The Step Command

A Closer Look

Submitted by: Kripa Varma

Contents

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 that

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 are briefly outlines the architecture of each of the relevant components and the control flow within the component as it is 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 : LPC11U24 (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.

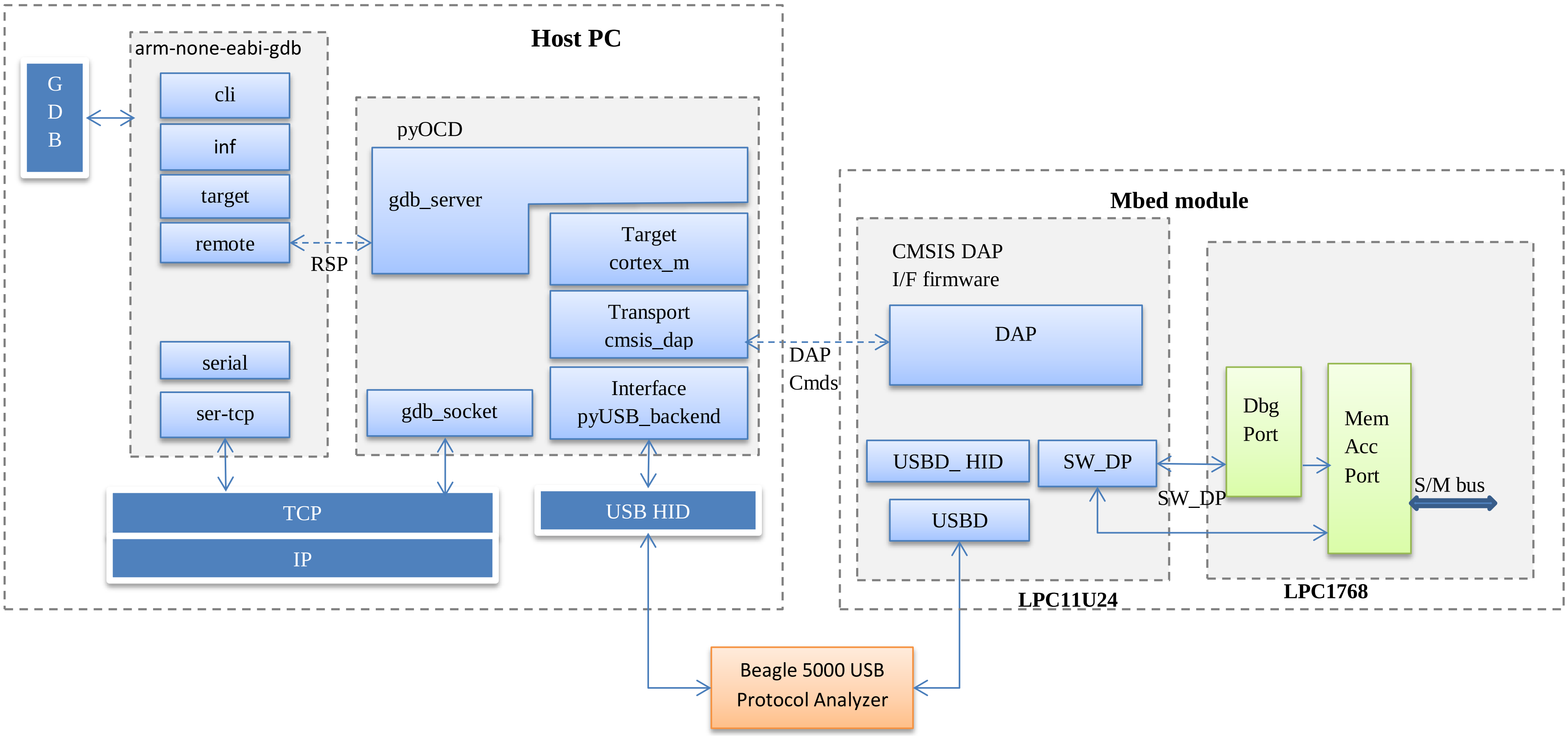


Figure 2-1 : Debugger setup

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 remoe" 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 (0xE000EDF0)
   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, 0x5,0xE000EDF0, 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 LPC11U24'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 and 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

