LibSWD
Serial Wire Debug Open Framework for Low–Level Embedded Systems Access

Tomasz CEDRO, Marcin KUZIA, Antoni GRZANKA
ORANGE LABS POLAND, WARSAW / TP R&D
Obrzeżna 7, 02-691 Warsaw, Poland.
WARSAW UNIVERSITY OF TECHNOLOGY
Faculty of Electronics and Information Technologies
Nowowiejska 15/19, 00-665 Warsaw, Poland.
POLISH INTERDISCIPLINARY NEUROSCIENCE GROUP
http://www.tomek.cedro.info
tomek@cedro.info

***Abstract*—Modern microelectronics has settled for good in embedded systems that run our everyday life in areas of home entertainment, telecommunications, medical equipment, various industrial applications, even military and aerospace systems. Increasing complexity of these systems requires new tools for development, testing and security analysis. Presented work is an ongoing effort to create from scratch a Free and Open framework for low–level access (In–Circuit–Emulation and On–Chip–Debug) into ARM–Cortex [7] based devices that use new SWD bus (a JTAG alternative). LibSWD [1] is a BSD–licensed software library and it is being integrated into well known Open–Source applications such as UrJTAG [11] and OpenOCD [12].**

I. Introduction

A. Software and Hardware

MODERN embedded systems are the state of the art mixture of hardware and software. Hardware (electronic components) makes it possible for software (computer programs) to operate, but on the other hand it is the software that makes hardware intelligent, communicative and user friendly with help of the Operating System. Before device reaches state of the final product it needs to conquer long and hard road of design, development and testing, utilizing unimaginable resources and thousands of minds at work. Only few people really know how hard and important it is to create basic components that makes Operating System (high–level components) functional – bootloaders, drivers, compilers, linkers, and many more low–level services that create a virtual bridge between world of software and hardware. Figure 1 shows how many various peripherals are included in just one simple chip of the modern microcontroller.

B. Low–Level Access

The hardware and physical parts of the system are often referred as *low–level* because they make operation of logical and software structures (so called *high–level* functions) possible. On the other hand it is still possible to access low–level system resources with use of special physical interface connection, even if the software is not yet operational (i.e. no bootloader and operating system), to test the software components from the hardware point of view (i.e. debugging the bootloader or the operating system components), or simply to test the hardware itself.

C. Standards

JTAG is the best known standard (`IEEE1149.1` [3]) of the low–level access method to various microcontroller architectures internals such as CPU, Peripherals and Memory, making it possible to take full control over target system at the hardware level. This is very important feature for developers, testers and security researchers to have unlimited access to every information in the system. But as JTAG has some limitations, the alternative solution called *Serial Wire Debug (SWD)* was introduced by ARM Incorporated [4] and implemented in their new *ARM Cortex* devices family.

This paper describes low–level access basics into modern microprocessor systems with use of *LibSWD* software library that implements the Serial Wire Debug transport in Platform Independent, Free and Open–Source manner.

II. Joint Test Action Group (JTAG)

A. What is JTAG

As mentioned in previous section the JTAG is the well known low–level access mechanism to access internals of the microprocessor systems. The `IEEE1149.1` [3] standard defines a finite state machine (see Figure 2) that defines access to the Test Access Port / Debug Access Port registers (Instruction and Data Registers) in order to access system internals such as CPU, Peripherals, Memory, Debug Unit, etc.

However, the internal organization of the specific registers and access ports itself is target dependent and vary across different CPU architectures. Microprocessors designed by ARM use *ARM Debug Interface* standard [5] (described later in this paper), microprocessors designed by MIPS use

