



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich



## **Bachelor's Thesis Nr. 489b**

Systems Group, Department of Computer Science, ETH Zurich

A tool for debugging JTAG

by

Edem Memetov

Supervised by

Prof. Timothy Roscoe, Daniel Schwyn

February 2024 - August 2024

# **DINFK**



## Abstract

Debuggers not hosted on the target can provide users with bare-metal access to the machine. This way, one can establish a debug session even when no operating system is present, e.g., right after the power-up. The JTAG communication protocol is often used to interface the external debuggers with the target's infrastructure. On the Enzian research computer, such a connection between an ARM processor and the specialized adapter fails significantly limiting the debugging capabilities of Enzian's CPU. Here, we analyze the debugging infrastructure on Enzian and find an underlying problem with the reset signal sent to the CPU. To this end, we employ JTAG adapters that expose low-level interface to the user. We also implement a JTAG controller based on an STM32 Nucleo board that directly controls the protocol signals. We demonstrate that it can successfully drive JTAG scan chains and communicate with the rest of the processor's debug infrastructure through the Debug Access Port (DAP). Additionally, we propose an alternative method for attaching a debugger to Enzian's CPU using the onboard programming module and the Open On-Chip Debugger (OpenOCD). The thesis highlights the importance of adhering to the standard when implementing a design as this is the cause of most errors we encountered in this work.

---

## Acknowledgements

---

I would like to express my gratitude to the Systems Group for giving me the opportunity to work on this fascinating topic. I am especially thankful to my supervisor, Daniel Schwyn, for his invaluable feedback and guidance throughout the project. Finally, I want to thank my family for their unwavering support over the past six months.

---

# Contents

---

<b>Contents</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 JTAG . . . . .	3
2.1.1 Test Access Port . . . . .	3
2.1.2 Test Logic . . . . .	5
2.1.3 TAP controller . . . . .	5
2.1.4 Registers . . . . .	7
2.2 SWD . . . . .	8
2.3 ARM Debug Interface and ARM Debug Access Port . . . . .	8
2.3.1 The Debug Port . . . . .	9
2.3.2 DP registers . . . . .	11
2.3.3 AP registers . . . . .	12
2.3.4 ROM Tables and Debug register files . . . . .	13
2.4 JTAG infrastructure on Enzian . . . . .	13
2.5 DAP infrastructure on ThunderX . . . . .	15
<b>3 Implementation</b>	<b>16</b>
3.1 Fixing the JTAG issues with ThunderX . . . . .	16
3.1.1 Replicating the Problem . . . . .	16
3.1.2 Platform Cable USB II . . . . .	18
3.1.3 STM32 Nucleo board . . . . .	23
3.1.4 Solution . . . . .	29
3.1.5 Further problems with the Autodetection . . . . .	29
3.1.6 CoreSight Access Tool . . . . .	31
3.2 OpenOCD . . . . .	34
3.2.1 Motivation for OpenOCD . . . . .	34
3.2.2 Brief Overview . . . . .	36

3.2.3	Simple setup . . . . .	36
3.2.4	Interface file for the onboard adapter . . . . .	37
3.2.5	ThunderX's configuration file . . . . .	39
<b>4</b>	<b>Evaluation</b>	<b>42</b>
4.1	STM32 . . . . .	42
4.1.1	JTAG level . . . . .	42
4.1.2	Data input . . . . .	42
4.1.3	DAP level . . . . .	43
4.1.4	Clock speed . . . . .	45
4.2	OpenOCD . . . . .	47
4.2.1	OpenOCD's limitations and boot problem . . . . .	47
4.2.2	JTAG and DAP API . . . . .	48
<b>5</b>	<b>Conclusion</b>	<b>51</b>
<b>A</b>	<b>Appendix</b>	<b>54</b>
A.1	Failed autodetection log . . . . .	54
A.2	Successful autodetection log . . . . .	55
A.3	MEM-AP memory read . . . . .	56
A.4	Register read failure in OpenOCD . . . . .	57
A.5	OpenOCD to STM32 connection log . . . . .	59
	<b>Bibliography</b>	<b>60</b>

## Chapter 1

---

# Introduction

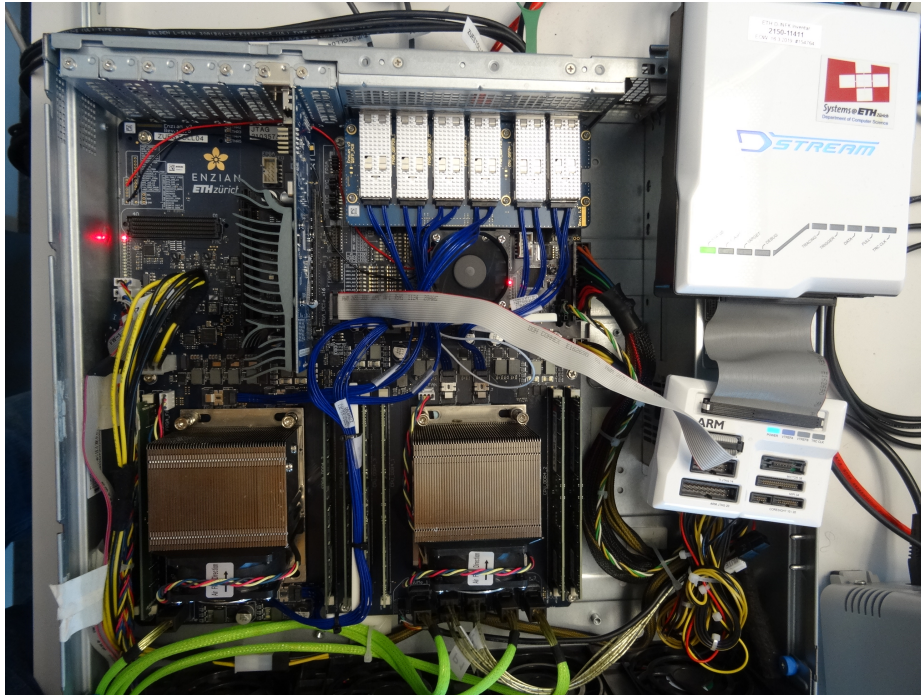
---

The JTAG standard was developed in the 1980s by the Joint Test Action Group and later codified in IEEE 1149.1 [24]. The standard introduced a serial interface and a transport protocol originally employed for the boundary scan, a mechanism for design verification and testing of printed circuit boards. However, nowadays, JTAG is not limited to this. The chip manufacturers extend the protocol and use it to expose the device to the outside world. In this way, the JTAG port serves as a bridge between the adapter and the device components. Many manufacturers also provide their own JTAG adapters and the associated software. Thus effectively building a closed ecosystem. For the CPUs, the port usually connects the adapter to the debug infrastructure on the chip, which allows the user to remotely connect to the target and launch a debug session with the processor through the specialized IDE. Sometimes, particularly in small embedded systems, such a connection goes directly to the memory bus, circumventing the microprocessor. In this scenario, the JTAG port is used to flash the program into the main memory.

The Enzian research computer [14] developed by Systems Group at ETH Zürich heavily utilizes JTAG. Various board components, including the Marvell Cavium ThunderX-1 CPU and the Xilinx Virtex Ultrascale+ FPGA, are exposed to the user through the JTAG connectors. ThunderX connects the JTAG port to the debug and tracing infrastructure compliant with the ARM Debug Interface (ADI) [13]. ARM provides its own IDE, ARM Development Studio that interacts with the ADI via various adapters. The lab has one of them called DSTREAM [4].

The mixture of different chip, board, probe, and software designers can easily introduce inconsistencies. When we plug the DSTREAM adapter into ThunderX's JTAG port (Figure 1.1) and try to initiate the bare-metal debug session from the ARM Development Studio, the connection fails. As it often happens with complex systems, it is hard to pinpoint the root of the problem: whether it lies with the CPU, with the way the processor is integrated into

the PCB, or if the problem is with how DSTREAM drives ThunderX's debug infrastructure. It is also unclear whether the fault is at the level of the JTAG protocol or higher at the ADI.



**Figure 1.1:** DSTREAM connected the CPU's JTAG header on Enzian

The main goal of our work is to locate and fix this problem so that we can establish a debug session with ThunderX. Along the way, we will present JTAG adapters and the associated software that enable users to manually drive the JTAG scan chains or interact with the CPU's debug infrastructure through the ARM Debug Interface. We will also develop a JTAG controller using an STM32 Nucleo board that provides the most basic bit-by-bit access to the JTAG port. We will thoroughly analyze JTAG's and ADI's communication models so that, in the end, we can construct the signal sequences required to initiate data transactions with the CPU's debug infrastructure through the JTAG port. We will run the sequences on our Nucleo board that show the ability of our JTAG adapter to successfully interact with the programming model of the ARM Debug Interface. We will conclude the thesis by exploring in more detail the JTAG infrastructure on Enzian and show how the onboard programming module and the open-source debugger, OpenOCD, can be used to replace (albeit with quite a few unresolved errors) the current ARM Development Studio → DSTREAM → ThunderX debugging setup on Enzian.



## Chapter 2

---

# Background

---

The chapter begins by describing the JTAG standard in enough detail to build a JTAG controller. The ADI section follows, providing the necessary background for our work with ThunderX's debug infrastructure. The chapter concludes by highlighting specific details and nuances about JTAG and ADI on Enzian and ThunderX.

## 2.1 JTAG

As mentioned in the introduction, JTAG's original application was the boundary scan (Figure 2.1). It introduces register cells that capture and control logic signals on the chip, in particular, pins on the component's boundary. The cells form a chain called a boundary-scan register so the signals can be sequentially shifted out through the serial interface called the Test Access Port (TAP). Nowadays, the JTAG infrastructure on the chip often acts as a bridge between debug adapters and the target's debugging modules. The JTAG infrastructure consists of the TAP and the test logic. The test logic is the circuitry behind the TAP. Its task is to process the signals coming in from the port. The operation of the test logic is managed by the finite state machine called the TAP controller.