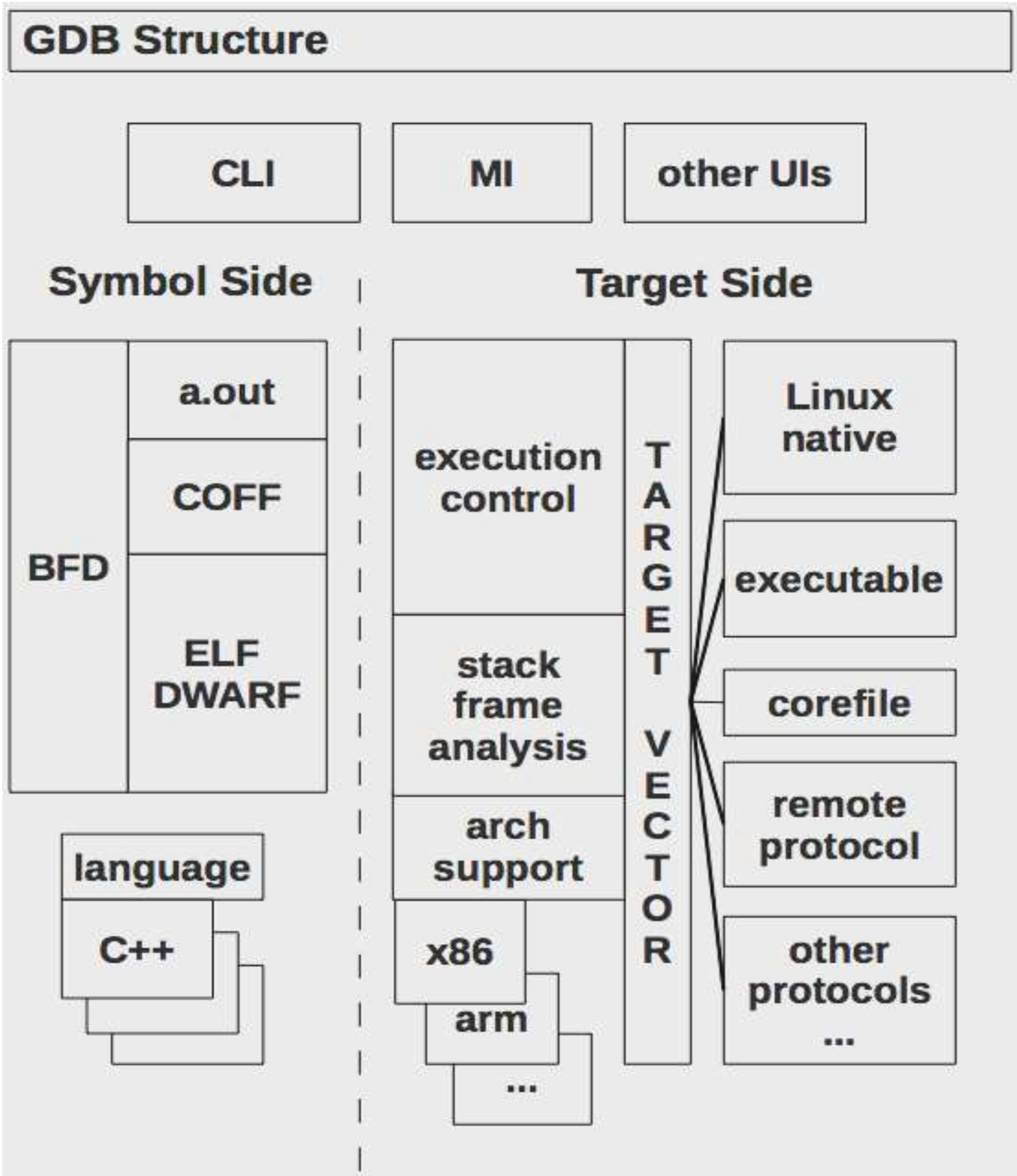


Figure 4-1 GDB Internal blocks

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.

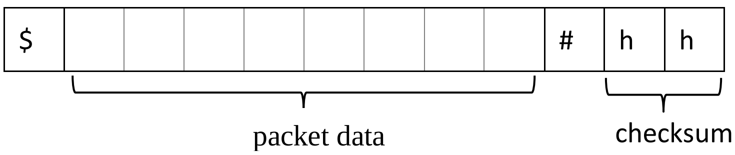
4.2 The Target Side

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 Protoco**l (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. **<u>In the case under study the pyOCD/test/gdb_server.py implements the RSP protocol.</u>**

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.

