5 0x000000000048564f in putpkt (buf=0x1b7cb80 "Hc0") at remote.c:7108
#6 0x000000000047ae56 in set_thread (ptid=..., gen=0) at remote.c:1808
#7 0x000000000047af10 in set_continue_thread (ptid=...) at remote.c:1825
#8 0x0000000000480bb1 in remote_resume (ops=0xb9c8a0 <remote_ops>, ptid=..., step=1, signal=GDB_SIGNAL_0) at remote.c:4756
#9 0x00000000005efa65 in delegate_resume (self=0xb9c8a0 <remote_ops>, arg1=..., arg2=1, arg3=GDB_SIGNAL_0) at target-delegates.c:47
#10 0x00000000005f0b26 in target_resume (ptid=..., step=1, signal=GDB_SIGNAL_0) at target.c:2212
#11 0x00000000005a8c98 in resume (step=1, sig=GDB_SIGNAL_0) at infrun.c:2008
#12 0x00000000005a90c7 in proceed (addr=18446744073709551015, signal=GDB_SIGNAL_DEFAULT, step=1) at infrun.c:2379
#13 0x00000000005a215d in step_once (skip_subroutines=0, single_inst=0, count=1, thread=1) at infcmd.c:1124
#14 0x00000000005a1e24 in step_1 (skip_subroutines=0, single_inst=0, count_string=0x0) at infcmd.c:980
#15 0x00000000005a1b6d in step_command (count_string=0x0, from_tty=1) at infcmd.c:880
#16 0x00000000004aa992 in do_cfunc (c=0x19ab870, args=0x0, from_tty=1) at ./cli/cli-decode.c:187
#17 0x00000000004adb15 in cmd_func (cmd=0x19ab870, args=0x0, from_tty=1) at ./cli/cli-decode.c:1886
#18 0x00000000006d0eff in execute_command (p=0x1957191 "", from_tty=1) at top.c:479
#19 0x00000000005cd630 in command_handler (command=0x1957190 "s") at event-top.c:433
#20 0x00000000005cdbe9 in command_line_handler (rl=0x1b8d7b0 "") at event-top.c:630
#21 0x00000000007343a7 in rl_callback_read_char () at callback.c:220
#22 0x00000000005cd175 in rl_callback_read_char_wrapper (client_data=0x0) at event-top.c:167
#23 0x00000000005cd54b in stdin_event_handler (error=0, client_data=0x0) at event-top.c:373
#24 0x00000000005cc140 in handle_file_event (data=...) at event-loop.c:766
#25 0x00000000005cb627 in process_event () at event-loop.c:343
#26 0x00000000005cb6ee in gdb_do_one_event () at event-loop.c:407
#27 0x00000000005cb71e in start_event_loop () at event-loop.c:432
#28 0x00000000005cd1a7 in cli_command_loop (data=0x0) at event-top.c:182
#29 0x00000000005c3a2c in current_interp_command_loop () at interp.c:328
#30 0x00000000005c45ad in captured_command_loop (data=0x0) at main.c:383
#31 0x00000000005c0800 in catch_errors (func=0x5c4592 <captured_command_loop>, func_args=0x0, errstring=0x869ad5 "", nask=RETURN_MASK_ALL) at exceptions.c:506
#32 0x00000000005c5b3c in captured_main (data=0x7ffff2815dd0) at main.c:1164
#33 0x00000000005c0800 in catch_errors (func=0x5c49aa <captured_main>, func_args=0x7ffff2815dd0, errstring=0x869ad5 "", nask=RETURN_MASK_ALL) at exceptions.c:506
#34 0x00000000005c5b65 in gdb_main (args=0x7ffff2815dd0) at main.c:1172
#35 0x0000000000457c94 in main (argc=2, argv=0x7ffff2815ed0) at gdb.c:33
(gdb)

```

Figure 4-3 GDB step command backtrace

5 pyOCD Internals

pyOCD (python On Chip Debugger) is an Open Source python 2.7 based library for programming and debugging ARM Cortex-M microcontrollers using CMSIS-DAP.

There are 6 main classes that forms the backbone of pyOCD. And the hierarchy(by inclusion and not by inheritance) of these classes can be represented somewhat as in the figure below. Each object has a member variable which is a reference to the object that is in the next lowe level. For eg: *target* class has a *transport* member variable, *transport* class has an *interface* member variable, *flash* class has a *target* member variable etc. The lower layer's methods are accessed through this member variable.

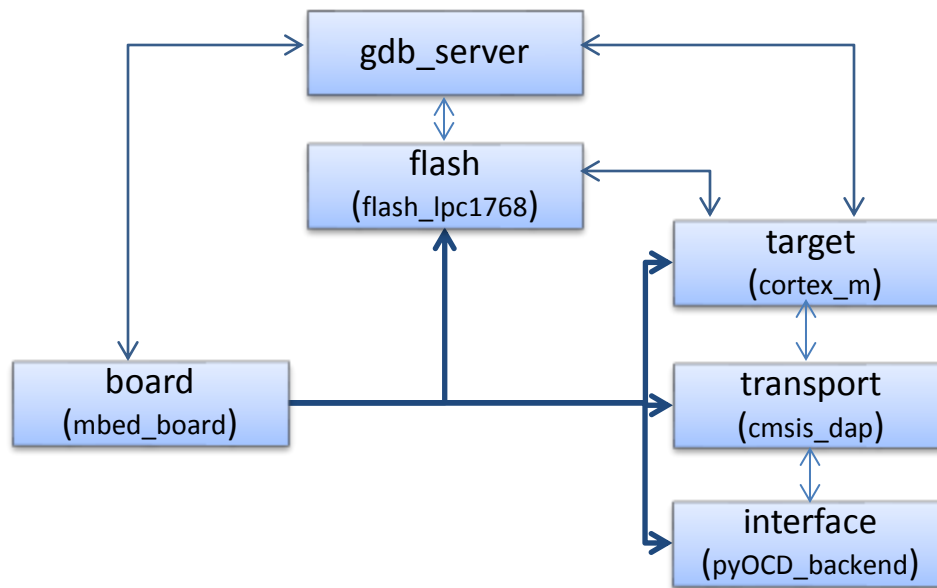


Figure 5-1 pyOCD Internals

5.1 Interface

This class interfaces directly with the pyUSB to talk to USB Stack (libUSB and USB Driver). This module contains basic functionalities to write and read data to and from an interface, and a static method to get all connected boards. In the case of a Linux system pyUSB_backend interface object is instantiated. In the case of Windows, pyWinUSB_backend interface is used.

5.2 Transport

This class provides a mechanism to translate register/memory read write requests to DAP Commands and returns the result of the operation. Transport class object talks directly to the Interface object to send/receive DAP Commands/ Response. The transport used for this study is CMSIS_DAP.

5.3 Target

This class maps the execution control gdb commands (like `step`, `resume`, `halt`, `readMemory`) to respective Register Read/Write operations of the target CPU. It also defines `readMemory/writeMemory` functions, core register mapping, maintains a list of breakpoints and all relevant information about the target CPU. The Target object talks directly to the Transport object. LPC1768 (child of Cortex_m class) is the target class that is instantiated in this setup.

5.4 Flash

This class provides methods for performing flash operations and contains flash algorithm in order to flash a new binary into the target. It interfaces directly with the Target class.

5.5 Board

This class associates a target, a flash, a transport and an interface to create a board. When an object of Mbed_board is instantiated, it creates the respective objects for flash (Flash_lpc1768 class), target (LPC1768 class), transport (CMSIS_DAP) and interface (PyUSB class). The board init function below links the objects together.