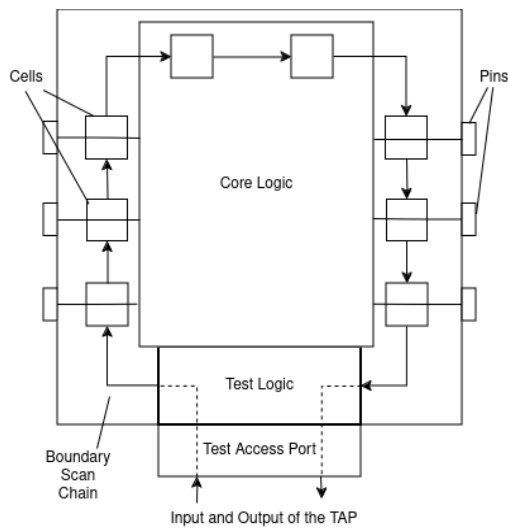
### 2.1.1 Test Access Port

The Test Access Port is a serial interface to the test logic. The TAP consists of the following connections:

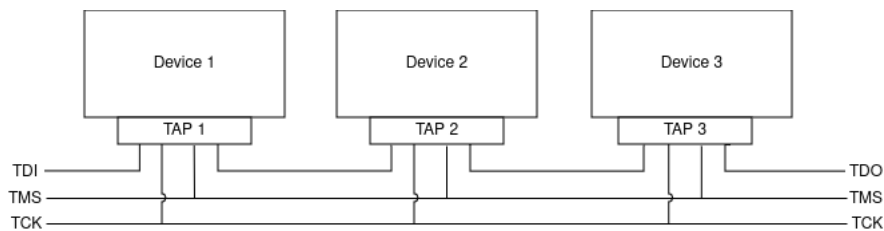
- **Test clock input (TCK)**
- **Test mode select input (TMS)** is a signal that drives the state machine of the TAP controller. The signal is sampled by the test logic on the rising edge of the clock TCK.

## 2. BACKGROUND

---



**Figure 2.1:** Typical Boundary Scan Setup



**Figure 2.2:** Daisy chain of 3 Test Access Ports

- **Test data input (TDI)** supplies the test logic (the rest of the JTAG system positioned behind the TAP) with instructions and data. The signal is sampled on the rising edge of the clock TCK.
- **Test data output (TDO)** is a signal shifted out from test logic. To avoid race conditions with the input signals, TDO is guaranteed to become stable on the falling edge of the clock TCK.
- **Test reset input (TRST\*)** is an optional active low signal. Logic 0 asynchronously resets the state machine of the TAP controller.

The TAPs of different devices can be interconnected. Usually, the TAPs share the control and clock signals, while the I/O signals are daisy-chained (i.e., the output of the predecessor is fed into the input of the successor in the sequence) (Figure 2.2). This is the setup used by Enzian (Section 2.4). Nonetheless, any other sensible arrangement is also allowed.

### 2.1.2 Test Logic

The test logic exposes to the user a set of instructions that operate on the associated shift registers. The test logic holds the current instruction in a special register called the instruction register. The JTAG standard mostly focuses on defining instructions needed for the boundary scan, but manufacturers may extend the functionality of the test logic by introducing new commands.

As mention earlier, a synchronous finite state machine called the TAP controller controls the test logic. The transitions happen according to the TMS value during the rising edge of the clock TCK. Based on the state of the TAP controller, the test logic interprets the incoming TDI signal as either a new instruction or as input data. For the former, the input is then shifted into the instruction register. While for the latter, the input is loaded into the test data register associated with the current instruction. In either case, the TDO signal latches onto the opposite end of the register forming a TDI-shift register-TDO path. The shift process is sequential and happens one bit per clock cycle starting with the least significant bit. TDI and TDO always share the same register. We will go through parts of the test logic in more detail.

### 2.1.3 TAP controller

The finite state machine (Figure 2.3) contains distinctly visible state sequences that correspond to instruction (red) and data (green) processing. They both operate in the same way by selecting the correct shift register, loading the previous value into the register, shifting in new data or shifting out the result, and passing the contents to the test logic. The rest of the state machine implements the reset and idling functionality. In particular:

- **Test-Logic-Reset** is the initial state of the TAP controller after the power-up. It brings the system into a known state by loading the binary code of IDCODE (if implemented) or BYPASS into the instruction registers. The state machine is designed so that the Test-Logic-Reset is reachable from any state if TMS is held high for five consecutive clock periods. Alternatively, low TRST forces the state machine into Test-Logic-Reset.
- **Run-Test/Idle** is a looping state that the state machine can enter between the scan operations. It is used by instructions such as a self-test RUNBIST that requires additional clock cycles to finish its execution.
- **Select-IR and Select-DR** connect the TDI and TDO lines to the instruction or test data register, respectively. The test data register is chosen based on the current instruction.
- **Capture-DR** latches the test data register onto the TDI-TDO scan chain. **Capture-IR** latches the instruction register onto the scan chain. The latched value of the instruction register is always the same. According

## 2. BACKGROUND

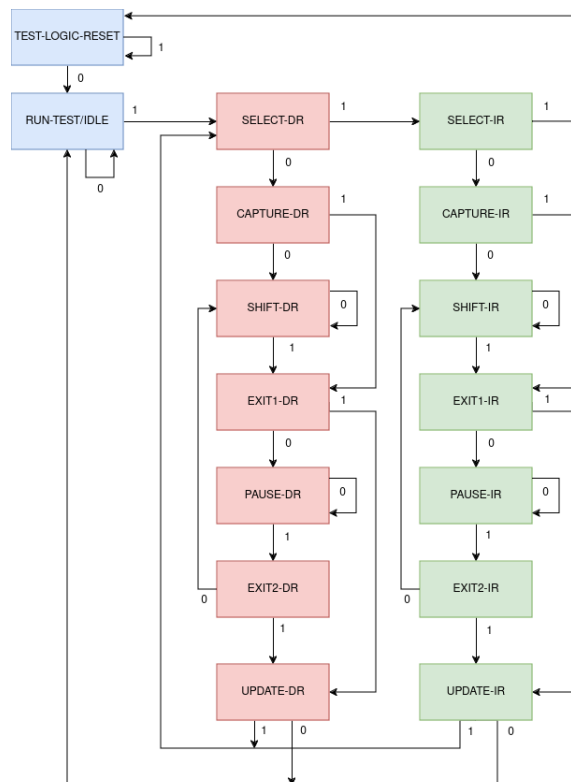


Figure 2.3: JTAG state machine

to the standard, the two least significant bits should be 01. The rest is implementation-specific.

- **Shift-IR and Shift-DR.** Each pass through these states shifts 1 bit along the TDI-shift register-TDO path on the rising edge of the clock. The last bit is shifted during the transition to Update-IR/Update-DR.
- **Update-IR and Update-DR** pass the data in the instruction or test data register to the rest of the test logic for processing. Note that during the shift phase, the register is usually not visible to the rest of the system as it constantly changes.
- **Exit1-IR/Pause-IR/Exit2-IR** and their DR counterparts indicate the end of the shifting process. The TAP controller may temporarily halt the procedure or completely terminate it. Two separate exit states allow us to implement a choice between resuming and finishing the shifting after leaving Pause-IR while also providing an immediate path to the Update state, circumventing the Pause state.

### 2.1.4 Registers

The instruction register contains the binary code of the current instruction. Except for BYPASS, these values are implementation-specific and often poorly documented. The size of the register is also not set by the standard. When testing devices in the lab, we encountered devices with 4-bit, 8-bit, and 12-bit instruction registers.

As mentioned earlier, each JTAG instruction is associated with the specific test data register. The standardized instructions are called public. Among them, we have extensively used IDCODE and BYPASS. We ignored others since they belong to boundary scan procedures. Manufacturers can extend the instruction set with private instructions. For instance, APACC and DPACC implemented in the JTAG-Debug Port of the ARM CoreSight architecture provide access to the debug logic (Section 2.3.1).

**BYPASS** instruction selects the test data register of length 1 with an initial value of 0. Data shifted through the BYPASS register does not affect the rest of the test logic. The intended use case of BYPASS is to minimize to 1 clock cycle the path through unused devices on the daisy chain.

BYPASS is the only instruction with the binary code set by the standard: 11...1.<sup>1</sup> Additionally, the designs often default the behavior of the test logic to BYPASS when an undefined binary code is loaded into the instruction register. The uniform binary code value of the BYPASS register and its fixed size make it possible to count the devices on the scan chain. The algorithm is described in detail in Reverse Engineering paper by Felix Domke [18]. It starts with navigating each TAP controller on the chain into the BYPASS mode by shifting in a long sequence of 1s into the instruction registers (If the sequence contains hundreds of bits, we can be sure that all controllers are in the BYPASS mode even for very long scan chains since the usual instruction register width is orders of magnitude smaller). In the next step, the algorithm inputs 11...1011...1 into the data path. Since each BYPASS test data register contains 1 bit, the number of devices on the chain is equal to the offset of bit 0 on the TDO line compared to TDI. DSTREAM uses a similar procedure to enumerate the scan chain (Section 3.1.5).

**IDCODE** uniquely identifies the device. Its 32-bit test data register contains manufacturer, part number, and version fields. IDCODE is loaded by default into the instruction register when the state machine enters Test-Logic-Reset. This makes it trivial to determine all the devices on the board by immediately shifting out the contents of all data registers after the reset.

IDCODE is an optional instruction. If it is not implemented, the test logic defaults to BYPASS. To differentiate between these 2 cases, the least significant

---

<sup>1</sup>Although earlier versions of the standard also assigned 00...0 to EXTEST.

bit of the identification register is always set to 1. (Recall that the BYPASS always starts as 0)

### 2.2 SWD

Serial Wire Debug protocol (SWD) [37] was developed by ARM as an alternative to JTAG for communication with the debug infrastructure of ARM-based systems. The authors addressed the shortcomings of the JTAG protocol caused by its original design for PCB testing. The main difference is the lower pin count: SWD only uses a clock signal, SWCLK, and a bidirectional data signal, SWDIO. SWD's communication model is packet-based.