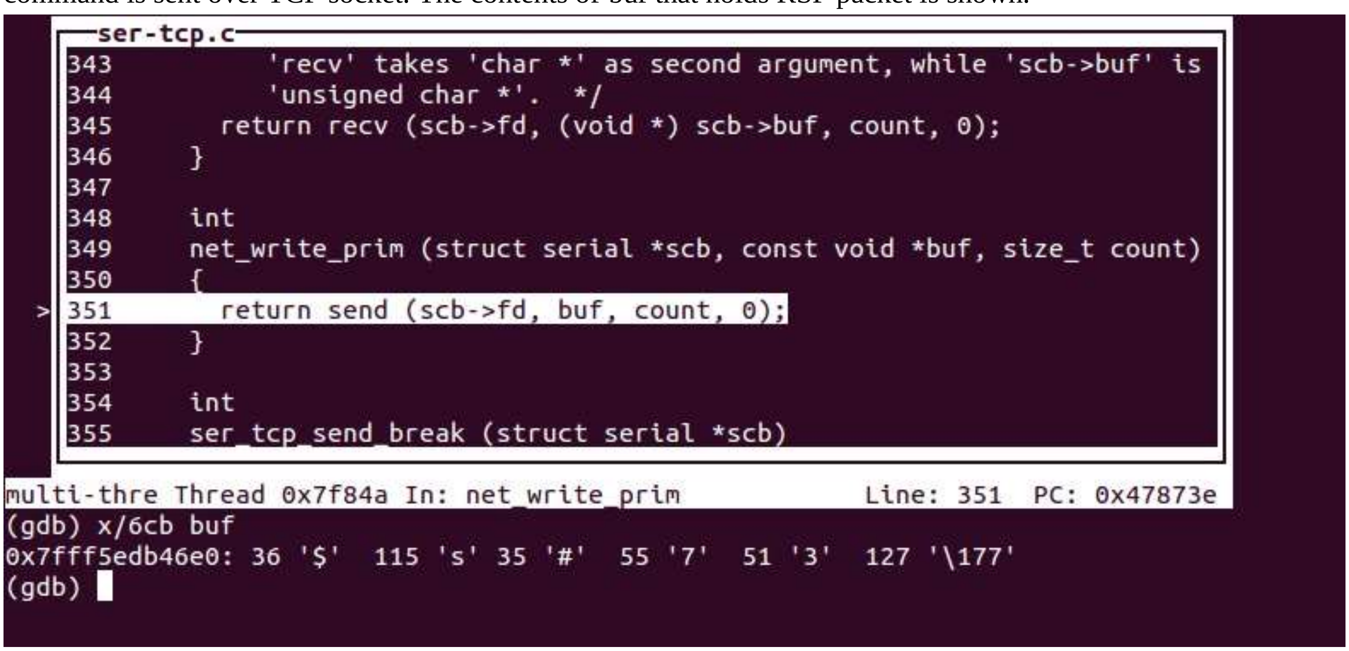


Figure 4-2 GDB RSP Step command view

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

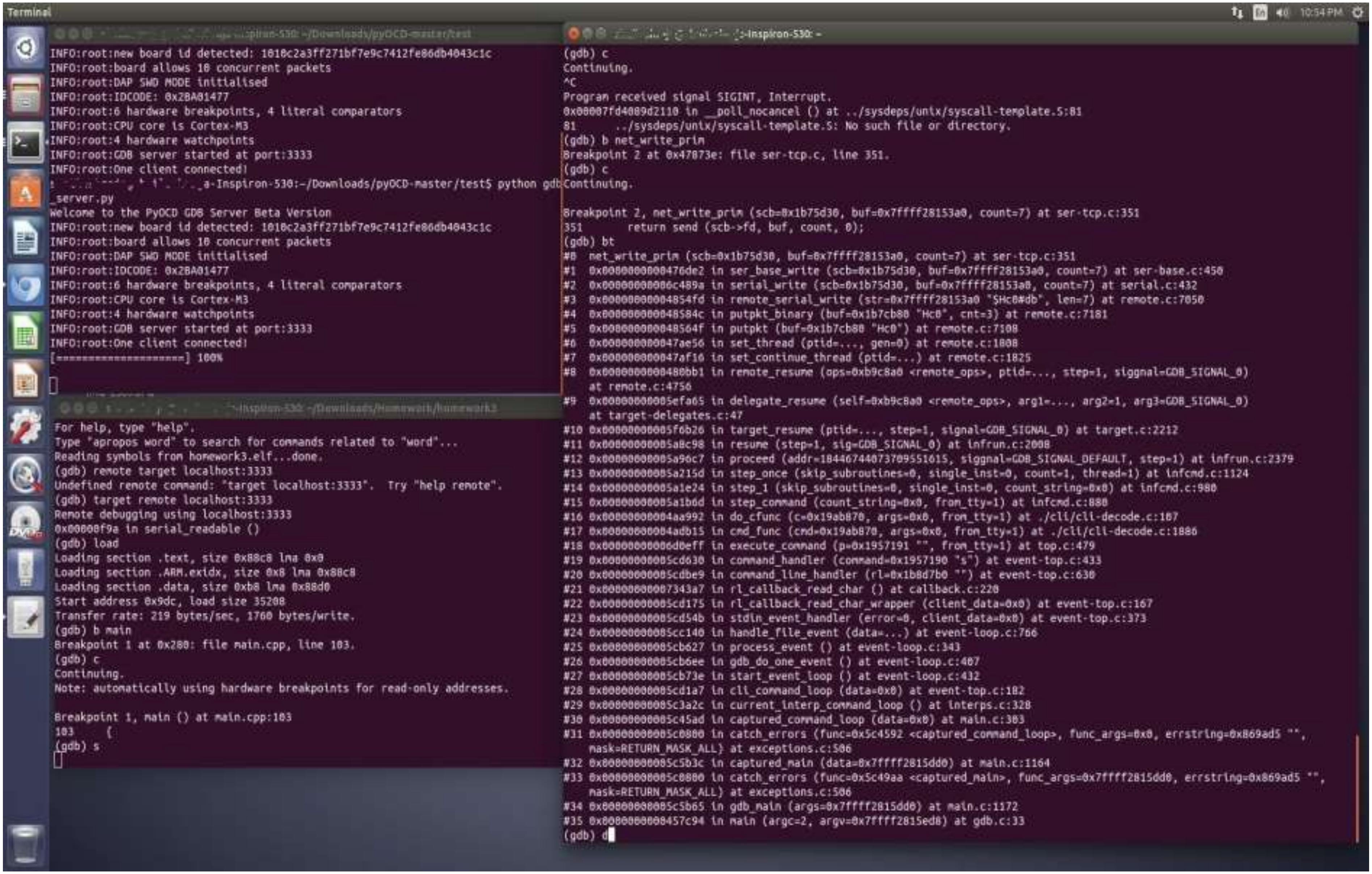


Figure 4-3 GDB step command backtrace

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.