```
def __init__(self, target, flash, interface, transport = "cmsis_dap", frequency = 1000000):
    if isinstance(interface, str) == False:
        self.interface = interface
    else:
        self.interface = INTERFACE[interface].chooseInterface(INTERFACE[interface])
    self.transport = TRANSPORT[transport](self.interface)
    self.target = TARGET[target](self.transport)
    self.flash = FLASH[flash](self.target)
    self.target.setFlash(self.flash)
    self.debug_clock_frequency = frequency
    self.closed = False
    return
```

5.6 GDBServer

This class implements a GDB server listening for connection from gdb client on a specific port. It implements the RSP (Remote Serial Protocol). The GDBServer calls methods of FlashBuilder object to program the flash and calls methods of Target object to perform all other target operations.

5.7 Tracing “Step” through pyOCD

The python program **`gdb_server.py`**, calls the Mbed_board’s static method `chooseBoard()` which enumerates the connected boards and initializes the selected mbed_board. Which then instantiates all the other objects, linking them together. Then it starts GDB_Server object listening for TCP connection from the GDB client.

The path followed by “step” command within pyOCD can be represented as below:

```

gdbserver/gdbserver.py:  run()
                        handleMsg()
                        step()
target/cortex_m.py:    step()
                    writeMemory(DHCSR, value)
transport/cmsis_dap.py: writeMem()          // called twice , 1st TAR, 2nd DRW
                        dapTransfer()      // imported from cmsis_dap_core.py
interface/pyUSB_backend.py: write()
                           ep_out.write() // call pyUSB

```

The step() function inside cortex_m.py:

```

def step(self, disable_interrupts = True):
    """ perform an instruction level step. This function preserves the previous
    interrupt mask state
    """
    # Confirm that the target is halted.
    dhcsr = self.readMemory(DHCSR)
    if not (dhcsr & (C_STEP | C_HALT)):
        logging.debug('cannot step: target not halted')
        return
    # clear Debug Cause bits
    self.writeMemory(DFSR, DFSR_DWTTRAP | DFSR_BKPT | DFSR_HALTED)

    # Save previous interrupt mask state
    interrupts_masked = (C_MASKINTS & dhcsr) != 0

    # Mask interrupts - C_HALT must be set when changing to C_MASKINTS
    if not interrupts_masked and disable_interrupts:
        self.writeMemory(DHCSR, DBGKEY | C_DEBUGEN | C_HALT | C_MASKINTS)

    # Single step using current C_MASKINTS setting
    if disable_interrupts or interrupts_masked:
        self.writeMemory(DHCSR, DBGKEY | C_DEBUGEN | C_MASKINTS | C_STEP)
    else:
        self.writeMemory(DHCSR, DBGKEY | C_DEBUGEN | C_STEP)

    # Wait for halt to auto set (This should be done before the first read)
    while not self.readMemory(DHCSR) & C_HALT:
        pass

    # Restore interrupt mask state
    if not interrupts_masked and disable_interrupts:
        # Unmask interrupts - C_HALT must be set when changing to C_MASKINTS
        self.writeMemory(DHCSR, DBGKEY | C_DEBUGEN | C_HALT )

    return

```

The screenshot below shows the DAP transfers involved in the execution of “step” command, printed out from pyOCD/target/cortex_m.py and pyOCD/transport/cmsis_dap_core.py. Each transfer takes two register operations: Write to TAR and Read/Write of DRW. They are registers of DAP Memory Access port explained in Section 8. The format of each DAP Transfer Packet is as follows:

CmdId(8), DAPIdx(8), count(8), { { req1 (8), data1 (32) }, { req2 (8), data2 (32) }, }

CmdId : indicates type of DAP Command that follows and value 0x05 indicates DAP Transfer.

Count : the number of Access Requests that follows.

req : RegAddr[3:2], RnW(1), APnDP(1)

Reg Addr values:

```
AP_REG = {'CSW': 0x00,
          'TAR': 0x04,
          'DRW': 0x0C}
```

The DAP Command for setting C_STEP of DHCSR thus is:

0x5, 0x0, 0x2, { 0x5, 0xE000EDF0, 0xD, 0xA05F000B }