### 2.3 ARM Debug Interface and ARM Debug Access Port

CoreSight technology [3] developed by ARM defines a standard for debug and trace infrastructure of ARM-based processors. The ARM Debug Interface (ADI) [8] is a crucial component of the CoreSight Architecture. It connects external debuggers to the debug and tracing modules on ARM-based chips. In our work, we focused solely on ADIV5, as it is the version of the interface implemented in ThunderX [13]. The physical implementation of the ADI is called the Debug Access Port (DAP). DAP is divided into the Debug Port (DP) and the Access Port (AP).

**The Debug Port** is responsible for establishing a physical connection with the debug adapters. For this purpose, it can utilize the JTAG transport protocol. This version of the Debug Port is called JTAG-DP. JTAG-DP must integrate the Test Access Port, the TAP controller, and the rest of the test logic into the module. Instead of JTAG, the DP also has the option to use the SWD protocol (SWD-DP). Or even combine JTAG and SWD in a single module (SWJ-DP). In that case, the dual DP module can switch between two modes of operation via a 16-bit signal sequence on the TMS (for the JTAG→SWD switch) or the SWDIO (SWD → JTAG) lines. (Note that on such dual purpose ports, TMS and SWDIO share the same pin.) The DP forwards the incoming input data or read requests to the second half of the system, the Access Port, which programs the debug peripherals. The DP provides the user with feedback about the success of the transactions in the form of acknowledgment fields. Additionally, it implements performance optimizations such as a transaction counter that allows us to trigger multiple sequential memory accesses with a single I/O request. The DP is also responsible for power management and the reset logic of the DAP.

**Access Ports** connect the DAP to various onboard peripherals. The version of the Access Port called JTAG-AP is used for legacy devices that are part of JTAG scan chains. However, the most common are Memory Access Ports

(MEM-AP), which use memory map as a programming model for external debuggers. MEM-APs are further subdivided by the type of memory bus. The version that uses AHB (AHB-AP) is designed “to directly connect to an AHB-based memory system” [1]; thus, having access to the main memory bypassing the CPU. This type of access port is used, for example, on the STM32 Nucleo boards. On the other hand, APB-AP are generally designed to connect to an APB bus for a dedicated debugging and tracing memory space that is separate from the main memory but still accessible from the CPU. In that case, debug peripherals and system topology information are mapped onto 4K pages in the address space of APB-AP. Pages associated with debug peripherals are called debug register files. At the same time, the topology information is stored in ROM tables.

According to the manual, a DAP must include a single Debug Port that can multiplex communication to multiple Access Ports. We can mix Access Ports of different types. A single JTAG-AP can access up to eight JTAG daisy chains, while multiple debug peripherals may be mapped into the memory space of a single MEM-AP.

The manufacturers may extend the standard functionality of a MEM-AP by allowing variable memory size accesses (Large Data Extension) or variable addressing length (Large Physical Address Extension). Another optional extension called packed transfer can combine I/O to multiple bytes at a fixed offset into a single-word operation. Note that this concept is orthogonal to the transaction counter.

AP and DP implement the above functionality through the series of 32-bit registers gathered together in register banks. Additionally, JTAG-DP extends the JTAG instruction set with two custom commands, DPACC and APACC, that allow the external debugger to interact with the AP and DP registers. The general overview of the Debug Access Port is shown in Figure 2.4.

### 2.3.1 The Debug Port

In this section, we describe JTAG-DP, the DP version utilized by ThunderX [13]. Apart from implementing standard BYPASS and IDCODE instructions, JTAG-DP extends the instruction set by introducing ABORT, DPACC, and APACC commands. The binary codes of JTAG-DP instruction are standardized, although the width of the instruction register can be either 4 or 8 bits.

**ABORT** stops any AP operation, such as reading and writing to the memory space.

**DPACC** and **APACC** are used to access the registers in the DP and AP register banks, respectively. The test data registers of DPACC and APACC follow the same structure: 35-bit registers with bits [34:3] holding the transfer

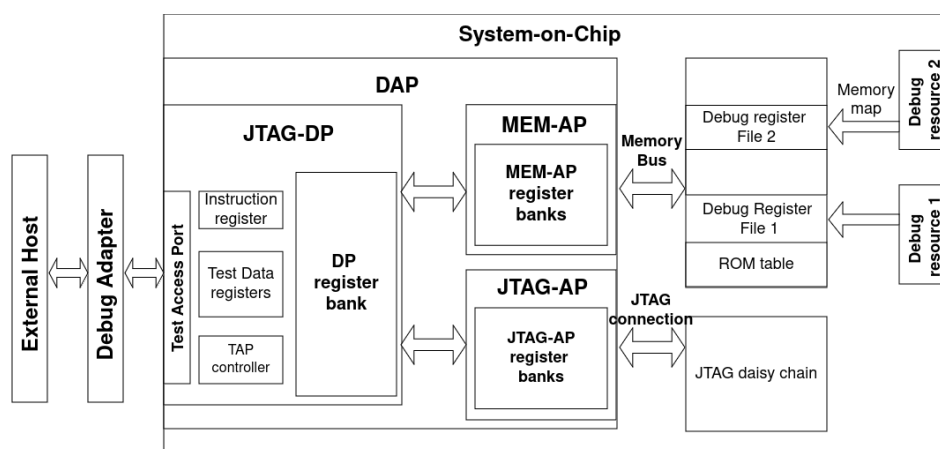


Figure 2.4: Debug Access Port

data and [2:0] control bits. The exact meaning of these fields depends on the execution context, particularly on the current state of the TAP controller and on the previous JTAG instruction.

If the current instruction is DPACC, then the control bits [2:0] of the DPACC test data register latched onto the scan chain in CAPTURE-DR hold the response sequence of the previous transaction (either APACC or DPACC). The bits indicate whether the transaction has finished (OK/FAULT response has value 0b010) or not (WAIT response has value 0b001). Note that the logic does not differentiate between successful and unsuccessful transactions. This information is stored separately in the control register CTRL/STAT in the DP register bank. We must initiate a separate DPACC read to get its contents. In particular, CTRL/STAT contains a sticky error flag STICKYERR that indicates a memory access error. Since the bit is sticky, it persists throughout the rest of the memory access transactions. Until it is cleared, all further memory accesses are discarded by the test logic in JTAG-DP. Additionally, if the current DPACC was directly preceded by a successful register read (either APACC or DPACC), then the data bits [34:3] hold the transaction output. Otherwise, the contents of the transfer field are implementation-defined.

When the shifted input from the scan chain is passed to the test logic in the UPDATE-DR state, DPACC holds the information about the subsequent DP register access. The least significant control bit [0] in DPACC indicates whether it is a read (0b1) or a write operation (0b0). Bits [2:1] select the target register in the DP register bank. In the case of a write operation, the data field contains the value we want to write. Note that in the initial version of the Debug Port (DPv0), the register bank consisted only of 3 registers, so 2 bits were enough to address them. Newer versions have 5 (DPv1) and 8 (DPv2) registers in the bank. In that case, field [2:1] forms only part of the address; the second half comes from the DPBANKSEL field in the SELECT register of



the DP register bank. To bootstrap, the SELECT register is addressed by just the [2:1] field with values 10. So that the general DP register access becomes the following sequence of JTAG instructions: 1. DPACC that writes to the SELECT register and updates the DPBANKSEL field. 2. DPACC that writes to the intended register. The addressing ([2:1] and DPBANKSEL values for each register can be found in Table B3-5 of the ADI specification [8].

APACC operates the same way as DPACC, except that we have to specify one of the APs connected to the Debug Port (as there can be multiple). Also, the AP registers are split into multiple register banks, each of which can hold up to 4 registers. To facilitate these changes, the APSEL and APBANKSEL fields of the SELECT register pick the AP and the register bank, respectively. Meanwhile, the [2:1] bits of the APACC form an offset into the selected register bank. Summing it up, the general AP register access becomes the following sequence of JTAG instructions: 1. DPACC that writes to the SELECT register and updates the APSEL and APBANKSEL fields. 2. APACC that writes to the intended register. The exact address for each register can be found in Table C2-6 of the ADI specification [8].

Note that the standard does not set the time it takes to process the access instruction, starting from passing the register contents to the rest of the test logic in CAPTURE-DR and loading the output in UPDATE-DR. Thus, on page B3-100, the specification [8] suggests that the debuggers navigate the state machine between CAPTURE-DR and UPDATE-DR through the looping IDLE state so that the debugger can adjust to the implementation-dependent waiting times.

### 2.3.2 DP registers

As we mentioned in the previous section, each version of the Debug Port introduced new registers. We will concentrate on DPv1 because this version is implemented by ThunderX's DAP. (We determine that later in Section 3.1.6.)

- **The Debug Port Identification Register, DPIDR.** Apart from the manufacturer identity and the device's part number, DPIDR contains fields indicating the version of the Debug Port (value 1 corresponds to DPv1) and supported extensions such as transaction counter (MIN field). The DPIDR is organized such that its fields overlap with their counterparts in the JTAG's IDCODE register. Additional fields of DPIDR (such as MIN) that are not present in IDCODE are placed in the bits occupying the second half of the Partnumber field in IDCODE. Thus, DPIDR and IDCODE can be the same or vary only slightly. As we determine later (Section 3.7), this is the case on ThunderX.
- **Control/Status register, CTRL/STAT,** controls the operation of the

Debug Port. Apart from response fields such as the Sticky error flag mentioned in Section 2.3.1, CTRL/STAT sets the power mode of the Debug Port, controls the system reset, and selects modes of operations such as transaction counter mode TRNCNT. If the value of TRNCNT is  $n$ , a single AP memory operation initiates  $(n+1)$  sequential memory accesses starting from the address contained in the TAR register of the Access Port. Note that the standard does not specify the default values for some mode fields, such as transaction counter TRNCNT and the transfer mode TRNMODE. They have to be set manually at the start of the operation. Also, the debuggers have to configure the power domain of the DAP right at the start of the operation (Section B2.3 of the ADI specification [8]).

- **AP Select Register, SELECT.** As discussed in Section 2.3.1, despite its name, SELECT also takes part in the DP register addressing for newer versions of the Debug Port. In total, SELECT consists of three fields: APSEL selects one of the Access Ports associated with the Debug Port, APBANKSEL selects the register bank of the target for the next APACC transaction, and DPBANKSEL builds part of the offset into the DP register bank.