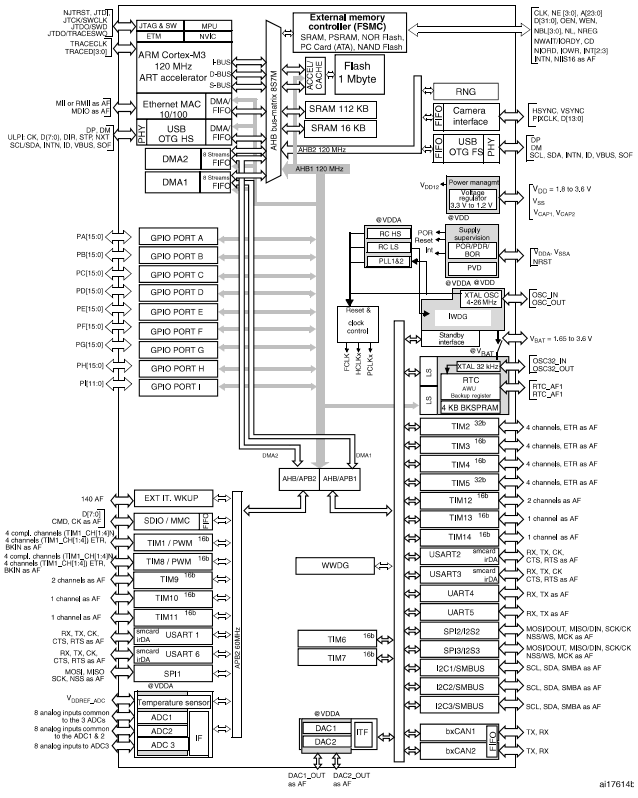

Fig. 1. ARM Cortex CPU in STM32 microcontroller, block diagram [9].

EJTAG standard [8], other devices will probably use their own internal organization. This makes things complicated and require target specific software implementation of the tools, while the electrical interface and the access method remains common for all JTAG aware devices.

B. Signalling

The JTAG Test Access Port requires following electrical signals to operate:

- TDI (Test Data Input) that provides input bitstream into scan chain.
- TDO (Test Data Output) that provides output bistream from the scan chain.
- TCK (Test ClocK) that provides synchronous clock source to the target system.
- TMS (Test Mode Selection) that provides operation mode selection of the target system and is vital for proper state machine operations.
- TRST (Test ReSeT) that can reset target access port.
- GND (Ground) that provides voltage reference point of zero potential.
- VCC (Voltage Supply) that provides reference voltage for interface input buffers and voltage level shifters.

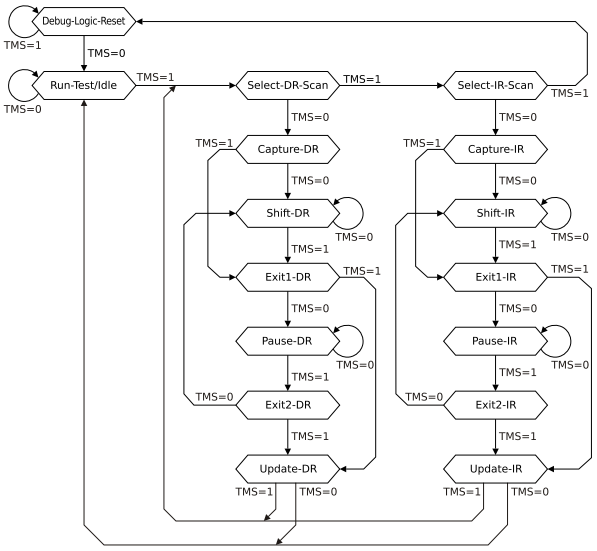

Fig. 2. JTAG TAP/DAP State Machine [6].

- SRST (System ReSeT) that can reset whole target system (useful for development purposes and remote management).

C. Practice

For some systems TRST signal can be omitted and equivalent functionality can be achieved only with use of TMS and TCK pins, also SRST can be sometimes activated using target internals. Therefore at least six signals are required for JTAG connection to work (TRST and SRST can be ommited), and those signals have corresponding pins somewhere on the system PCB as a group of test points or even a dedicated connector.

Finding JTAG connector on an unknown board is a very complex subject (large enough for a separate publication) and usually this requires knowledge obtainable from technical/service specification of the target CPU/SoC, or even hardware reverse–engineering techniques.