```
Terminal
Inspiron-530: ~/Downloads/pyOCD-master

Ptarget Step starts <<
INFO:root:
1. Read DHCSR
INFO:root: dapTransfer req count = 2
INFO:root: req[0] = 0x5
INFO:root: data[0] = 0xE000EDF0
INFO:root: req[1] = 0xF
05:00:02:05:f0:ed:00:e0:0f
INFO:root:
2. Write DFSR
INFO:root: dapTransfer req count = 2
INFO:root: req[0] = 0x5
INFO:root: data[0] = 0xE000ED30
INFO:root: req[1] = 0xD
INFO:root: data[1] = 0x7
05:00:02:05:30:ed:00:e0:0d:07:00:00:00
INFO:root:
3. Write DHCSR, set C_HALT & C_MASKINTS
INFO:root: dapTransfer req count = 2
INFO:root: req[0] = 0x5
INFO:root: data[0] = 0xE000EDF0
INFO:root: req[1] = 0xD
INFO:root: data[1] = 0xA05F000B
05:00:02:05:f0:ed:00:e0:0d:0b:00:5f:a0
INFO:root:
4. Write DHCSR, set C_MASKINTS & set C_STEP
INFO:root: dapTransfer req count = 2
INFO:root: req[0] = 0x5
INFO:root: data[0] = 0xE000EDF0
INFO:root: req[1] = 0xD
INFO:root: data[1] = 0xA05F000D
05:00:02:05:f0:ed:00:e0:0d:0d:00:5f:a0
INFO:root:
5. Read DHCSR, check if C_HALT set
INFO:root: dapTransfer req count = 2
INFO:root: req[0] = 0x5
INFO:root: data[0] = 0xE000EDF0
INFO:root: req[1] = 0xF
05:00:02:05:f0:ed:00:e0:0f
INFO:root:
6. Write DHCSR, clr C_MASKINTS
INFO:root: dapTransfer req count = 2
INFO:root: req[0] = 0x5
INFO:root: data[0] = 0xE000EDF0
INFO:root: req[1] = 0xD
INFO:root: data[1] = 0xA05F0003
05:00:02:05:f0:ed:00:e0:0d:03:00:5f:a0
INFO:root:
>> target Step done
```

Figure 5-2 pyOCD DAP Commands

6 USB Frame Formats

6.1 Token Packet format

Sync	PID	ADDR	ENDP	CRC5	EOP
------	-----	------	------	------	-----

6.2 Data Packet format

Sync	PID	Data	CRC16	EOP
------	-----	------	-------	-----

6.3 Handshake Packet

Sync	PID	EOP
------	-----	-----

6.4 PID Field Encoding

Group	PID Value	Packet Identifier
Token	0001	OUT Token
	1001	IN Token
	0101	SOF Token
	1101	SETUP Token
Data	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
Handshake	0010	ACK Handshake
	1010	NAK Handshake
	1110	STALL Handshake
	0110	NYET (No Response Yet)
Special	1100	PREamble
	1100	ERR
	1000	Split
	0100	Ping

PID Field Encoding as an 8 bit field

PID ₀	PID ₁	PID ₂	PID ₃	nPID ₀	nPID ₁	nPID ₂	nPID ₃
------------------	------------------	------------------	------------------	-------------------	-------------------	-------------------	-------------------

6.5 Device Configuration capture

mbd_connect_to_pc - Total Phase Data Center

File Edit Analyzer View Help

436.5 KB

Sp	Index	ms.ms.us	Len	Err	Dev	Ep	Record	Summary
FS	81	0:06.351.276	0 B		00	00	SetAddress	Address=02
FS	90	0:06.377.275	18 B		02	00	Get Device Descr...	Index=0 Length=18
FS	103	0:06.378.009	122 B		02	00	Get Configuration...	Index=0 Length=255
FS	104	0:06.378.009	8 B		02	00	SETUP txn	80 06 00 02 00 00 FF 00
FS	108	0:06.378.123	64 B		02	00	IN txn	09 02 7A 00 04 01 00 80 FA 09 04 00 00 02 08 06 50 07 07 05 82 02 40 00...
FS	112	0:06.378.252	58 B		02	00	IN txn	10 00 02 09 04 02 00 02 0A 00 00 00 07 05 04 02 40 00 00 07 05 84 02 40...
FS	116	0:06.378.307	0 B		02	00	OUT txn	
FS	120	0:06.379.103	74 B		02	00	Get String Descr...	Index=3 Length=255
FS	121	0:06.379.103	8 B		02	00	SETUP txn	80 06 03 03 09 04 FF 00
FS	125	0:06.379.215	64 B		02	00	IN txn	4A 03 31 00 30 00 31 00 30 00 39 00 64 00 66 00 64 00 36 00 39 00 35 00...