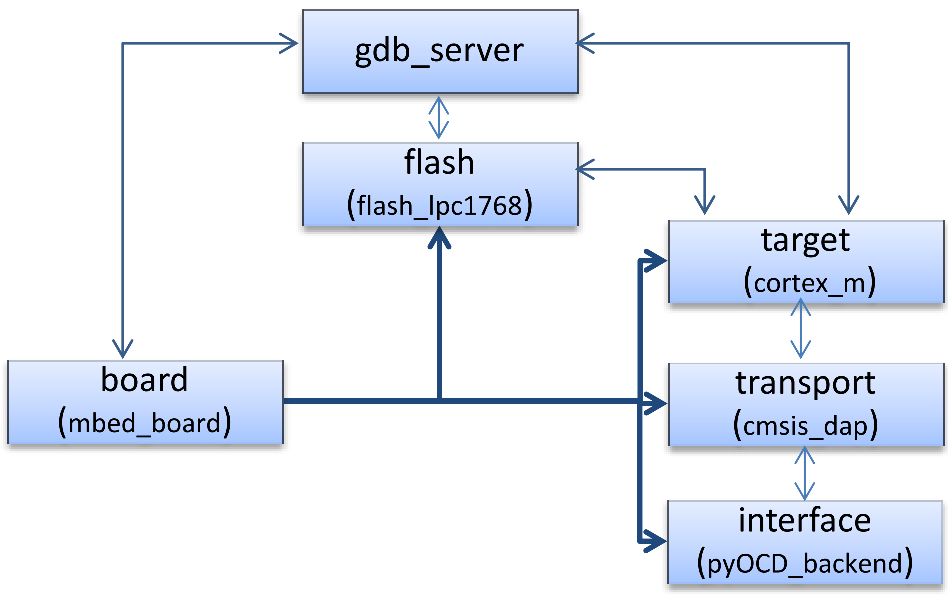


Figure 5-1 pyOCD Internals

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 boar 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 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}
```

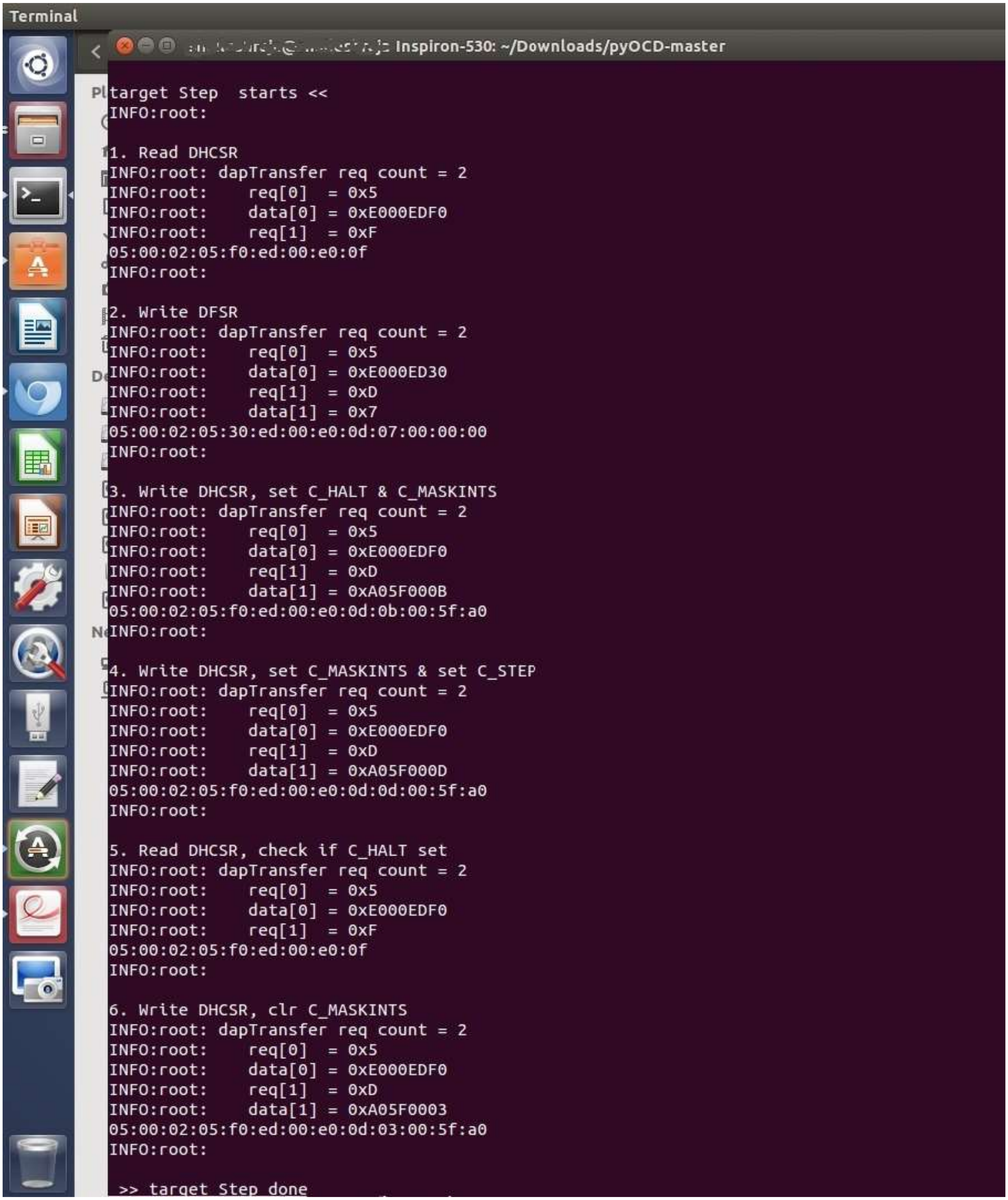


Figure 5-2 pyOCD DAP Commands

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

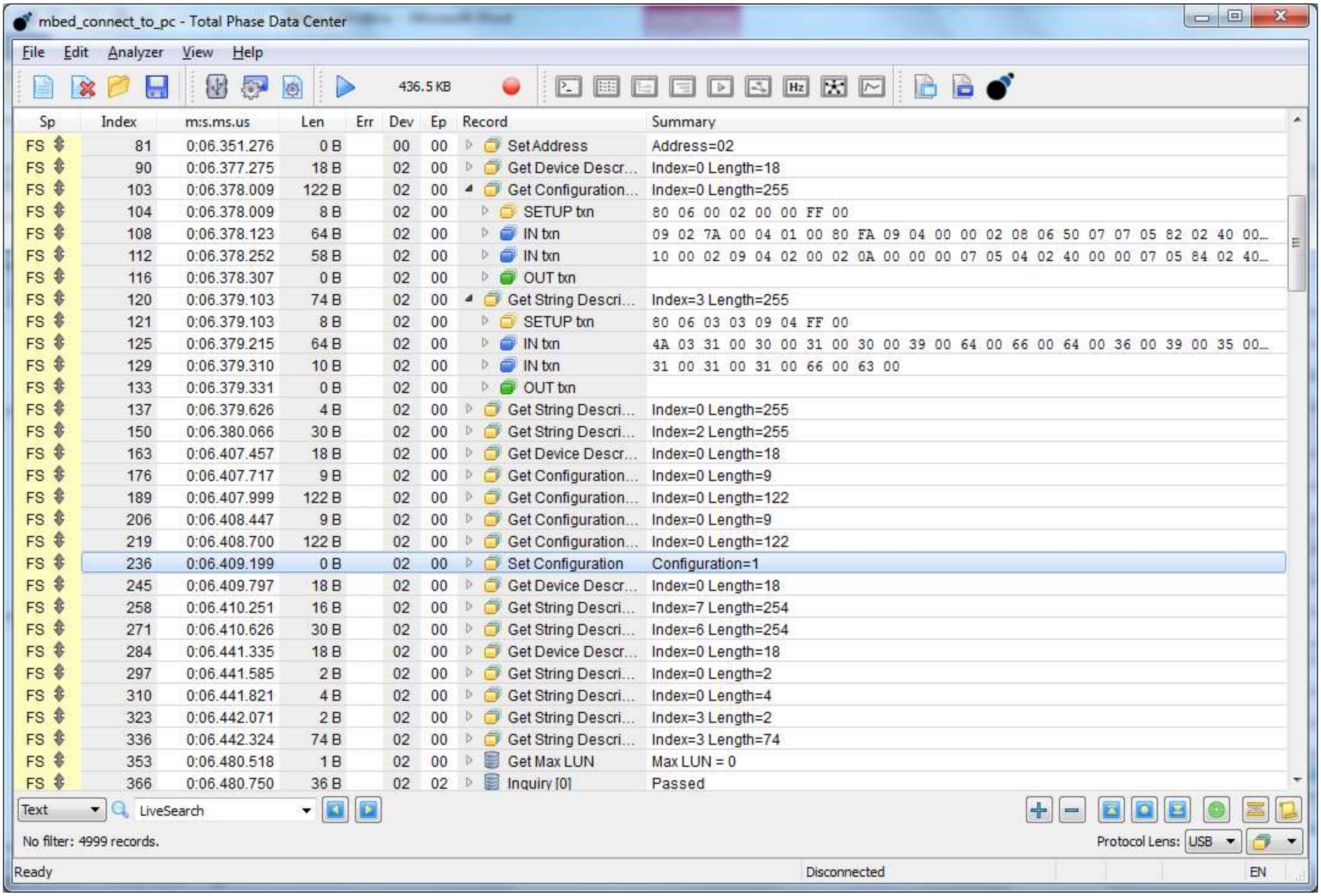
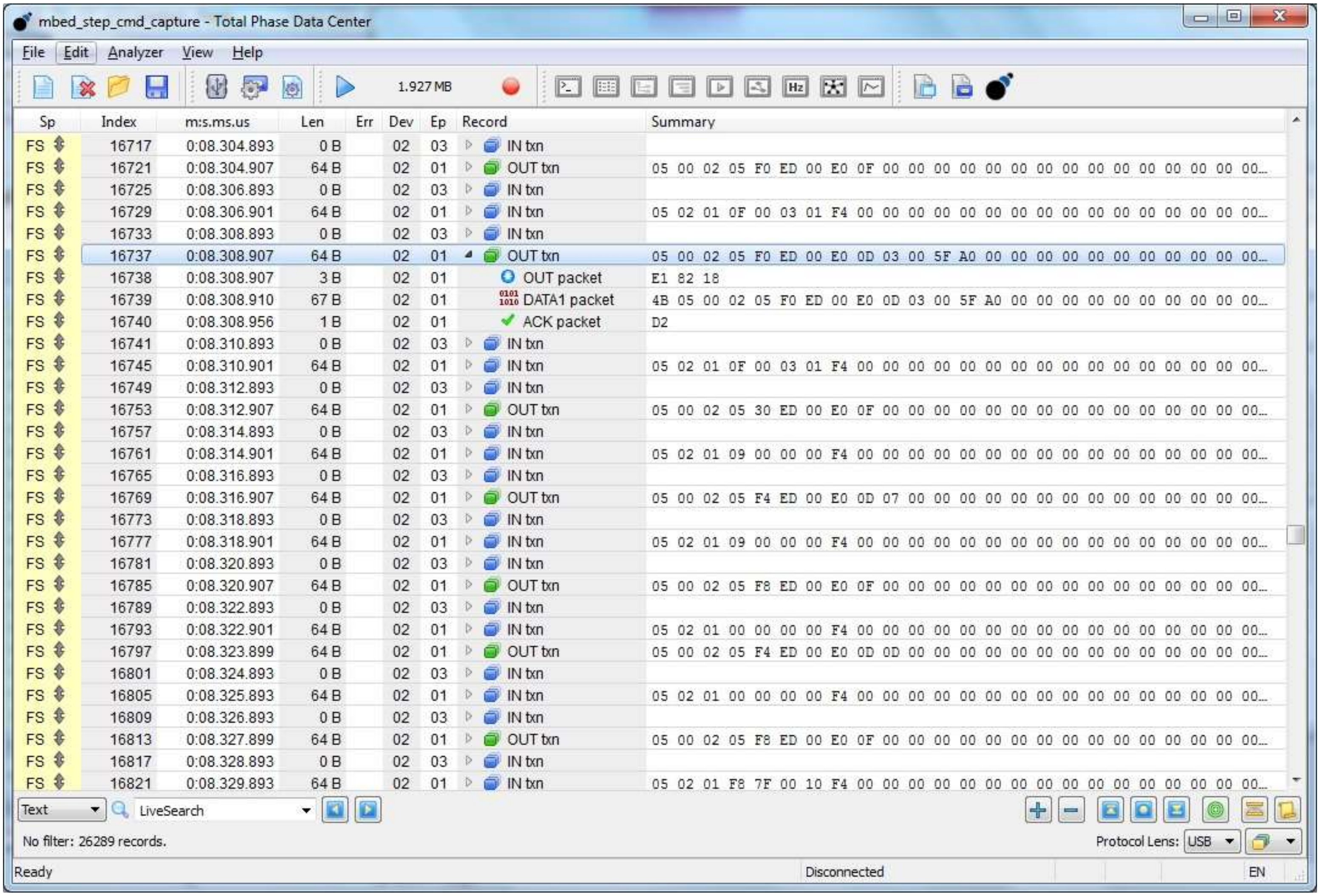


Figure 6-1 USB Protocol Analyzer capture - Mbed USB Device Configuration

6.6 Step Command Capture



Sp	Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
FS	16717	0:08.304.893	0 B		02	03	IN txn	
FS	16721	0:08.304.907	64 B		02	01	OUT txn	05 00 02 05 F0 ED 00 E0 0F 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16725	0:08.306.893	0 B		02	03	IN txn	
FS	16729	0:08.306.901	64 B		02	01	IN txn	05 02 01 0F 00 03 01 F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16733	0:08.308.893	0 B		02	03	IN txn	
FS	16737	0:08.308.907	64 B		02	01	OUT txn	05 00 02 05 F0 ED 00 E0 0D 03 00 5F A0 00 00 00 00 00 00 00 00 00...