Accessing mentioned signals also brings some security risk to the target system, therefore very often Test Access Port functional block is disabled or at least hidden by spreading test points across whole PCB.

D. Multiple Devices

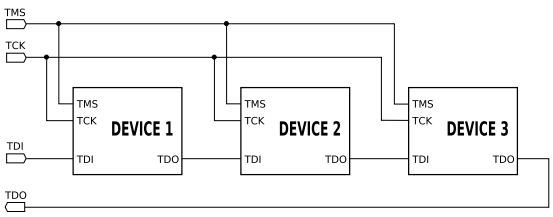

Fig. 3. JTAG Scan Chain [10].

As presented on Figure 3 JTAG aware devices can be daisy–chained together serially one after another, where input of the

chain (TDI) is the input to the first device, output of the first device becomes input of the second device, and the last device output is the chain output (TDO). In this case TCK, TMS and TRST control signals are common to all devices, only one interface is necessary to access all of the chain elements, only one device can be active at time. Unfortunately, whole chain is as fast as its slowest component and the malfunction of one component breaks whole chain.

III. Debug Access Port

A. What is DAP

Although Serial Wire Debug (SWD) is somehow alternative to JTAG method of low–level access to ARM Cortex devices, both JTAG and SWD exist to transport commands to/from *Debug Access Port* (DAP). Debug Access Port is a dedicated silicon on–chip device that serves as a gateway between debug host (software+interface) and the microprocessor internals (memory, peripherals, debug unit, etc).

To understand how low–level access works in ARM microprocessor based embedded systems it is essential to understand how Debug Access Port (DAP) works, please read the „ARM Debug Interface" specification [5] for detailed information.

DAP is far more complex than Bootloader or Operating System Serial Port Console with Commandline Interface (CLI) implementation because it works on a hardware register transfer level not in a software domain. Its purpose is to access dedicated on–chip functional blocks using Access Ports (AP) as their interface. Reading and writing to selected AP registers triggers operations that can result in memory access (when using MEM–AP), general bus access (AHB–AP), debug functionality, and others.

Not all internal peripherals have their corresponding Access Ports, therefore only those functionalities with AP implemented in the hardware can be accessed from outside by DAP.

B. Organization

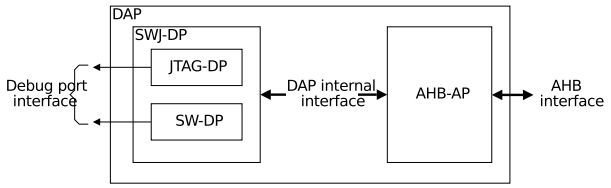

Fig. 4. ARM Cortex Debug Access Port organization [6].

Figure 4 shows the internal organization of the Debug Access Port. Debug Port Interface (also known as Test Access Port in JTAG terminology) is the physical signalling connector. DAP Internal Interface connects DAP to multiple Access Ports (AHB–AP as example in this case).

C. Functionality

Depending on hardware capabilities, from the debugger perspective, DAP can support JTAG transport (using JTAG–DP), SWD transport (using SW–DP), or both. The block responsible for decoding the transport bitstream is called *Serial Wire and JTAG Debug Port* (SWJ–DP), it contains both SW–DP and JTAG–DP. Methods for read, write and error handling operations are a bit different for JTAG and SWD, but in general they access the same DAP registers.

DAP registers makes it possible to read/write Access Ports in order to perform operations on selected on–chip subsystems. Therefore, DAP is only a gateway between debugger host/software and the target on–chip subsystem, it can store read/write results, error codes, AP addresses and eventually abort their operations, but DAP alone without target AP is pretty useless.