We omit the discussion of other DPv1 registers that are irrelevant for JTAG-DP (DLPIDR) or that we never used (RDBUFF).

### 2.3.3 AP registers

We list registers implemented by MEM-AP. ThunderX does not have JTAG Access Ports.

- **Identification register, IDR.** In addition to the designer identification, it also indicates the class of the access port (JTAG-AP or MEM-AP). The type field further divides MEM-APs according to their memory bus.
- **Debug Base Address Register, BASE.** The least significant bit indicates whether the MEM-AP is connected to debug components. If so, the BASE register points to the ROM table. If only one debug register is mapped into the memory space of the MEM-AP, the ROM table is not mandatory, and the BASE register can hold the address of the debug register file.
- **Configuration register, CFG** indicates whether the Access Port supports optional Large Data and Large Physical Address Extensions.
- **Control/Status Word register, CSW,** controls the operation of the Memory Access Port. For example, it can disable the AP, change the data access size (if the Large Data Extension is implemented), or deny access to the debug software. The reset value of some register fields

is unknown. Similarly to CTRL/STAT, the debugger must manually configure the CSW register before initiating memory operations.

- **Transfer Address Register, TAR**, contains the address of the AP memory access. To facilitate the Large Physical Address Extension, the neighboring register that follows TAR in the register bank is reserved for storing the 32 most significant bits of the 64-bit address. If the Large Physical Address Extension is not present, the contents of that register are ignored.
- **Data Read/Write register, DRW**. APACC's access to this register initiates writes/reads to the location specified in TAR.

Again, we do not include the discussion of other MEM-AP registers that we do not use in our work. Figure 2.5 shows a complete register programming model in the Debug Access Port, similar to how it is depicted on page 53 of the CSAT manual [9].

### 2.3.4 ROM Tables and Debug register files

The debugger starts to reconstruct the topology of the debug infrastructure by accessing the ROM table. The ROM table stores an array of addresses that point to all debug register files in the memory space. To identify the corresponding devices, there are Component and Peripheral Identification Registers (CIDR and PIDR) at the end of each debug register file (and the ROM table). The Class field of the CIDR stores the type of the file. For example, 0x1 is allocated for the ROM table, while 0x9 indicates the Debug component of the CoreSight architecture. The fields in PIDRs, similarly to the IDCODE register in JTAG, contain information about the component's designer and its part number. The debug register files of CoreSight components must contain additional registers that describe the part in more detail (Section B2.3 in the CoreSight architecture specification [7]). For example, the (MAJOR, MINOR) field pair of the Device Type Identification Register (DEVTYPE) might be (0x5, 0x1) == (Debug Logic, Processor Core) or (0x4, 0x1) == (Debug Control, Trigger Matrix).

## 2.4 JTAG infrastructure on Enzian

The JTAG daisy chain on Enzian contains four devices: Marvell Cavium ThunderX-1 CPU, Xilinx Virtex Ultrascale+ FPGA, the Baseboard Management Controller, and the clock and data recovery device [20]. A pin header is positioned between each device and the chain. A device is visible to the rest of the chain (i.e., it is in the "connected mode") when the jumpers are configured to forward every JTAG signal to the TAP of the device. If, instead, the jumpers directly join the TDI and TDO lines while leaving other pins

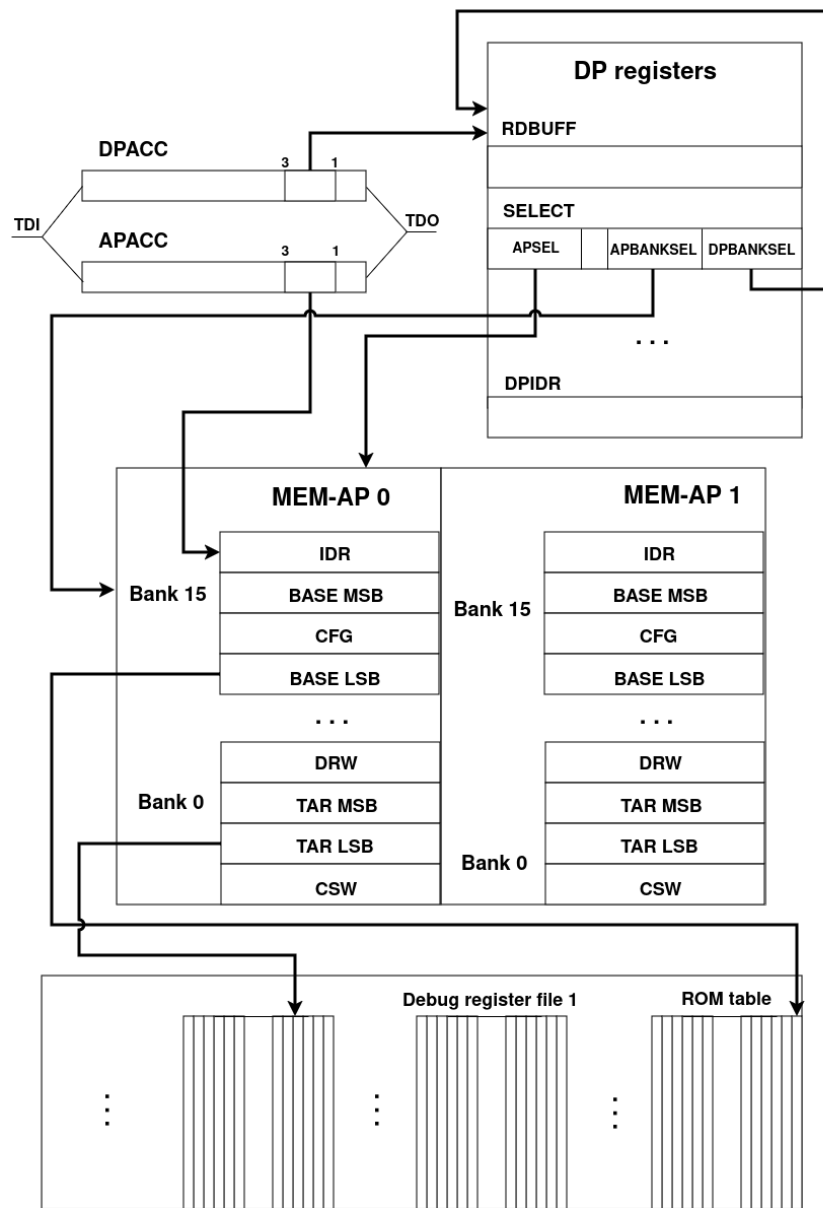


Figure 2.5: DAP programming model

disconnected, the device operates in “bypass mode,” effectively rendering the device absent from the chain. The chain itself is driven by the Surface-mount Programming Module JTAG-SMT3-NC. The module is accessible from the outside using a USB-B port.

Apart from this, all the devices are exposed to external JTAG controllers via native JTAG headers. For example, the CPU header is compatible with the 14-pin ARM adapter. (the pinout is depicted in Section 3.11 of the ARM

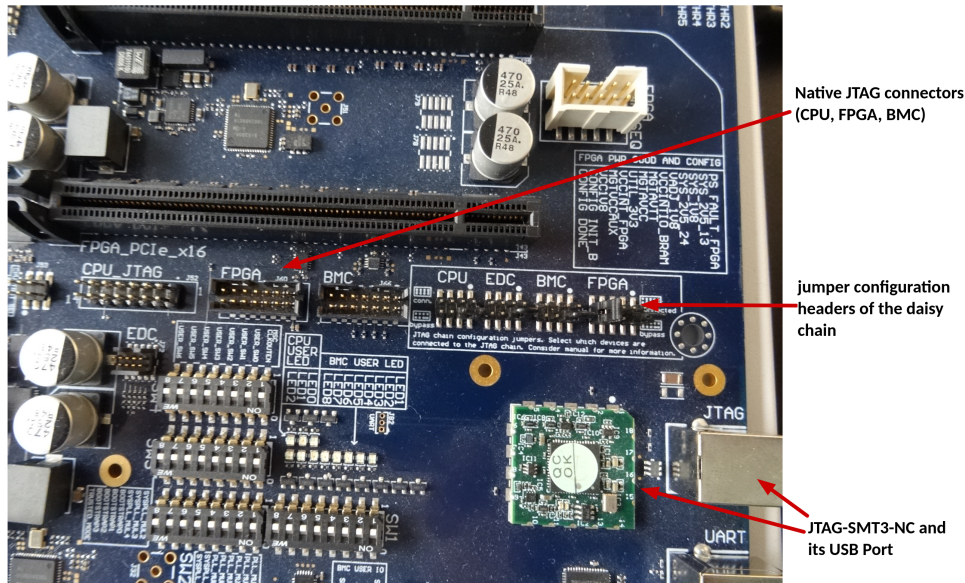


Figure 2.6: JTAG infrastructure on the Enzian board

DSTREAM System and Interface Design Reference [2].) Note that with this approach, the jumper configuration of the device on the daisy chain must be configured in a bypass position to prevent interference with the adapter working through the JTAG header.

Figure 2.6 shows the relevant part of Enzian's PCB.

## 2.5 DAP infrastructure on ThunderX

ThunderX features an ARM ADiv5-compliant debug infrastructure. The DAP consists of a JTAG-Debug Port and two Access Ports. One is a standard APB-AP that connects the DAP to the 32-bit debug address space. The space contains the debug register files of the cores, their cross-trigger interfaces, and their performance monitoring units. APB-AP shares the memory bus with ThunderX's cores. This way, the chip's debug infrastructure can be accessed internally and externally. Following the standard, the ROM table describes the topology. The second Access Port (CVM-AP) is Cavium-specific. It allows external debuggers to access the system registers of each core.

# Implementation

---

The first half of this chapter details how we went about fixing the JTAG access to ThunderX on Enzian. The section also includes our implementation of the Nucleo board-based JTAG adapter. In the second half, we provide a comprehensive guide to configuring and establishing a debug session with ThunderX using the Open On-Chip Debugger (OpenOCD).