FS	16738	0:08.308.907	3 B		02	01	OUT packet	E1 82 18
FS	16739	0:08.308.910	67 B		02	01	DATA1 packet	4B 05 00 02 05 F0 ED 00 E0 0D 03 00 5F A0 00 00 00 00 00 00 00 00...
FS	16740	0:08.308.956	1 B		02	01	ACK packet	D2
FS	16741	0:08.310.893	0 B		02	03	IN txn	
FS	16745	0:08.310.901	64 B		02	01	IN txn	05 02 01 0F 00 03 01 F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16749	0:08.312.893	0 B		02	03	IN txn	
FS	16753	0:08.312.907	64 B		02	01	OUT txn	05 00 02 05 30 ED 00 E0 0F 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16757	0:08.314.893	0 B		02	03	IN txn	
FS	16761	0:08.314.901	64 B		02	01	IN txn	05 02 01 09 00 00 00 F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16765	0:08.316.893	0 B		02	03	IN txn	
FS	16769	0:08.316.907	64 B		02	01	OUT txn	05 00 02 05 F4 ED 00 E0 0D 07 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16773	0:08.318.893	0 B		02	03	IN txn	
FS	16777	0:08.318.901	64 B		02	01	IN txn	05 02 01 09 00 00 00 F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16781	0:08.320.893	0 B		02	03	IN txn	