D. Registers

Debug Access Port consists of following registers:

- IDCODE – Identification Code Register that contains unique 32–bit identification bitstream with manufacturer code, device and version code according to the JEDEC standard.
- ABORT – Abort Register is used to abort ongoing or stalled operations, also to clear error flags on SW–DP.
- CTRL/STAT – Control/Status Register contains all important information on status of the DAP, also allows to control DAP functions.
- SELECT – AP Select Register function is to select Access Port address to be accessed and Register Bank within the selected AP.
- RDBUFF – Read Buffer has different meaning in JTAG–DP (to initiate a read operation) and SW–DP (to return last AP read result, because reading the data register in AP initiates a read and produces WAIT state by default, however sequential reads from RDBUFF will give unpredictabe result!).
- WCR – Wire Control Register has the same address as CTRL/STAT but it is accessed when SELECT bit 0 value is set to 1. Its purpose is to control physical SWD bus parameters.
- RESEND – Read Resend Register is available only on SW–DP and its purpose is to recover read data from corrupted transfer without repeating AP transfer.

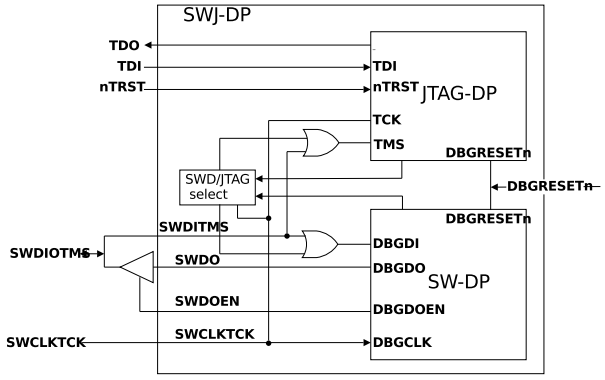

Fig. 5. Serial Wire and JTAG Debug Port organization [6].

IV. Serial Wire Debug

A. What is SWD

As explained in previous sections Serial Wire Debug (SWD) is an alternative to JTAG transport method to perform operations on on–chip Debug Access Port (DAP) and peripheral specific Access Ports (AP) in ARM Cortex devices. To understand the complexity of its software implementation it is first important to understand SWD basics. It is packed–based half–duplex serial protocol, where each transfer is initialized and controlled by a debugger host.

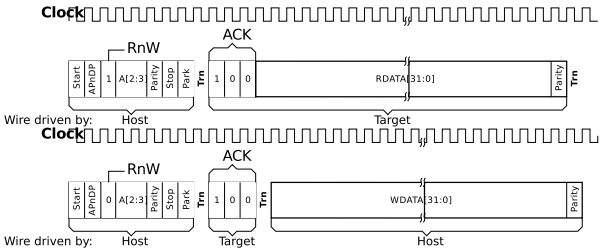

Fig. 6. Successful Read and Write SWD Operations [6].

Figure 7 presents example successful read and write operation diagrams (these are not exact timing diagrams!) which nicely visualise SWD packet elements:

- REQUEST – defines access type (read, write) and location (AP/DP register address). It also contains parity control bit to avoid protocol errors.
- TRN – turnaround gives time for interface buffers to switch bus direction from read to write. TRN length in clock cycles can be set using WCR DP register.
- ACK – returns the Target response status – it can be OK, WAIT (to retry transfer), FAULT (when unrecoverable error occurs), or no response when Protocol Error Sequence occurs (target is powered down, target does not understand the request, etc).
- DATA – holds the 32–bit data payload.
- PARITY – even parity bit to control DATA integrity.

B. Signalling

Figure 9 tells more about SWJ–DP signalling. JTAG and SWD use the same signal / port pins – JTAG TMS is the SWD SWDIOTMS, TCK is the SWCLKTCK. SWD only use two signal pins as opposed to five signal pins used by JTAG. To activate SW–DP or JTAG–DP special 16–bit sequence should be applied on the TMS signal, however JTAG–DP is active by default for backward compatibility reasons. It is therefore possible to use one Serial Wire and JTAG interface to access SWJ enabled device, which is elegant solution.