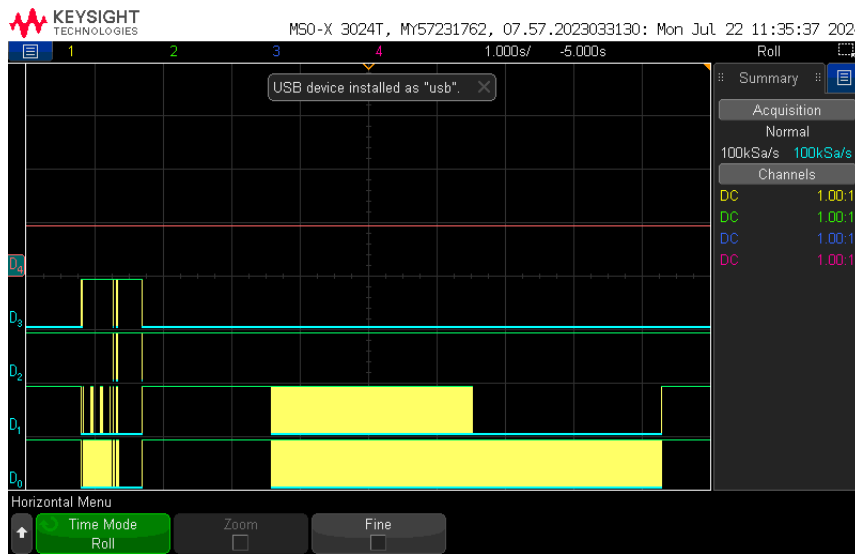
### 3.1 Fixing the JTAG issues with ThunderX

#### 3.1.1 Replicating the Problem

As the first step, we replicated the problems that arise when a debugger connects to ThunderX and gathered the associated logs. For this, we used a debugging device for ARM-based targets called DSTREAM [4]. The central part of the device is the debug adapter itself, the DSTREAM unit. It supports direct (USB) and remote (TCP) connections from the host computer. A separate DSTREAM probe connects the unit to the target. It features various connector standards, among which there are three different JTAG interfaces. We ensured that the CPU's configuration jumpers on the Enzian's daisy chain were set into the bypass position and then connected an ARM JTAG 14 cable from the DSTREAM probe to the CPU's JTAG header. Furthermore, we captured the signals on the oscilloscope. For this, we used the fact that ARM JTAG 14 and the 20-pin ARM Standard JTAG are electrically connected on the DSTREAM probe. So, we plugged the logic probes of the oscilloscope into the 20-pin connector.

We tried connecting to the processor from ARM Development Studio (ARM DS). We followed the steps described in the "Configuring a connection to a bare-metal hardware target" section in the ARM Development Studio Getting Started Guide [10]. We picked the "ThunderX-r2 AP0" option during the target selection and the first core of ThunderX as our debug target ("Debug

### 3.1. Fixing the JTAG issues with ThunderX



**Figure 3.1:** Autodetection. The signals are D1:TCK, D1:TDI, D2:TMS, D3:TRST, D4:TDO (high-lighted in red)

ThunderX-r2.00”). Finally, we had to configure the DSTREAM connection by selecting JTAG as the transport protocol, setting the clock speed of the DSTREAM unit, and passing its IP address. This setup results in the “No connection to the target error” in ARM DS.

We also tested the autodetection (Section 12.7 in ARM development Studi Getting Started Guide), which explores the topology of a debug target unknown to ARM DS and automatically builds the associated platform configuration file. To start the procedure, we must follow the same steps as for the hardware debug connection, except that instead of the target selection, we use the “Add a new platform. . .” option.

From the start, DSTREAM failed the autodetection: According to the output log on the PCE console (see Section A.1), it failed to enumerate the scan chain both using JTAG and SWD protocols. It then tried to access the APIDR register to identify the Access Ports on the DAP, but it was unsuccessful.

Furthermore, the TDO readings on the oscilloscope remained unchanged throughout the test (Figure 3.1). The fact that ThunderX’s DAP remained completely unresponsive leads us to believe there might be a problem with the JTAG infrastructure on the Debug Port. To test this, we decided to replace DSTREAM with the Platform Cable USB II, a low-level JTAG adapter that allowed us to drive the state machine manually.

#### 3.1.2 Platform Cable USB II

The Platform Cable USB II [38] is a device primarily utilized for AMD FPGA programming using SPI or JTAG protocols. It exposes parts of the JTAG interface that are sufficient for basic JTAG debugging.

We can interact with Platform Cable USB II through the command line interface of the Xilinx System Debugger (XSDB). The binary of the command line tool is called `xsdb`. It is located in the binary directory of any Vivado or Vitis installation. To make the devices accessible from XSDB, we must connect them to the Xilinx Hardware server. Luckily, the Platforms Cables in the lab were already visible to the server, and we could skip their configuration. We now go through the relevant JTAG commands in `xsdb`. It is taken from the XSDB reference manual [39]. We can connect to the Xilinx hardware server and list all JTAG adapters visible to the debugger with `connect` and `jtag targets` instructions. To select one of the adapters, we specify its index in the list: `jtag targets <index>`.

Communication with the target controller proceeds in sequences of JTAG operations. `jtag sequence` creates a new empty sequence and outputs its name. We can add operations to the sequence by issuing JTAG commands that cover most of the standard functionality. The sequence's name must precede every JTAG operation that we want to add. The names are usually hard to type, so we can alias them using TCL syntax: `set <alias_name> [jtag sequence]`. We substitute the alias by prepending the '\$' symbol: `$(alias_name)`. Throughout the rest of the section, we refer to the output of `jtag sequence` as `<sequence name>`.

Valid JTAG commands that we can bind to the sequence include:

- `<sequence name> state <name of the state> [count]` controls the TMS signal by navigating the TAP controller into the state passed in the parameter.
- `<sequence name> irshift/drshift [options] [bits [data]]` brings the state machine into the SHIFT-IR / SHIFT-DR state. The input specified as a command parameter is shifted into the instruction/data register. The input is represented either as constant logic 0 or 1 (`-tdi` option) or as a bit sequence of an arbitrary length (`-bits` option) and arbitrary contents in a binary (`-binary`), integer (`-integer`) or hexadecimal (`-hex`) form. In the binary form, the leftmost bit is the first bit shifted into the test data register. As for binary to hexadecimal conversion, the manual states that "the first bit shifted out is the least significant bit of the first byte of the in the string" [39], i.e., we divide the binary sequence into bytes, then read each byte from right to left and transform it into a hexadecimal number. E.g., `1101'0100'0101'1011` → `2B'DA`. The option `-state <new state>` sets the next state after the shift command



finishes the execution. The exact names of the states are listed in the manual.

- `<sequence name> delay [usec]` halts the execution for a usec microseconds
- `<sequence name> get_pin <pin>` and `sequence set_pin <pin> <value>` manually drive the JTAG signals. The manual states that support for these commands depends on the JTAG adapter. For example, the JTAG port on the Platform Cable does not have the TRST pin. However, the tool forcibly closes the connection channel even if we try to manipulate mandatory signals such as TDI or TDO.
- `<sequence name> run` executes the commands bound to the sequence.
- `<sequence name> clear` deletes all JTAG operations from the sequence.

The output is displayed in XSDB by supplying `irshift/drshift` instructions with the `-capture` option. By default, it is in the hexadecimal form. We can change it with `-binary` or `-integer` options for `sequence run`. XSDB also uses the same binary to hexadecimal conversion in the case of the output. In the binary form, the leftmost bit of the output string corresponds to the first bit that comes out of the TAP.

The Platform Cable USB II can operate at different predefined frequencies selected by `jtag frequency` command. `jtag frequency -list` prints the frequency range supported by the adapter. For Platform Cable USB II, it goes from 125KHz to 30MHZ.

#### Tests

In order to verify our understanding of the setup, we analyzed a known working JTAG scan chain of another computer, from now on referred to as `schibenstoll02`. `Schibenstoll02` has Mercury XU5 system-on-chip [19] integrated on the otherwise unpopulated Enizan's PCB. The JTAG scan chain on SoC includes the DAP and the Zynq Ultrascale+ TAP. The scan chain is connected to the BMC's JTAG line. We ensured that the BMC's configuration jumpers for the daisy chain were in the bypass mode and connected the Platform Cable to the BMC's JTAG pin header using the JTAG flying wire adapter (Figure 3.2). This allowed us to capture the signals on the oscilloscope by attaching the analog probes to the metallic hoops on the cable.

We started with reading the identification registers of the devices on the scan chain (Listing 3.1). As mentioned in Section 2.1.4, it is the default operation upon the reset. Thus, after we had navigated the state machine into the RESET state (line 3), we could immediately start shifting the data register without supplying any instructions (line 4). We decided to shift