FS	16785	0:08.320.907	64 B		02	01	OUT txn	05 00 02 05 F8 ED 00 E0 0F 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16789	0:08.322.893	0 B		02	03	IN txn	
FS	16793	0:08.322.901	64 B		02	01	IN txn	05 02 01 00 00 00 00 F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16797	0:08.323.899	64 B		02	01	OUT txn	05 00 02 05 F4 ED 00 E0 0D 0D 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16801	0:08.324.893	0 B		02	03	IN txn	
FS	16805	0:08.325.893	64 B		02	01	IN txn	05 02 01 00 00 00 00 F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16809	0:08.326.893	0 B		02	03	IN txn	
FS	16813	0:08.327.899	64 B		02	01	OUT txn	05 00 02 05 F8 ED 00 E0 0F 00 00 00 00 00 00 00 00 00 00 00 00 00...
FS	16817	0:08.328.893	0 B		02	03	IN txn	
FS	16821	0:08.329.893	64 B		02	01	IN txn	05 02 01 F8 7F 00 10 F4 00 00 00 00 00 00 00 00 00 00 00 00 00 00...

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.

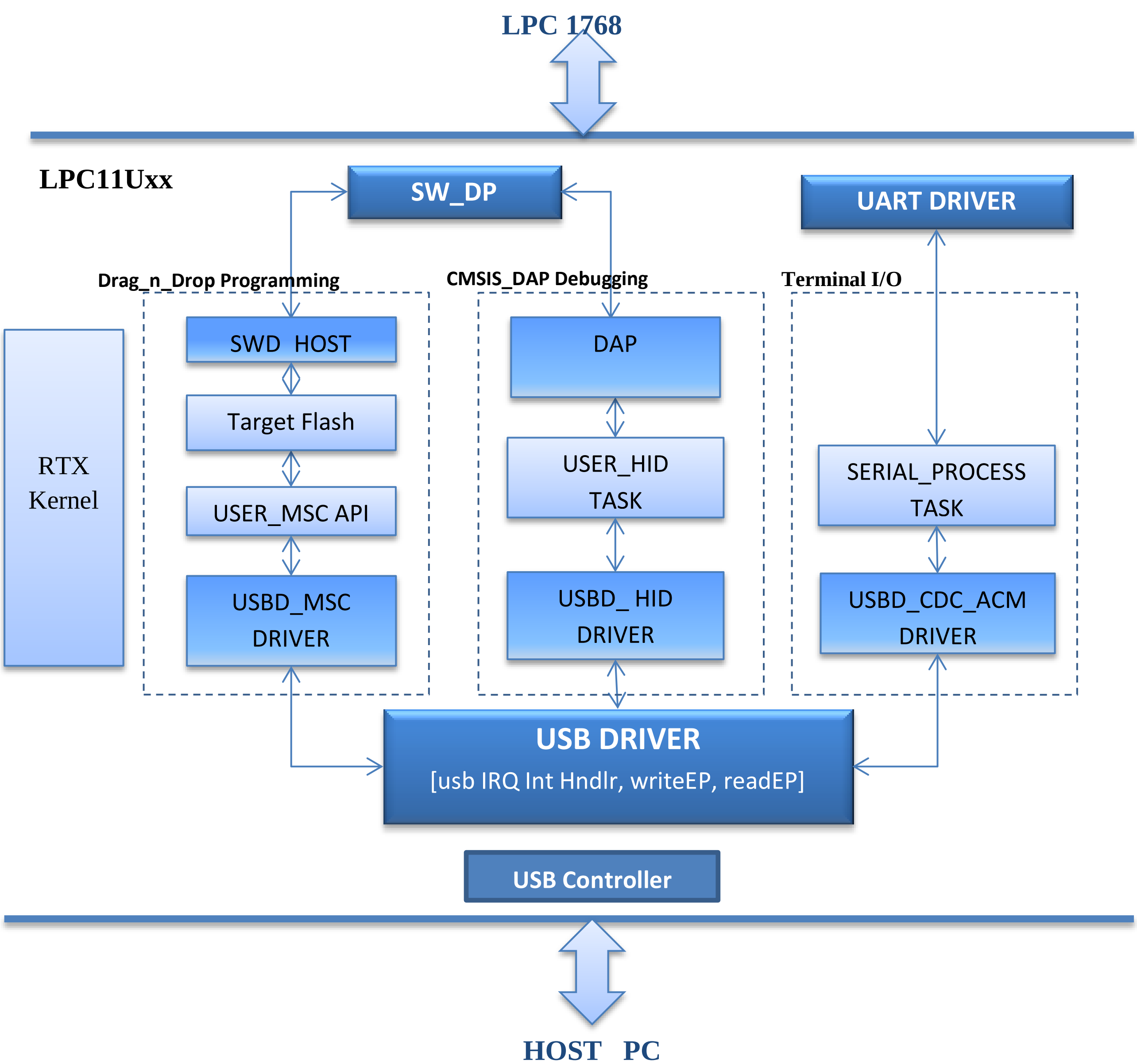


Figure 7-1 CMSIS-DAP Interface Firmware