V. LibSWD

A. What is LibSWD

LibSWD is a first in the world platform and hardware independent Open–Source framework to deal with with Serial Wire Debug Port in accordance to ADI (Arm Debug Interface, version 5.0 at the moment) specification [5]. It is

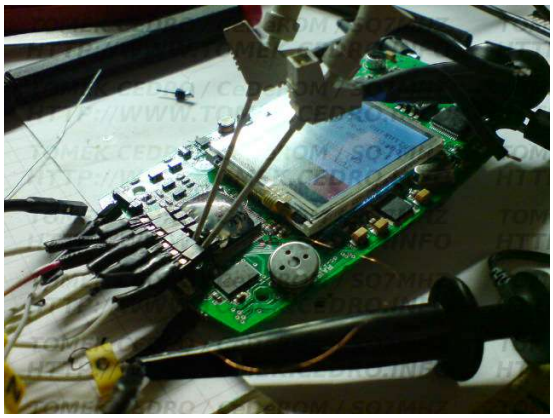

Fig. 7. SWD/JTAG connector tapped into a real hardware [2].

released under 3–clause BSD license and documented with Doxygen. For more information please visit project website at http://libswd.sf.net.

B. What it can do

Serial Wire Debug is an alternative to JTAG (IEEE1149.1) transport layer for accessing the Debug Access Port in ARM-Cortex based devices. LibSWD provides methods for accessing the Debug Access Port and various Access Ports registers on the target.

From programmer and user perspective it is as simple as calling `swd_dap_detect()` and `swd_{dp,ap}_{read,write}` functions to perform low–level operations on the target hardware. LibSWD takes care of bitstream generation on the wire using simple but flexible API that can reuse capabilities of existing applications for easier integration, specifically the existing interface drivers code.

Every bus operation such as control, request, turnaround, acknowledge, data and parity packet is named a „command" represented by a `swd_cmd_t` data type that builds up the queue that later can be flushed into real hardware using standard set of (application–specific) driver functions. This way LibSWD is almost standalone and can be easily integrated into existing utilities for low–level access and only requires in return to define driver bridge that controls the physical interface interconnecting host and target.

Drivers and other application–specific functions are `extern` type and located in external file crafted for that application and its hardware. Figure 8 shows how easily LibSWD can be integrated into existing low–level access applications.

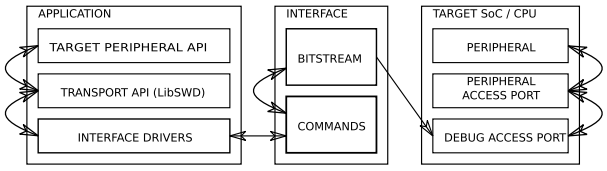

Fig. 8. LibSWD integration with existing low–level access applications.

C. SWD Context

The most important data type in LibSWD is `swd_ctx_t` structure, a context that represents logical entity of the swd bus (transport layer between host and target) with all its parameters, configuration and command queue. Context is being created with `swd_init()` function that returns pointer to allocated virgin structure, and it can be destroyed with `swd_deinit()` function taking the pointer as argument. Context can be set only for one interface–target pair, but there might be many different contexts in use if necessary, so amount of devices in use is not limited.

D. Functions organization

All functions in general operate on pointer type and return number of processed elements on success or negative value with `swd_error_code_t` on failure. Functions are grouped by functionality that is denoted by function name prefix (ie. `swd_bin*` are for binary operations, `swd_cmdq*` deals with command queue, `swd_cmd_enqueue*` deals with creating commands and attaching them to queue, `swd_bus*` performs operation on the swd transport system, `swd_drv*` are the interface drivers, etc).

Because programs using LibSWD for transport can queue multiple operations and don't handle errors of each transaction appropriately, `swd_drv_transmit()` function verifies the ACK and PARITY operation results directly after execution (read from target) and return error code if necessary. When error is detected and there were some pending operations enqueued for execution, they are discarded and removed from the queue (they would not be accepted by the target anyway), the queue is then again ready to accept new transactions (i.e. error handling operations).