### 3. IMPLEMENTATION

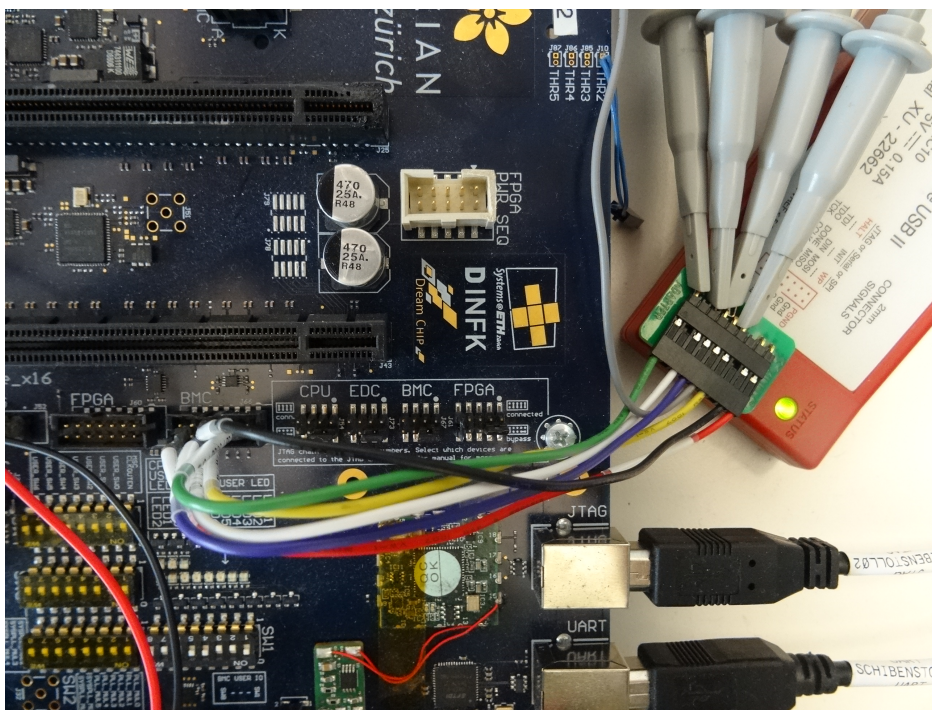


Figure 3.2: Platform Cable USB II connected to schibenstoll02

more than the necessary 64 bits (which corresponds to the size of the ID-CODE test data register) to verify that the scan chain contains only two devices. This is indeed confirmed by the fact that the output remains constant 1 after 64 bits. If we reverse the first 32 bits of the output, we get the IDCODE of the last device on the chain:  $7704a05b \rightarrow$  (XSDB conversion)  $1110'1110'0010'0000'0000'01011'1011'0101 \rightarrow 1010'1101'11010'0000'0000'0100'0111'0111$ . The designer [11:1] and part number fields [27:12] identify it as an ARM Debug Access Port [11]. Figure 3.3 shows the start of the procedure captured on the oscilloscope. Four signals are displayed from top to bottom: TDO, TDI, TMS, and TCK. For example, in the fourth horizontal division of the TMS, we can see the bit sequence 0100 that navigates the state machine from TEST-LOGIC-RESET into SHIFT-DR. After this, the TDO line contains the two least significant bytes of the identification code:  $1110'1110'0010'0000$ . Note that the Platform Cable generates the clock signal in batches of 4.

Next, we determined the total instruction register length of the scan chain (Listing 3.2). We employed the algorithm described in Section 2.1.4. It leverages the general idea for the scan line width estimation. The algorithm shifts  $11\dots1011\dots1$  into the instruction register and counts the number of clock cycles for the bit 0 to appear on the output line. The delay corresponds to the total length of the registers on the scan line. The output string starts with the contents of the instruction register loaded in the CAPTURE-IR

### 3.1. Fixing the JTAG issues with ThunderX

```
1 xsdb% set seq [jtag sequence]
2 ::jtagseq#0
3 xsdb% $seq state RESET
4 xsdb% $seq drshift -capture -tdi 1 128
5 xsdb% $seq run
6 7704a05b93107204fffffffffffffff
```

Listing 3.1: IDCODE read

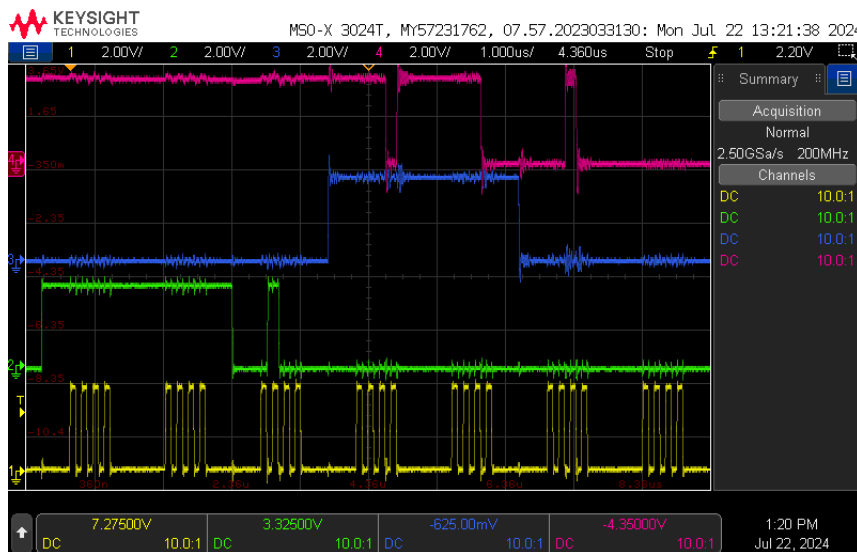


Figure 3.3: IDCODE read Platform Cable, The signals are from top to bottom: TDO, TDI, TMS, TCK

state, followed by the input string. Bit 0 is at positions 17 and 33 of the input and output strings, respectively. Thus, the total instruction register length of the scan chain is 16. Note that although the 2 LSBs loaded into the instruction register in CAPTURE-IR must be 01 (see Section 2.1.3), the rest is implementation specific. So, in general, we cannot determine the register length by recognizing some specific bit sequence on the output line. However, in this case, the rule can help us to find the individual lengths. The DAP is the last device on the chain, so its instruction register appears first on the output line. Its length can only be 4 or 8-bit. If it were an 8-bit version of the JTAG-DP, the second half of the output, 07 == 01110000 (XSDB's hexadecimal to binary conversion), would correspond to Zynq Ultrascale's TAP, which clearly cannot be the case. Thus, the ARM DAP and Zynq Ultrascale's TAP have 4-bit and 12-bit instruction registers, respectively.

Now, we could test the functionality of the instruction register (Listing 3.3). Since Zynq UltraScale's manual does not disclose the binary codes of the instructions, the only one that we could test was BYPASS (0xFFF). On the

### 3. IMPLEMENTATION

---

```
1 xsdb% set seq [jtag sequence]
2 ::jtagseq#1
3 xsdb% $seq state RESET
4 xsdb% $seq irshift -capture -hex 64 ffffffff
5 xsdb% $seq run
6 5107ffffffffff
```

---

**Listing 3.2:** Instruction length estimation

other hand, the binary codes of the ARM DAP are well documented. We picked IDCODE 0b1110. Since ARM DAP is the last device on the chain, its binary code has to be shifted in first (Line 3). With the instructions selected, we navigated to the test data register and shifted in a long sequence of 1s (Line 4). The result is 7704a05bfeffffff, which corresponds to the identification number of the ARM DAP followed by input signal delayed by one bit 0. This bit is loaded into the BYPASS register in CAPTURE-DR.

```
1 xsdb% set seq [jtag sequence]
2 ::jtagseq#1
3 xsdb% $seq irshift -state IDLE -hex 16 0xfeff
4 xsdb% $seq drshift -capture -tdi 1 64
5 xsdb% $seq run
6 7704a05bfeffffff
```

---

**Listing 3.3:** Manual selection of the instructions

As the last example, we want to showcase how to embed the XSDB commands in TCL scripts, which can greatly increase the complexity of the tests. This way, we were able to determine the binary code of the IDCODE instruction in Zynq Ultrascale's TAP controller. The central part of the script, the for loop, is shown in the Listing 3.4. It stores the first 64 output bits for every possible instruction register value of the TAP controller while keeping the IDCODE instruction for the ARM DAP. We can deduce the binary code of IDCODE by searching the output file (Listing 3.5) for the identification number we determined earlier. Interestingly, four different binary codes trigger the IDCODE command. Moreover, this script can be easily modified to implement the length-scan test for the reverse engineering of JTAG instructions [18].

Convinced that we can successfully drive the Platform Cable, we attached it to the CPU's JTAG header on Enzian. We started with the same simple tests we had previously run on schibenstoll02: shifting out the contents of the instruction register, its width estimation, IDCODE, and BYPASS commands. Unfortunately, none of them worked. The TDO line always remained constant.

We wondered whether the problem was with the Platform Cable itself. In

```
1 for {set i 0} {$i < 65536} {incr i} {
2     puts $fp "-----\nrunning sequence $i"
3     set irsequence "0111[hexToBinary [format "%03x" $i]]"
4     puts $fp "shifting irsequence: $irsequence"
5     puts $irsequence
6     set seq [jtag sequence]
7     $seq state RESET
8     $seq irshift -capture -state IDLE -bits 16 $irsequence
9     $seq drshift -capture -tdi 1 64
10    set ans [$seq run]
11    puts $fp $ans
12    $seq clear
13 }
```

---

**Listing 3.4:** Instruction register space enumeration

```
1 running sequence 0
2 shifting irsequence: 0111000000000000
3 1105 7704a05bfcffffff
4 . . .
5 running sequence 612
6 shifting irsequence: 0111001001100100
7 1105 7704a05b93107204
```

---

**Listing 3.5:** The output file of the script

particular, we did not know if the missing TRST line was essential or whether the unusual way the adapter drove the clock signal influenced the working of the DAP, as this behavior is not mentioned in the JTAG standard. Additionally, we did not completely control the execution of Platform Cable. For example, we could not control the exact pathing of the TAP controller through the state machine. If we wanted to see the exact state transitions, we had to hard code them in our scripts with the `state` command or inspect the TMS signal on the oscilloscope. It was cumbersome to capture the signal sequences on the oscilloscope, especially for long sequences, due to the high operating frequency of the adapter. To mitigate these issues, we developed our own JTAG adapter, which gives us complete control over the execution environment. For this task, we selected an STM32 Nucleo board.

#### 3.1.3 STM32 Nucleo board

The STM32 Nucleo are development boards that feature an ARM Cortex-M 32-bit microcontroller along with an STLINK debugger/programmer adapter and various other peripherals. For debugging and programming, STLINK is connected to a computer via a USB port. In turn, STLINK and MCU

are connected via SWD and UART buses. The former is used to flash the programs and control the code execution from the host computer. While the latter provides an interface for program I/O. The Nucleo boards vary in flash memory size and core speed. We used STM32 Nucleo-G071RB [33]. The microcontroller on the Nucleo board is STM32G071RB, accessible from the outside via the STLINK/V2-1 debugger adapter.

The DirtyJTAG [17] project provides JTAG adapter firmware for STM32 development boards. It relies on host computer software to control the adapter, similar to how XSDB and the Platform Cable USB II operate. Since we wanted to avoid this, we decided to implement our JTAG adapter from scratch.