FS	129	0:06.379.310	10 B		02	00	IN txn	31 00 31 00 31 00 66 00 63 00
FS	133	0:06.379.331	0 B		02	00	OUT txn	
FS	137	0:06.379.626	4 B		02	00	Get String Descr...	Index=0 Length=255
FS	150	0:06.380.066	30 B		02	00	Get String Descr...	Index=2 Length=255
FS	163	0:06.407.457	18 B		02	00	Get Device Descr...	Index=0 Length=18
FS	176	0:06.407.717	9 B		02	00	Get Configuration...	Index=0 Length=9
FS	189	0:06.407.999	122 B		02	00	Get Configuration...	Index=0 Length=122
FS	206	0:06.408.447	9 B		02	00	Get Configuration...	Index=0 Length=9
FS	219	0:06.408.700	122 B		02	00	Get Configuration...	Index=0 Length=122
FS	236	0:06.409.199	0 B		02	00	Set Configuration	Configuration=1
FS	245	0:06.409.797	18 B		02	00	Get Device Descr...	Index=0 Length=18
FS	258	0:06.410.251	16 B		02	00	Get String Descr...	Index=7 Length=254
FS	271	0:06.410.626	30 B		02	00	Get String Descr...	Index=6 Length=254
FS	284	0:06.441.335	18 B		02	00	Get Device Descr...	Index=0 Length=18
FS	297	0:06.441.585	2 B		02	00	Get String Descr...	Index=0 Length=2
FS	310	0:06.441.821	4 B		02	00	Get String Descr...	Index=0 Length=4
FS	323	0:06.442.071	2 B		02	00	Get String Descr...	Index=3 Length=2
FS	336	0:06.442.324	74 B		02	00	Get String Descr...	Index=3 Length=74
FS	353	0:06.480.518	1 B		02	00	Get Max LUN	Max LUN = 0
FS	366	0:06.480.750	36 B		02	02	Inquiry [0]	Passed

Text LiveSearch

No filter: 4999 records.

Ready Disconnected

Protocol Lens: USB

EN

Figure 6-1 USB Protocol Analyzer capture - Mbed USB Device Configuration

6.6 Step Command Capture

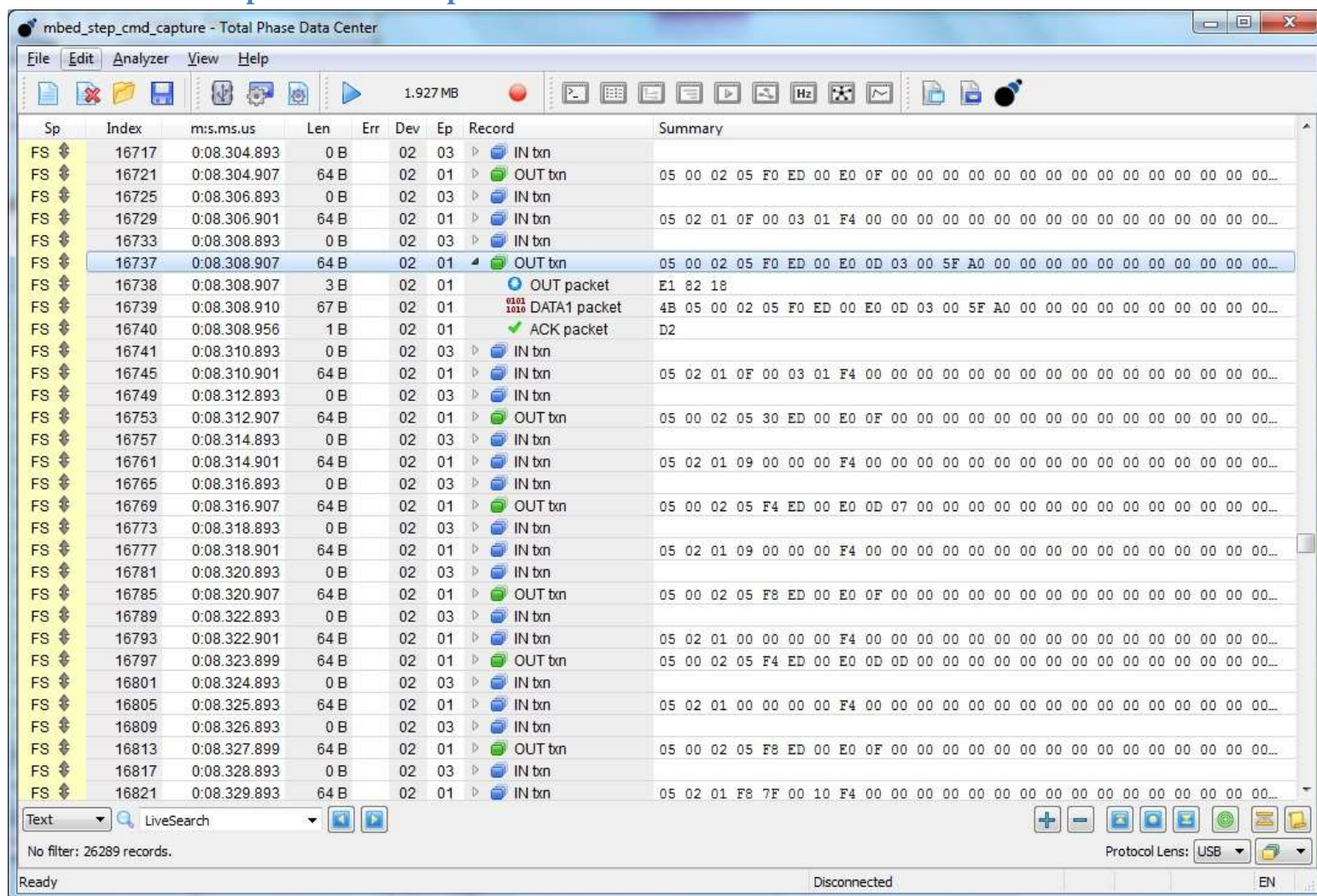


Figure 6-6 USB Protocol Analyzer Capture – Step command exchange

7 CMSIS-DAP Interface Firmware

CMSIS-DAP provides a standardized way to access the Coresight Debug Access Port (DAP) of an ARM Cortex microcontroller via USB HID Connection. CMSIS-DAP is generally implemented as an on-board interface chip, providing direct USB connection from a development board to a debugger running on a host computer on one side, and over JTAG (Joint Test Action Group) or SWD (Serial Wire Debug) to the target device to access the Coresight DAP on the other. This study setup uses SWD Port for debugging the target LPC1768.

The figure below is a high level functional block diagram of CMSIS_DAP Interface Firmware, created based on the release “K20DX128 Bootloader Update” of the source code.

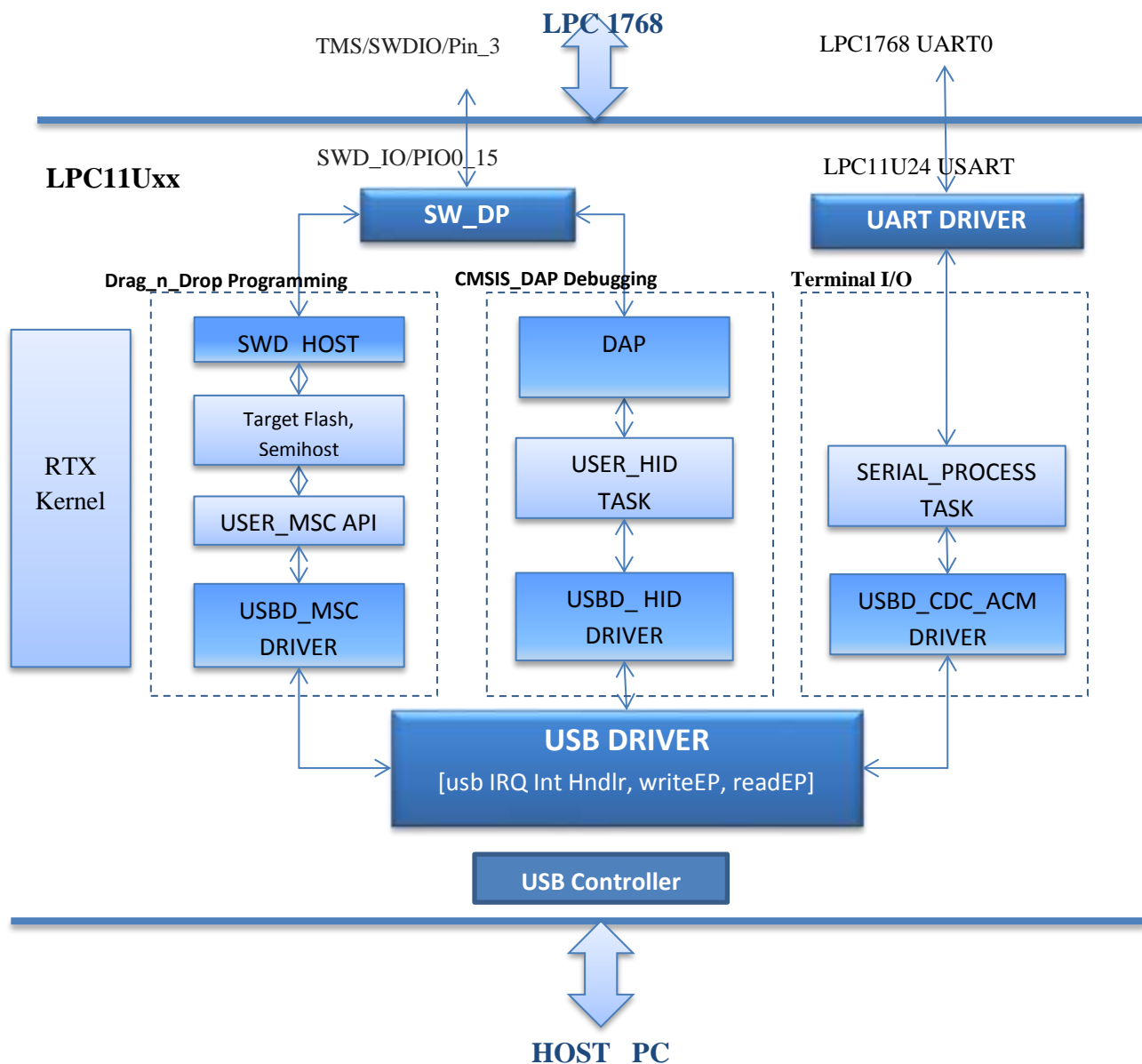


Figure 7-1 CMSIS-DAP Interface Firmware

7.1 USB Device Drivers

USBD_LPC11Uxx.c, implements USB Device Controller Driver. It provides hardware abstraction APIs, WriteEP() and ReadEP() to write to and read from a given EndPoint. It also implements the USB_IRQHandler(). It dispatches EndPoint Interrupts to the respective interface drivers' task. The function pointers for these tasks are held in an array (usb_lib.c: const USBD_RTX_P_EP[16]), each element of which corresponds to one end point. Depending on the configuration, EndPoint N may be a INT/BULK , IN/OUT EndPoint of HID/MSC/CDC_ACM. This correspondence is stored as an array of function pointers. The IRQHandler thus sets an event for Nth task in the array, upon detecting an EndPoint N interrupt.