Standard end–users are encouraged to only use high level functions (`swd_bus*`, `swd_dap*`, `swd_dp*`) to perform operations on the swd transport layer and the target's DAP (Debug Access Port) and its components such as DP (Debug Port) and the AP (Access Port). More advanced users however may use low level functions (`swd_cmd*`, `swd_cmdq*`) to group them into new high–level functions in order to automate some tasks (such as existing high–level functions does).

Functions of type `extern` are the ones to implement in external file by developers that want to incorporate LibSWD into their application. Context structure also has `void` pointer in the `swd_driver_t` structure that can hold address of the external driver structure to be passed into internal swd drivers (`extern swd_drv*` functions) that wouldn't be accessible otherwise.

E. Commands

Bus operations are split into *commands* represented by `swd_cmd_t` data type. They form a bidirectional command queue that is part of `swd_ctx_t` structure. Command type, and so its payload, can be one of: control (user defined 8–bit payload), request (according to the standard), ack, data, parity (data and parity are separate commands!), trn, bitbang and idle (equals to control with zero data). Command type is defined by `swd_cmdtype_t` and its code can be negative (for MOSI operations) or positive (for MISO operations) – this way bus direction can be easily calculated by multiplying two operation codes (when the result is negative bus will have to change direction), so the LibSWD „knows" when to put additional TRN command of proper type between enqueued commands.

Payload is stored within union type and its data can be accessed according to payload name, or simply with data8 (`char`) and data32 (`int`) fields. Payload for write (MOSI) operations is stored on command creation, but payload for read (MISO) operations becomes available only after command is executed by the interface driver. There are 3 methods of accessing read data – flushing the queue into driver then reading queue directly, single stepping queue execution (flush one–by–one) then reading context log of last executed command results (there are separate fields of type `swd_transaction_t` in `swd_ctx_t`'s log structure for read and write operations that are updated by `swd_drv_transmit()` function before write and after read), or providing a double pointer on command creation to have constant access to its data after execution.

After all commands are enqueued with `swd_cmd_enqueue*` function set, it is time to send them into physical device with `swd_cmdq_flush()` funtion. According to the `swd_operation_t` parameter commands can be flushed one–by–one, all of them, only to the selected command or only after selected command. For low level functions all of these options are available, but for high–level functions only two of them can be used – `SWD_OPERATION_ENQUEUE` (but not send to the driver) and `SWD_OPERATION_EXECUTE` (all unexecuted commands on the queue are executed by the driver sequentially) – that makes it possible to perform bus operations one after another having their result just at function return, or compose more advanced sequences leading to preferred result at execution time. Because high–level functions provide simple and elegant manner to get the operation result, it is advised to use them instead dealing with low–level functions (implementing memory management, data allocation and queue operation) that exist only to make high-level functions possible.

F. Drivers

Calling the `swd_cmdq_flush()` function leads to execution of not yet executed commands from the queue (in a manner specified by the operation parameter) on the SWD bus by `swd_drv_transmit()` function that use application specific `extern` functions defined in external file (ie. `libswd_urjtag.c` or `swd_libswd_drv_openocd.c` as examples) to operate on a real hardware using drivers from existing application. LibSWD use only `swd_drv_{mosi,miso}_{8,32}` (separate for 8–bit char and 32–bit int data cast type) and `swd_drv_{mosi,miso}_trn` functions to interact with interface buffers, so it is possible to easily reuse low–level and high–level devices for communications, as they have all

Algorithm 1 Simple LibSWD example to detect and read out the target IDCODE identification register.

```
#include <libswd.h>
#include <stdio.h>
int main(){
 swd_ctx_t *swdctx;
 int res, *idcode;
 swdctx=swd_init();
 if (swdctx==NULL) return -1;
 //define driver swd_drv* functions
 //use swdctx->driver->device=...
 res=swd_dap_detect(swdctx,\
  SWD_OPERATION_EXECUTE, &idcode);
 if (res<0){
  printf("ERROR: %s\n",\
   swd_error_string(res));
  return res;
 } else printf("IDCODE: 0x%X (%s)\n",\
    *idcode, swd_bin32_string(idcode));
 swd_deinit(swdctx);
 return 0;
}
```

information necessary to perform exact actions – number of bits, payload, command type, shift direction and bus direction. It is even possible to send raw bytes on the bus (`control` command) or bitbang the bus (`bitbang` command) if necessary.