We programmed the microcontroller in STM32CubeIDE [35], which is part of the STM32Cube software system [34]. The system was created to simplify the development of applications for the STM32 microcontrollers. Apart from IDE, it provides firmware libraries that implement high-level interfaces for accessing onboard peripherals. We used the Hardware Abstraction Layer library (HAL) [31] to interact with the GPIO pins (e.g., `HAL_GPIO_ReadPin` and `HAL_GPIO_WritePin` calls). STM32Cube also comes with STM32CubeMX [36], a configuration tool for hardware peripherals that automatically generates initialization functions for microcontroller components running at the beginning of the program.

We aimed to implement the most basic JTAG adapter, and thus we focused only on the necessary parts of the standard. According to the JTAG specification [24], four signals constitute the minimal requirements of a working JTAG probe: TCK, TDI, TDO, TMS:

- **TCK:** The clock signal is generated on the GPIO pin using the Pulse Width Modulation mode on the first channel of the 16-bit general-purpose timer TIM3 with the internal clock as a clock source running at 16MHz. We used the default parameters set by the IDE during the project creation, only adjusting the pulse of the PWM to half of the period counter such that the duty cycle of TCK remains at 50%. Figure 3.4 shows this setup in STM32CubeMX. The resulting clock speed of TCK is  $16MHz/65535 = 244Hz$ .
- **TDI & TMS:** We decided to implement the simplest form of the JTAG controller, which was completely static and did not interact with the user during the program execution. Thus, it was sufficient to store the input TDI and the control TMS signal sequences we wanted to supply to the test logic as static byte arrays and recompile the program for each test. We wrote the sequences by hand as bit strings. Python script (generated with the help of ChatGPT 4.0) then transformed the sequences into arrays and pasted them into the code. However, in

### 3.1. Fixing the JTAG issues with ThunderX

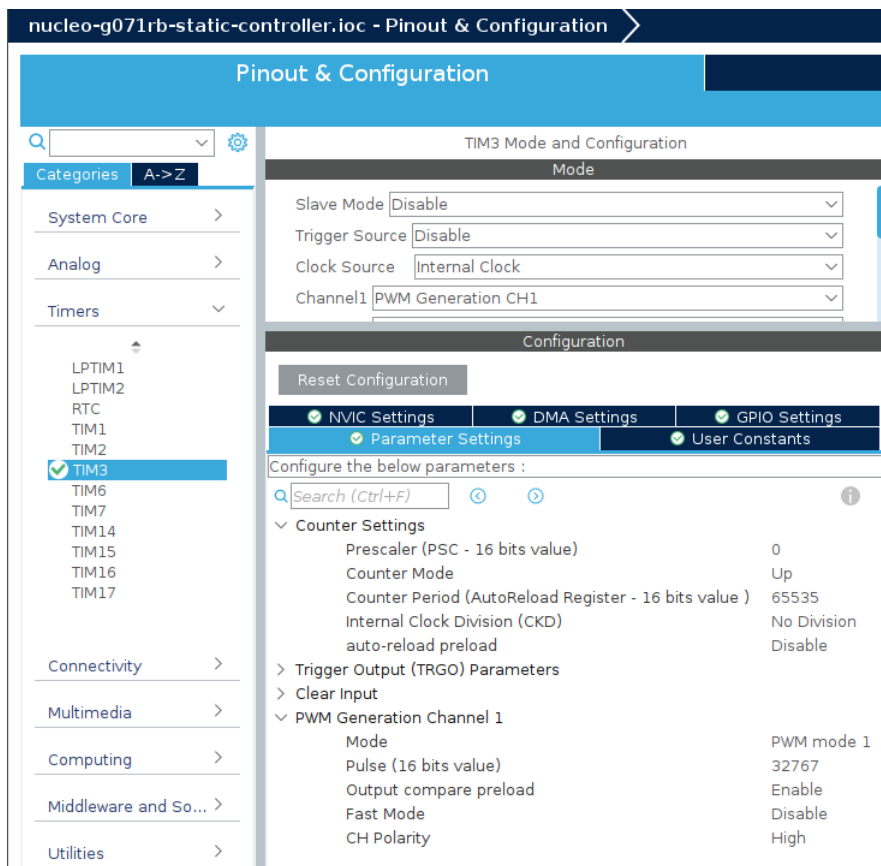


Figure 3.4: PWM configuration

the end, we have also implemented a more dynamic alternative that asks the user to supply signals through the UART console. After the microcontroller finishes executing the sequences, it resets the state machine and waits for the next test.

- **TDO:** Initially, we avoided any processing of the TDO signal: we could easily capture TDO on the oscilloscope and inspect it manually. It was possible to do so due to the low frequency of the JTAG controller (244Hz) and easily recognizable form of output for our simple test cases (known in advance bit sequence for IDCODE test and simple periodic signal for BYPASS). Ultimately, we opted to store the TDO bits in the static array the same way as we did for the input signals and output the result on the UART console.

We decided not to implement active triggering of TRST since we did not program any dynamic events in the microcontroller, and the state machine had built-in synchronous reset functionality. Instead, we assigned a GPIO pin that supplied logic 1 to the TRST line on the header to ensure the TAP

controller did not stay in the reset state.

We mapped the signals onto GPIO pins as follows: TCK:PA6, TDI:PA7, TMS:PB0, TDO:PC7, TRST:PA8. However, this can be easily adjusted in CubeMX.

We discuss in more detail the implementation of the static version of our program that does not rely on UART for input. Its main part (Listing 3.6) is the for loop, where the microcontroller alternately polls on the rising and falling edges of the clock TCK. On the falling edge, it updates the input signals TDI and TMS for the next clock cycle (lines 7-10). On the rising edge, it samples TDO as it is supposed to be stable at this point (lines 12-15). (the output of the test logic for the clock cycle happens on the falling edge.) When the Nucleo board finishes going through the input, it resets the state machine by driving TMS to 1 (lines 19-22) and prints the captured output signal in the binary and hexadecimal forms on the UART console. (One can open the UART console in STM32CubeIDE [32].) To print the formatted output using the C standard library function `printf()`, we modified the `_write()` system call in `syscall.c` that sends the data onto the UART console using the `HAL_UART_Transmit` command (Section 2.2.2.1 of the STM32CubeIDE manual [35]).

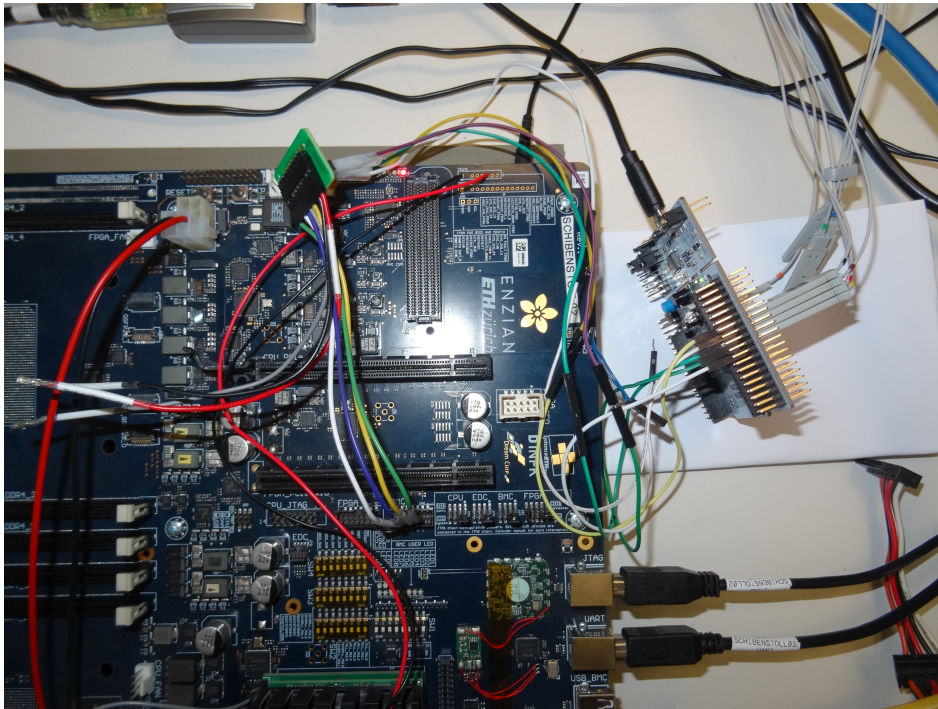
---

```
1 HAL_Delay(3000);
2 . . .
3 for (int i = 0; i<message_len; i++){
4     command_bit = command_sequence1[i];
5     data_bit = data_sequence1[i];
6     //change tdi & tms on the falling edge
7     while( HAL_GPIO_ReadPin(TCK_GPIO_Port, TCK_Pin)
8           == GPIO_PIN_SET){}
9     HAL_GPIO_WritePin(TSM_GPIO_Port, TSM_Pin, command_bit);
10    HAL_GPIO_WritePin(TDI_GPIO_Port, TDI_Pin, data_bit);
11    //sampling happens on the rising edge
12    while( HAL_GPIO_ReadPin(TCK_GPIO_Port, TCK_Pin)
13          == GPIO_PIN_RESET){}
14    output_sequence[i]
15        = HAL_GPIO_ReadPin(TDO_GPIO_Port, TDO_Pin);
16    . . .
17 }
18 //restore the default signal values
19 while( HAL_GPIO_ReadPin(TCK_GPIO_Port, TCK_Pin)
20       == GPIO_PIN_SET){}
21 HAL_GPIO_WritePin(TSM_GPIO_Port, TSM_Pin, 1);
22 HAL_GPIO_WritePin(TDI_GPIO_Port, TDI_Pin, 0);
23 . . .
24 while (1) {}
```

---

**Listing 3.6:** Main microcontroller loop





**Figure 3.5:** Nucleo board connected to schibienstoll02

Note that the JTAG controller would only work if we made a long delay at the start of the execution (line 1), where the TMS signal still had its default value of 1, setting the state machine into Test-Logic-Reset. Simply putting a long sequence of 1s at the beginning of the TMS array did not work.