USB_LIB.C , implements a task that dispatches SOF events to respective interface driver. It also creates the Function Device Drivers' EndPoint Event tasks, and declares the array of function pointers to these tasks.

USB_CORE.C & USB_CORE_*.C implement the Control Endpoint (Endpoint #0) transactions.

7.2 USB Function Drivers

There are MSC (Mass Storage Device) ,HID(Human Interface Device), and CDC (Communication Device Class) drivers. MSC driver's purpose is to facilitate for Drag_n_Drop Programming of the flash, while HID driver allows the Host's CMSIS_DAP Debugger to use SerialWire or SerialWireJTAG debugging and CDC driver facilitates terminal I/O by implementing the virtual COM port interface.

Each of these three drivers contain intermediate buffer for BULK/INT IN and OUT transactions, depending on which transactions they support.

USBD_MSC.C implements a task to handle BULKIN and BULKOUT endpoint events. Endpoint #2 is the MSC_BULK_IN/OUT Endpoint. It has only one buffer to store the data: USBD_MSC_BulkOut

USBD_HID .C implements a task to handle INTIN and INTOUT endpoint events. Endpoint #1 is the HID_INT_IN/OUT Endpoint. It has 3 associated buffers USBD_HID_InReport, USBD_HID_OutReport and USBD_HID_FeatReport.

USBD_CDC_ACM.C implements a task to handle INTIN (keep alive's on Endpoint #3) and another one to handle BULK_IN/OUT events on Endpoint #4. It has two associated buffers: USBD_CDC_ACM_SendBuf, USBD_CDC_ACM_ReceiveBuf, USBD_CDC_ACM_NotifyBuf.

When an OUT event is set to a driver from USBD_IRQHandler, the driver task reads data using USB Device Controller Driver's ReadEP() and stores to OUT/Receive Buffer and then notifies respective Upper Layer using callback functions.

When an IN event is set to a driver from USBD_IRQHandler, the driver task reads from the IN/Send Buffer and sends data to Host using WriteEP().

7.3 USB Application Layer

For each of the function driver, there is a corresponding USB Application.

USBD_USER_MSC.C : Defines callback functions that the MSC Driver's task directly calls to perform target flash operations. The target_flash.c in turn interfaces with swd_host.c functions to access DAP functions, which performs the specified operation via SerialWire Debug Access Port.

USBD_USER_HID.C : Implements the hid_process() task which is the task that performs CMSIS_DAP operations by invoking routines in DAP.c . DAP.C Implements functions that processes DAP commands and prepares DAP responses.

The HID Driver's task invokes usbd_hid_set_report() callback of usbd_user_hid.c upon an INTOUT event, that sets the DAP_PAQUET_RECIEVED event and wakes up the hid_process() task.

MAIN.C: Implements serial_process() task which loops infinitely reading from UART_DRIVER and writing to USBD_CDC_ACM and vice versa.

SW_DP.c: Implements SWD_Transfer() function that performs Serial Wire Debug Protocol transactions over the Serial Wire (that is LPC11U24's SWD_IO/PIO0_15 connected to LPC1768's TMS/SWDIO/Pin_3) .

UART.C: Implements the UART Driver – UART_IRQHandler, and read and write functions. USART of LPC11U24 is wired to UART0 of LPC1768.

7.4 Tracing “Step” through CMSIS_DAP Interface firmware

The path traced by “step” command within CMSIS_DAP Interface firmware:

```

USBD_LPC11Uxx.c : USB_IRQHandler()
                  isr_evt_set(USBD_EVT_OUT , USBD_RTX_EPTask[2/2] )
USBD_HID.c       : __task USBD_RTX_HID_EP_INT_Event()           // wakes up
                  USBD_HID_EP_INT_Event()
                  USBD_HID_EP_INTOUT_Event()

USB_USER_HID.c   : usbd_hid_set_report()                        // call back invoked
                  memcpy()                                       // copy to intermediate buffer
                  os_evt_set(DAP_PAQUET_RECEIVED, dapTask);
                  __task hid_process()                            // wakes up
                  usbd_hid_process()

DAP.c            : DAP_ProcessCommand(request, response)
                  DAP_SWD_Transfer(request, response)

SW_DP.C          : SWD_Transfer(request, response)              // called once for each request in DAP Cmd
                  SW_WRITE_BIT()                                // called once per bit
                  PIN_SWDIO_OUT()
                  PIN_DELAY()

```

8 Serial Wire Debug Protocol

The fields involved in Serial Wire Debug Protocol transactions between host (LPC11UXX) and target(LPC1768) are given below:

Start : A single start bit, with value 1.

APnDP : A single bit, indicating whether the Debug Port or the Access Port Access Register is to be accessed. This bit is 0 for an DPACC access, or 1 for a APACC access.

RnW : A single bit, indicating whether the access is a read or a write. This bit is 0 for an write access, or 1 for a read access.

A[2:3] : Two bits, giving the A[3:2] address field for the DP or AP Register Address. Bits [1:0] of the address are always b00.

- For a DPACC access, the A[3:2] value is the address of the register in the SW-DP
- For an APACC access, the register being addressed depends on the A[3:2] value and the value held in the SELECT register.

The A[3:2] value is transmitted Least Significant Bit (LSB) first on the wire. This is why it appears as A[2:3] on the diagrams.

Parity : A single parity bit for the preceding packet. Even parity is used, that is:

- the number of bits set to 1 is odd, then the parity bit is set to 1
- the number of bits set to 1 is even, then the parity bit is set to 0.

Packet requests

The parity check is made over the APnDP, RnW and A[2:3] bits.

Data transfers (WDATA and RDATA)

The parity check is made over the 32 data bits, WDATA[0:31] or RDATA[0:31].

Stop : A single stop bit. In the synchronous SWD protocol this is always 0.

Park : A single bit. The host does not drive the line for this bit, and the line is pulled HIGH by the SWD interface hardware. The target reads this bit as 1.

Trn : Turnaround. This is a period during which the line is not driven and the state of the line is undefined. The length of the turnaround period is controlled by the TURNROUND field in the Wire Control Register. The default setting is a turnaround period of one clock cycle.

ACK[0:2] : A three-bit target-to-host response. The ACK value is transmitted LSB-first on the wire. This is why it appears as ACK[0:2] on the diagram.

WDATA[0:31] : 32 bits of write data, from host to target. The WDATA value is transmitted LSB-first on the wire.

RDATA[0:31] : 32 bits of read data, from target to host. The RDATA value is transmitted LSB-first on the wire. The figure below shows a successful SWD Write of DHCSR Addr to TAR register:

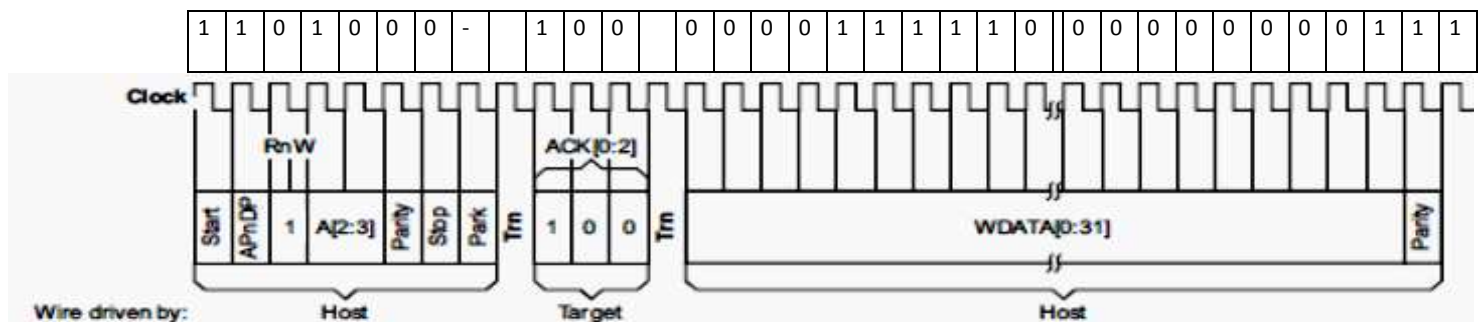


Figure 8-1 Serial Wire Bits Exchange for DHCSR Addr write to TAR Register (from ARM.com ARM Debug Interface v5 Specification)

9 DP, AP and Debug Registers

The DAP (Debug Access Port) is split into two main control units, the Debug Port (DP) and the Access Port (AP), and the physical connection to the debugger is part of the DP. The DAP supports two types of access, Debug Port (DP) accesses and Access Port (AP) accesses. All accesses are 32-bits.

One of the four registers within the DP is the AP Select Register, SELECT. This register specifies a particular Access Port, and a bank of four 32-bit words within the register map of that AP. It enables up to 256 Access Ports to be implemented, and gives access to any one of 16 four-word banks of registers on the selected AP.

In any AP access transaction from the debugger, the two address bits A[3:2] are decoded to select one of the four 32-bit words from the register bank indicated by the SELECT Register. In other words, they select a specific register within the selected four-register bank.

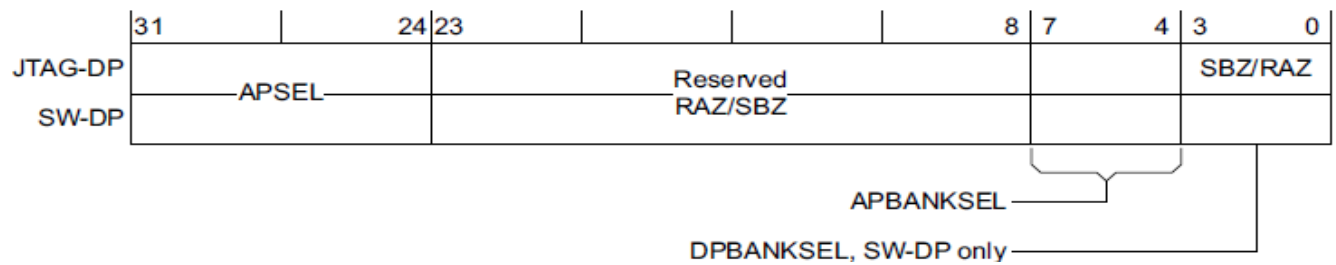
The Access Port used for CMSIS_DAP SWD Debugging is the AHP AP AP, which is a Memory Access Port. It is selected by writing 0 to SELECT.APSEL field.

9.1 SW-DP registers

Addr[3:2]	Name	Description	JTAG-DP	SW-DP
b00	ABORT	AP Abort Register	Yes	Yes