MOSI (Master Output Slave Input) and MISO (Master Input Slave Output) nomenclature was used to clearly distinguish transfer direction (from master–interface to target–slave), as opposed to ambiguous read/write statements, so after `swd_drv_mosi_trn()` master should have its buffers set to output and target inputs active.

Drivers, as most of the LibSWD functions, works on data pointers instead data copy and returns number of elements processed (bits in this case) or negative error code on failure.

It is also important to note that LibSWD can use different debug levels to produce verbose messages usually helpful in troubleshooting. LibSWD can even produce hardware bitstream debug information messages, so there is no need to use external measurement units such as oscilloscope to exactly reproduce and retrace all bus operations.

G. Summary

LibSWD project has its public website at http://libswd.sf.net. It is developed in C programming language as static and dynamic library with use of Autotools for platform independent build. The source code is Free and Open, still under development, and documented with Doxygen. The library is meant to give SWD access to external applications with minimal effort, reusing existing interface drivers, and almost no source code modifications.

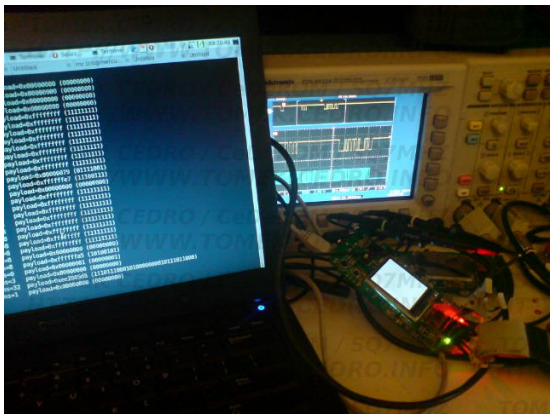

Fig. 9. LibSWD communicating with real hardware [2].

GIT repositories of LibSWD integrated into OpenOCD and UrJTAG software utilities are publicly available for testing. Project history and details are located at http://stm32primer2swd.sf.net. Feel free to contact library author for more information, suggestions and general feedback. Please remember to support Open–Source and Free–Software as you may need it one day – you will probably get as much as you give to the community for their efforts.

ACKNOWLEDGMENT

This research was possible due to personal involvement of Tomasz Bolesław CEDRO with unprecedented support from Orange Labs as part of the bigger international project conducted between Orange Labs Warsaw and Orange Labs Paris, and the Warsaw University of Technology as part of the Brain Computer Interface project conducted by Biocybernetic Aparatus Research Group, the Cybernetic Research Student Group and the Polish Interdisciplinary Neuroscience Group. Thank you for supporting Free and Open world!

REFERENCES

[1] *LibSWD – Serial Wire Debug Open Framework*, http://libswd.sf.net.
[2] *LibSWD technical details and project history*, http://stm32primer2swd.sf.net.
[3] *1149.1 – IEEE Standard Test Access Port and Boundary-Scan Architecture*, http://standards.ieee.org/findstds/standard/1149.1-1990.html.
[4] *ARM Incorporated*, http://www.arm.com.
[5] *ARM Debug Interface v5 Architecture Specification*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0031a/index.html.
[6] *ARM Information Center*, http://infocenter.arm.com.
[7] *ARM Cortex M3 Homepage*, http://www.arm.com/products/processors/cortex-m/cortex-m3.php.
[8] *MIPS Architecture Technical Specification Documents*, http://mips.com/products/product-materials/processor/mips-architecture/.
[9] *ST Mictoelectronics*, http://www.st.com/.
[10] *JTAG standard description on Wikipedia*, http://en.wikipedia.org/wiki/Jtag.
[11] *Universtal JTAG software toolkit and library*, http://urjtag.sf.net.
[12] *Open On–Chip–Debugger software utility*, http://openocd.sf.net.