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 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) .

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:

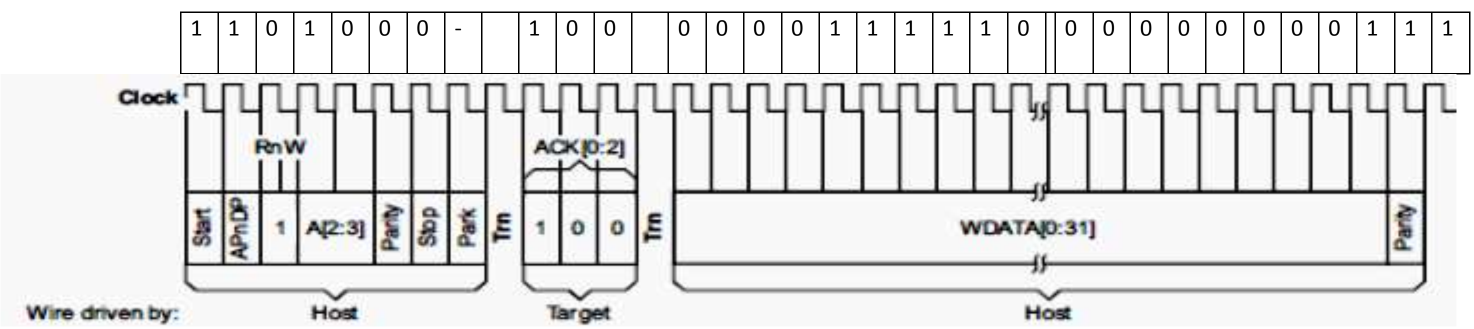


Figure 8-1 Serial Wire Bits Exchange for DHCSR Addr write to TAR Register

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.

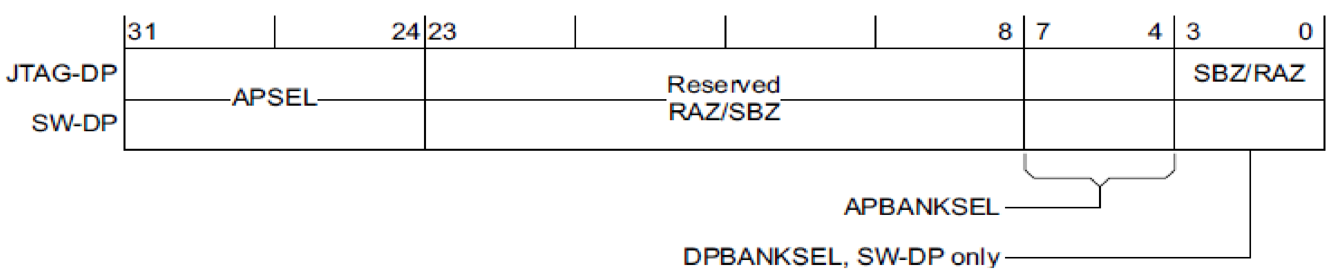
One of the four registers within the DP is the AP Select Register, SELECT. This register specifies a particular Access Port, and a bank of four 32-bit words within the register map of that AP. It enables up to 256 Access Ports to be implemented, and gives access to any one of 16 four-word banks of registers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



AP Select Register bit assignments

Bits	Function	R/W	Description
[31:24]	APSEL	W	Selects the current access port.

Bits	Function	R/W	Description
			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
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