b01	IDCODE	ID Code Register	Yes	Yes
b02	CTRL/STAT	DP Control/Status Register	Yes	Yes
b03	SELECT	Select Register	Yes	Yes
b04	RDBUFF	Read Buffer	Yes	Yes
b05	WCR	Wire Control Register	No	Yes
b06	TARGETID	Target Identification Register	No	Yes
b07	DLPIDR	Data Link Protocol Identification Register	No	Yes
b08	RESEND	Read Resend Register	No	Yes

9.1.1 AP SELECT Register

The AP Select Register is always present on all DP implementations. Its main purpose is to select the current *Access Port* (AP) and the active four-word register bank within that AP. On a SW-DP, it also selects the Debug Port address bank. It is at address 0x8 on write operations when the APnDP bit = 0, and is a write-only register. The figure below shows the Select register format (from ARM.com ARM Debug Interface v5 Specification).



AP Select Register bit assignments

Bits	Function	R/W	Description
[31:24]	APSEL	W	Selects the current access port. 0x00 - AHB-AP 0x01 - APB-AP 0x02 - JTAG-AP 0x03 - Cortex-M3 if present.
[23:8]	-	W	Reserved. SBZ/RAZ.
[7:4]	APBANKSEL	W	Selects the active four-word register window on the current access port.
[3:0]	DPBANKSEL	R/W	Selects the register that appears at DP register 0x4: 0x0 - CTRL/STAT, read/write 0x1 - DLCR, read/write 0x2 - TARGETID, read-only 0x3 - DLPIDR, read-only. All other values are reserved.

9.2 AHB-AP register summary

Offset	Name	Type	Reset	APBANKSEL	A[3:2]	Description
0x00	CSW	RW	-	0x0	b00	Control and Status Word Register
0x04	TAR	RW	-	0x0	b01	Transfer Address Register
0x0C	DRW	RW	-	0x0	b11	Data Read/Write Register
0x10	BD0	RW	-	0x1	b00	Banked Data Register0
0x14	BD1	RW	-	0x1	b01	Banked Data Register1
0x18	BD2	RW	-	0x1	b10	Banked Data Register2
0x1C	BD3	RW	-	0x1	b11	Banked Data Register3
0xF8	DBGDRAR	RO	0xE00FF003	0xF	b10	ROM Address Register

Offset	Name	Type	Reset	APBANKSEL	A[3:2]	Description
0xFC	IDR	RO	0x24770011	0xF	b11	Identification Register

9.3 Debug registers

The table below lists the debug registers of Cortex M3.

Address	Name	Type	Reset	Description
0xE000ED30	DFSR	RW	0x00000000	Debug Fault Status Register Power-on reset only.
0xE000EDF0	DHCSR	RW	0x00000000	Debug Halting Control and Status Register
0xE000EDF4	DCRSR	WO	-	Debug Core Register Selector Register
0xE000EDF8	DCRDR	RW	-	Debug Core Register Data Register
0xE000EDFC	DEMCR	RW	0x00000000	Debug Exception and Monitor Control Register

9.3.1 Debug Halting Control and Status Register

Bits	Type	Field	Function
[31:16]	W	DBGKEY	Debug Key. 0xA05F must be written whenever this register is written. Reads back as status bits [25:16]. If not written as Key, the write operation is ignored and no bits are written into the register.
[31:26]	-	-	Reserved, RAZ.
[25]	R	S_RESET_ST	Indicates that the core has been reset, or is now being reset, since the last time this bit was read. This is a sticky bit that clears on read. So, reading twice and getting 1 then 0 means it was reset in the past. Reading twice and getting 1 both times means that it is being reset now (held in reset still).
[24]	R	S_RETIRE_ST	Indicates that an instruction has completed since last read. This is a sticky bit that clears on read. This determines if the core is stalled on a load/store or fetch.
[23:20]	-	-	Reserved, RAZ.
[19]	R	S_LOCKUP	Reads as one if the core is running (not halted) and a lockup condition is present.
[18]	R	S_SLEEP	Indicates that the core is sleeping (WFI, WFE or SLEEP-ON-

Bits	Type	Field	Function
			EXIT). Must use C_HALT to gain control or wait for interrupt to wake-up. For more information on SLEEP-ON-EXIT see Table 7.1 .
[17]	R	S_HALT	The core is in debug state when S_HALT is set.
[16]	R	S_REGRDY	Register Read/Write on the Debug Core Register Selector register is available. Last transfer is complete.
[15:6]	-	-	Reserved.
[5]	R/W	C_SNAPSTALL	<p>If the core is stalled on a load/store operation the stall ceases and the instruction is forced to complete. This enables Halting debug to gain control of the core. It can only be set if:</p> <p>C_DEBUGEN = 1 C_HALT = 1</p> <p>The core reads S_RETIRE_ST as 0. This indicates that no instruction has advanced. This prevents misuse. The bus state is Unpredictable when this is used. S_RETIRE can detect core stalls on load/store operations.</p>
[4]	-	-	Reserved.
[3]	R/W	C_MASKINTS	Mask interrupts when stepping or running in halted debug. Does not affect NMI, which is not maskable. Must only be modified when the processor is halted (S_HALT=1).
[2]	R/W	C_STEP	Steps the core in halted debug. When C_DEBUGEN = 0, this bit has no effect. Must only be modified when the processor is halted (S_HALT=1).
[1]	R/W	C_HALT	Halts the core. This bit is set automatically when the core Halts. For example Breakpoint. This bit clears on core reset. This bit can only be written if C_DEBUGEN is 1, otherwise it is ignored. When setting this bit to 1, C_DEBUGEN must also be written to 1 in the same value (value[1:0] is 2'b11). The core can halt itself, but only if C_DEBUGEN is already 1 and only if it writes with b11).