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 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-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**	If the core is stalled on a load/store operation the stall ceases and

Bits	Type	Field	Function
			the instruction is forced to complete. This enables Halting debug to gain control of the core. It can only be set if: **C_DEBUGEN** = 1 **C_HALT** = 1 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.
[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: 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

Bits	Field	Function
		Register is also set to indicate the type of vector catch.
[2]	DWTRAP	*Data Watchpoint* (DW) flag: 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.

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

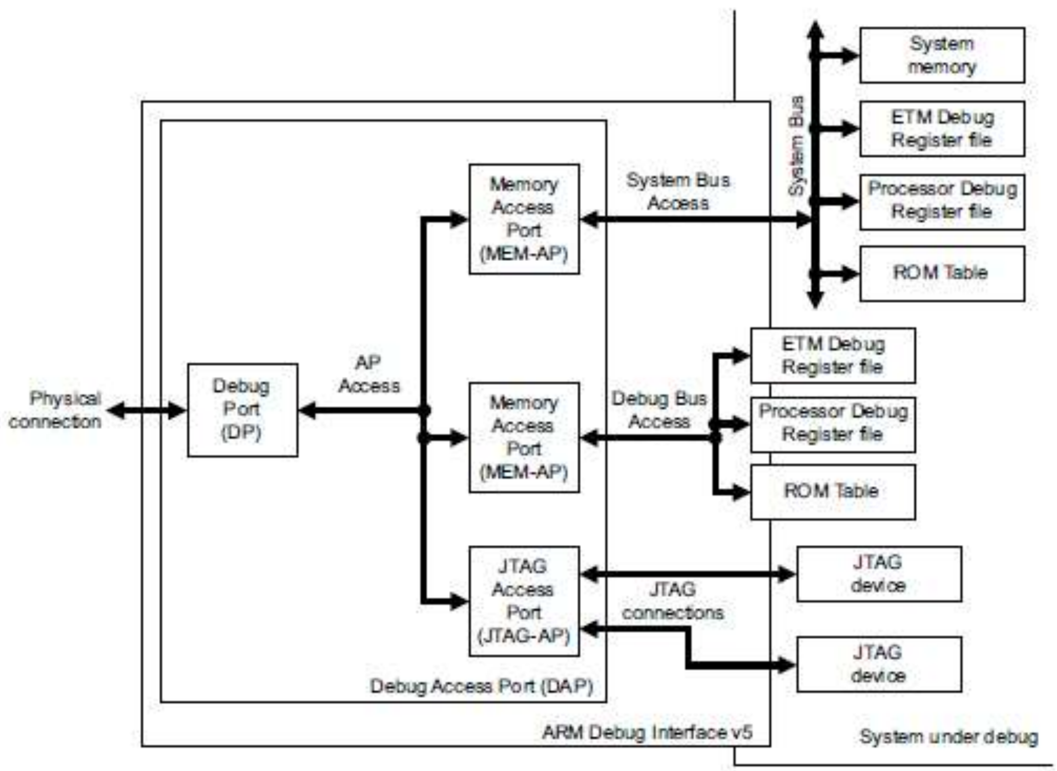


Figure 9-1 ARM Debug Interface – Access Ports

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

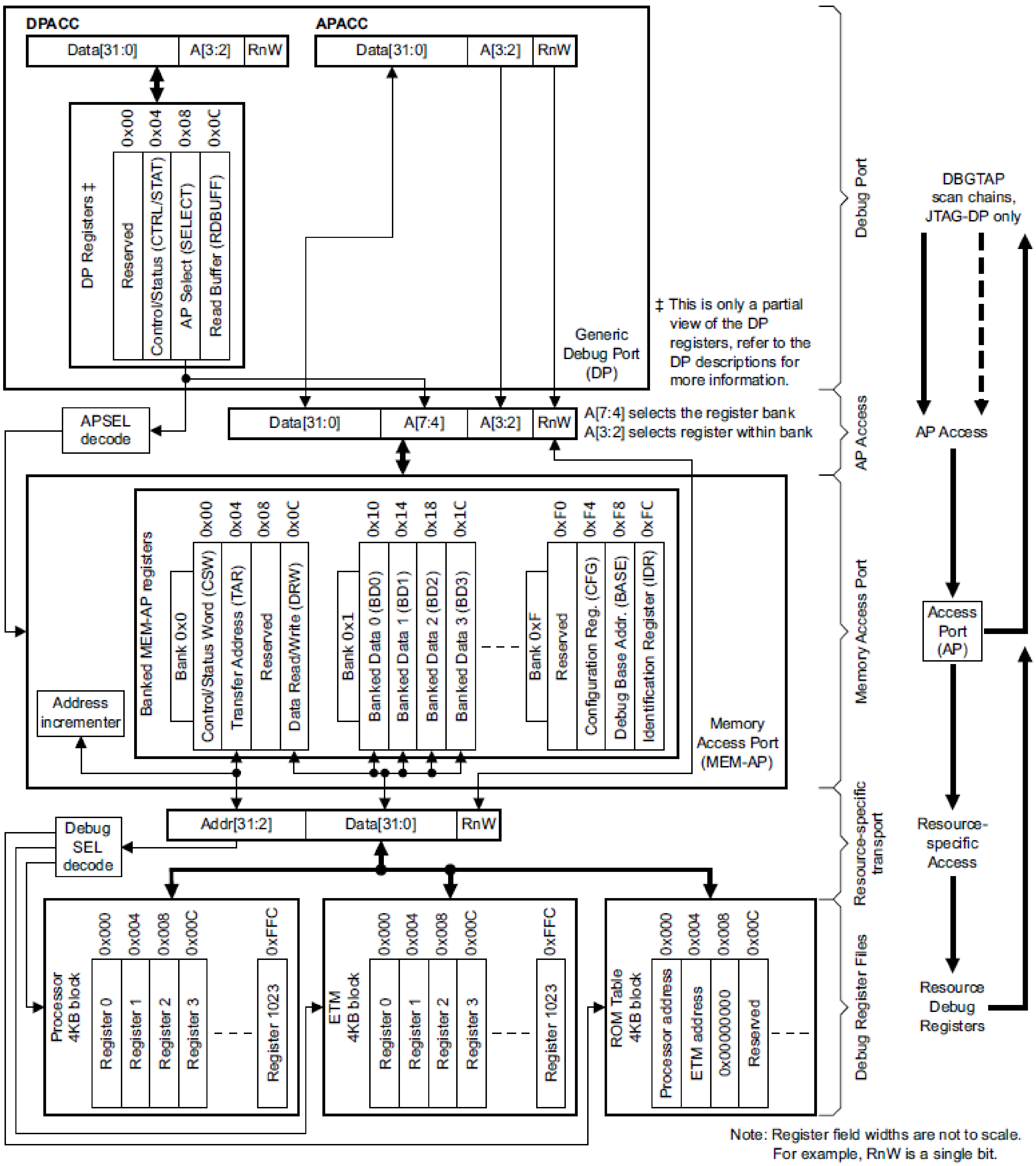


Figure 9-2 ARM Debug Interface Debug Port and Mem Access Ports

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