[0]	R/W	C_DEBUGEN	Enables debug. This can only be written by AHB-AP and not by the core. It is ignored when written by the core, which cannot set or clear it. The core must write a 1 to it when writing C_HALT to halt itself.

9.3.2 Debug Fault Status Register bit assignments

Bits	Field	Function
[31:5]	-	Reserved.
[4]	EXTERNAL	External debug request flag:

Bits	Field	Function
		1 = EDBGRQ has halted the core 0 = no EDBGRQ external debug request occurred. The processor stops on next instruction boundary.
[3]	VCATCH	Vector catch flag: 1 = vector catch occurred 0 = no vector catch occurred. When the VCATCH flag is set, a flag in the Debug Exception and Monitor Control Register is also set to indicate the type of vector catch.
[2]	DWTRAP	<i>Data Watchpoint (DW) flag:</i> 1 = DW match 0 = no DW match. The processor stops at the current instruction or at the next instruction.
[1]	BKPT	BKPT flag: 1 = BKPT instruction or hardware breakpoint match 0 = no BKPT instruction or hardware breakpoint match. The BKPT flag is set by the execution of the BKPT instruction or on an instruction whose address triggered the breakpoint comparator match. When the processor has halted, the return PC points to the address of the breakpointed instruction.
[0]	HALTED	Halt request flag: 1 = halt requested by DAP access to C_HALT or halted with C_STEP asserted 0 = no halt request.

9.3.3 Debug Core Register Selector Register

Bits	Type	Field	Function
[31:17]	-	-	Reserved
[16]	Write	REGWnR	Write = 1 Read = 0
[15:5]	-	-	-
[4:0]	Write	REGSEL	5b00000 = R0 5b00001 = R1 ... 5b01111 = DebugReturnAddress() 5b10000 = xPSR/Flags, Execution Number, and state information 5b10001 = <i>MSP</i> (Main SP) 5b10010 = <i>PSP</i> (Process SP) 5b10011 = RAZ/WI All unused values reserved

This write-only register generates a handshake to the core to transfer data to or from Debug Core Register Data Register and the selected register. Until this core transaction is complete, bit [16], **S_REGRDY**, of the DHCSR is 0.

The figure below shows ARM Debug Interface, with various Access Port options.

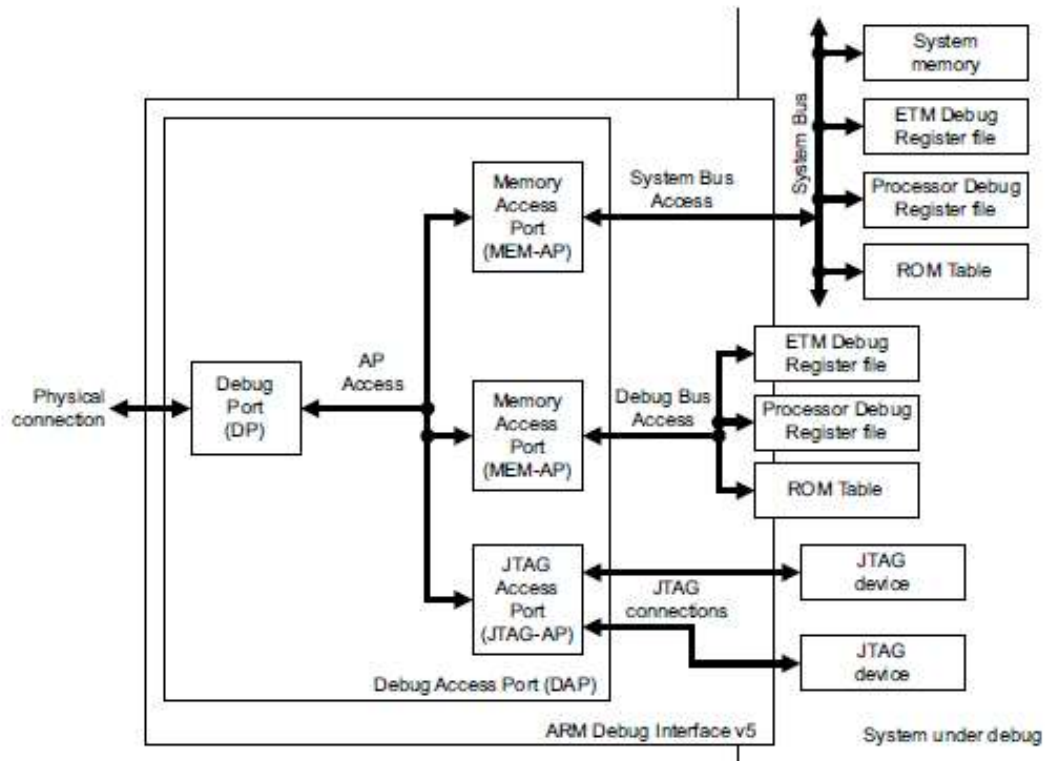


Figure 9-1 ARM Debug Interface – Access Ports (from ARM.com ARM Debug Interface v5 Specification)

The figure below shows how DP connects to Debug Components through Mem-AP.

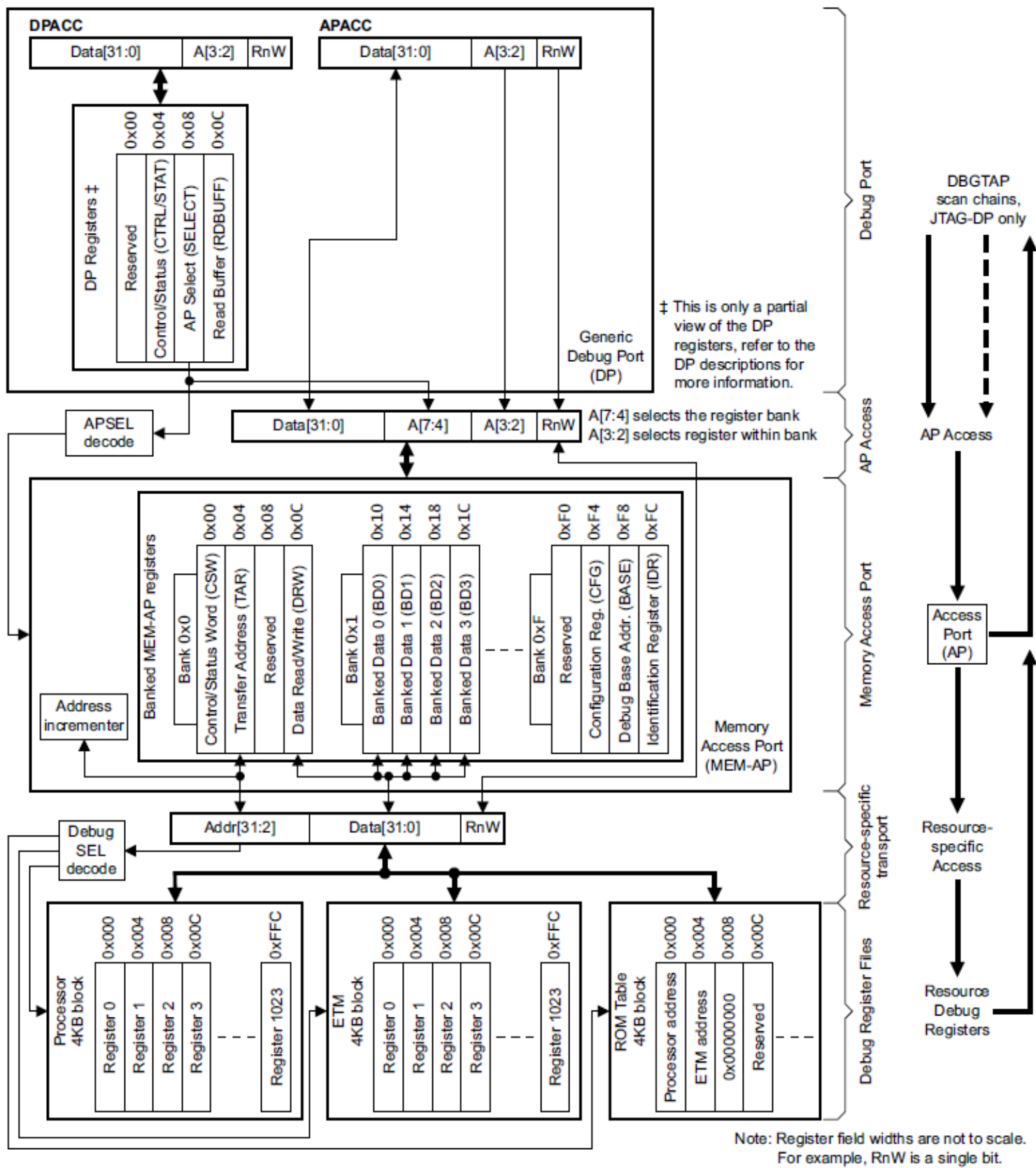


Figure 9-2 ARM Debug Interface Debug Port and Mem Access Ports (from ARM.com ARM Debug Interface v5 Specification)

10 Procedure

1. Compiled arm-none-eabi toolchain on Ubuntu 14.04 PC^[1]
Certain optional components were skipped to overcome build errors:
`./build-toolchain.sh --build_type=ppa,debug --build_tools=/usr/bin --skip_steps=mingw32 --skip_steps=manual`
2. Configured and built arm-none-eabi-gdb separately using debug flags:
 - a. installed libexpat1-dev, python-dev using apt-get
 - b. modified ./configure script to use `-O0` and `-g3`
 - c. `$make clean`
 - d. `$make distclean.`
 - e. `./configure --with-expat --with-python=/usr/bin --target=arm-none-eabi`
3. Installed libUSB, pyUSB, pyOCD
4. Installed Beagle 5000 USB Protocol Analyzer Drivers and DataCenter.
5. Compiled HelloWorld program using arm-none-eabi-gcc

Remote debugging the target using arm-none-eabi-gdb & debugging arm-none-eabi-gdb using gdb:
terminal1: for pyOCD

6. start pyOCD -> `$python2.7 /test/gdb_server.py --port=3334`

terminal 2: for arm-none-eabi-gdb

7. start arm gdb -> `$arm-none-eabi-gdb homework3.elf`
8. (gdb)target remote localhost:3334

terminal3: for gdb -tui

9. Get pid -> `$pidof arm-none-eabi-gdb`
10. start gdb in the gdb source directory -> `$sudo gdb -tui`
11. (gdb) set directories /path/of/gdb/source
12. (gdb) attach <pid-from-step9> // this stops execution of arm-none-eabi-gdb
13. (gdb) b net_write_prim
14. (gdb) c // this resumes execution of arm-none-eabi-gdb

terminal 2: for arm-none-eabi-gdb

15. (gdb) b main
16. (gdb) load
17. (gdb) c // (now mbed target program breaks at main())
18. (gdb) s // (now arm-none-eabi-gdb breaks at net_write_prim())

terminal 3: for gdb

19. (gdb) Ctrl-x a // turn TUI off to view back trace
20. (gdb) backtrace // the entire call stack for encoding step command as RSP and writing to socket
21. (gdb) x/6xb bu f // show the contents of RSP packet

11 References

1. [LaunchPad GCC Arm embedded](#)
2. [pyOCD](#)
3. ARMMbed, [CMSIS DAP Interface Firmware](#)
4. Arm.com , [ARM® Debug Interface v5 Architecture Specification](#)
5. Arm.com, [CoreSight™ Components Technical Reference Manual](#)
6. Embecosm , [Howto: GDB Remote Serial Protocol, Writing a RSP Server](#)