To test our program, we replicated the scripts we ran on the Platform Cable. We tried to connect the pins TDI, TDO, TCK, and TMS to schibienstoll02 using jump wires. The sockets of the jump wires were too thick for the BMC's connector. Our solution was to remove the sockets, isolate the exposed wires, and plug them into the Flying wire cable. The cable forwarded the signals to the BMC's connector. We decided not to supply TRST since it was not used by the Platform Cable. The pin headers on the Nucleo board are double-sided, so we were able to simultaneously plug the jump wires and the logic probes of the oscilloscope. (Figure 3.5)

Figure 3.6 displays the output of the IDCODE read test as it appears on the UART console. (I) and (IV) hold TMS high for five clock cycles to reset the test logic at the beginning and end of the test. We immediately navigate the state machine from TEST-LOGIC-RESET to SHIFT-DR (II). (III) shifts out 32 bits of the instruction register by looping the state machine in the SHIFT-DR state. The last bit is shifted out during the transition to Exit1-DR. Thus, the last bit in the TMS part of (III) is 1. The contents of the data register appear

TDI:	00000	0000	00000000000000000000000000000000	00000
TDO:	11111	1111	11101110001000000000010111011010	11111
TMS:	11111	0100	00000000000000000000000000000001	11111
	⏟	⏟	⏟	⏟
	I	II	III	IV

**Figure 3.6:** IDCODE read

TDI:	00000	00000	1111111111111111101111111111111111	00000
TDO:	11111	11111	100010101110000011111111111111110	11111
TMS:	11111	01100	00000000000000000000000000000001	11111
	⏟	⏟	⏟	⏟
	I	II	III	IV

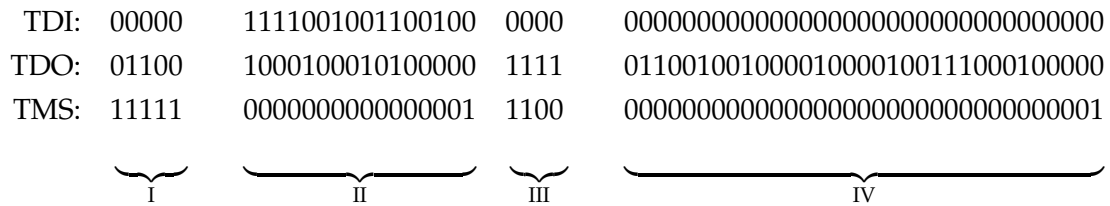
**Figure 3.7:** Register length estimation

on the scan line least significant bit first. We get the same IDCODE value of the ARM DAP 11101110001000000000010111011010 → 7704a05b (XSDB conversion binary to hex conversion) as in Listing 3.1.

Next, we repeated the instruction register width test (Figure 3.7) by navigating into SHIFT-IR (II) and shifting in the sequence 11...1011...1 (III). We got the same offset of bit 0, confirming the instruction register width of 16.

In Listing 3.4, we determined the IDCODE binary for Zynq Ultrascale’s TAP controller. We use that in Figure 3.8, where we tested the functionality of the instruction register. (We omit the reset sequences here to save space.) We navigated the state machine from TEST-LOGIC-RESET to SHIFT-IR (I), shifted in BYPASS for ARM DAP and IDCODE for the FPGA (II), and moved from CAPTURE-IR to SHIFT-DR (III). Similarly to Listing 3.3, the identification number of the FPGA appeared on the output delayed by one clock cycle due to the BYPASS test data register of the ARM DAP.

Convinced that we had a working implementation of the JTAG controller, we connected it to the CPU’s TAP on Enzian. We repeated all the tests that we ran on the Platform Cable. However, this time, we tried different paths through the state machine (e.g., the ones that went through or avoided the IDLE state). We also tested different frequencies and duty cycles by adjusting the corresponding factors in the Pulse Width Modulation settings in STM32CubeMX. Nothing changed; the TDO line remained high throughout all the tests.



**Figure 3.8:** Explicit BYPASS and IDCODE

### 3.1.4 Solution

The fact that the DAP stayed completely unresponsive throughout all the tests led us to believe that the TAP controller was stuck in the reset. This indeed turned out to be the case. Upon inspection of the board schematics [20] (page 117 of the mainboard document), we noticed that a pin header separates the TRST signal of the CPU’s JTAG header from the TRST line of the Test Access Ports. In Figure 2.6, this 6-pin connector is to the left of the CPU’s JTAG connector. If disconnected, the TRST lines to the TAPs are set to logic 0 by pull-down resistors, thus keeping the TAP controller in the reset state. This explains why ThunderX’s TAP stayed completely unresponsive for Platform Cable USB II and STM32 JTAG controller tests. This behavior violates rule 4.6.1b) of the JTAG standard, which states that “the design of the circuitry fed from that input shall be such that an undriven TRST\* input produces a logical response identical to the application of a logic 1.” [24]. After we had forwarded the incoming TRST to the reset line of the DAP, we pulled the state machine out of the reset state. When we repeated the steps described in Section 3.1.1, the autodetection created a valid platform configuration file, and we could connect to ThunderX from the ARM Development Studio. All the conventional debug functionality was present: we were able to halt the processor, step through the code, disassemble the code, and explore the contents of the system registers. (Figure 3.9)

### 3.1.5 Further problems with the Autodetection

Although the autodetection was successful enough to create a valid platform configuration file, some parts of the procedure still failed, namely, the scan chain enumeration and the exploration of the second Access Port (lines 10 and 39 in the session log A.2). ARM documentation lists the autodetection steps and provides their brief decription [6, 27].

For the scan chain enumeration, ARM DS employs a BYPASS algorithm similar to the one described in Section 2.1.4. ARM DS brings each TAP controller into a BYPASS mode and shifts in a long sequence of 1s. The number of bits 0 that appear on the output line corresponds to the device

### 3. IMPLEMENTATION

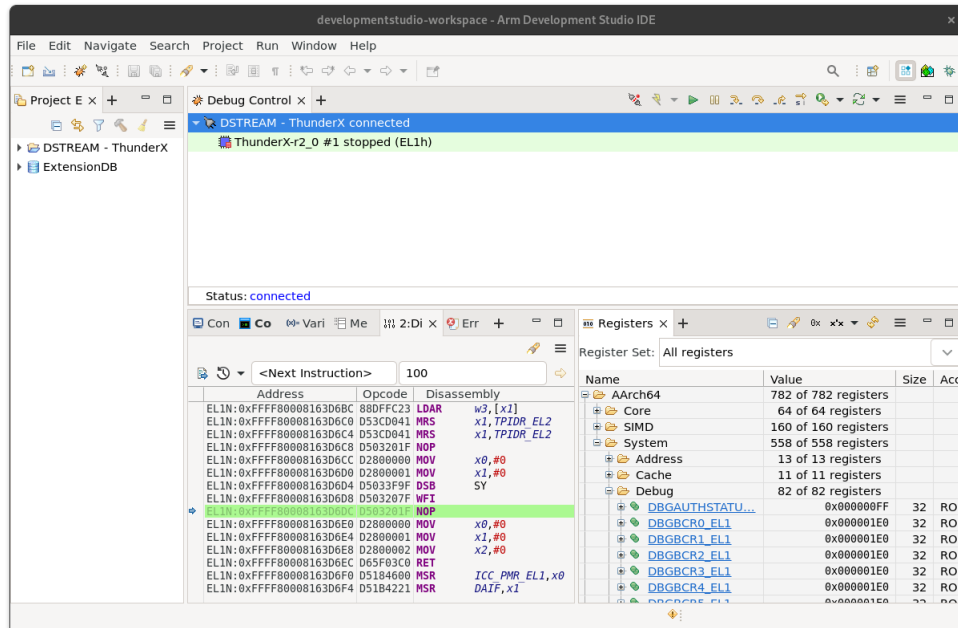


Figure 3.9: Debug connection to ThunderX

count. This procedure happens right at the beginning of the autodetection, so we confirmed it by analyzing the signal sequences on the oscilloscope. The TDO line indeed remained at level 1 throughout the shift phase of the test data register. One potential cause of faulty autodetection is the rest signal [27]. However, in our case, the TAP controller clearly left the reset state throughout the execution. For example, during the SHIFT-IR stage, the TDO line contained the bit sequence 0b1000, i.e., the test logic successfully shifted out the default contents of the instruction register.

Another possibility is the faulty implementation of the bypass mode [6]. We wanted to investigate the BYPASS command on ThunderX's DAP further using our custom JTAG controller. We connected the 5 JTAG signals (TDI, TDO, TMS, TCK, and TRST) mapped to the pins of the Nucleo board to the ThunderX's ARM JTAG 14 port. We also ensured that the jumper forwarded the reset signal to the DAP and that the CPU's configuration jumpers on the daisy chain were configured for bypass mode. The signal sequence shown in Figure 3.10 reruns the BYPASS test and fails to identify the device on the scan chain. Our hypothesis is that the TDI signal is scanned into the BYPASS test data register before the TAP controller enters SHIFT-DR. Given that the register width is one, the first bit that appears on the output line when we start shifting is not the default content of the test data register (i.e., bit 0). Instead, it is the value on the TDI line from the previous clock cycle when the state machine was in CAPTURE-DR. Our claim is further supported by

```

TDI: 11111 11111 1111 1111 11111111111111111111111111111111
TDO: 11111 11111 1000 1111 11111111111111111111111111111111
TMS: 11111 01100 0001 1100 00000000000000000000000000000001
  
```

**Figure 3.10:** Enzian, BYPASS test 1

```

TDI: 11111 11111 1111 0001 000000000000
TDO: 11111 11111 1000 1111 1000000000000
TMS: 11111 01100 0001 1100 000000000001
  
```

**Figure 3.11:** Enzian, BYPASS test 2

```

TDI: 11111 11111 1111 1110 111111111111
TDO: 11111 11111 1000 1111 0111111111111
TMS: 11111 01100 0001 1100 000000000001
  
```

**Figure 3.12:** Enzian, BYPASS test 3

the BYPASS tests in Figures 3.11 and 3.12. The shifted bit is highlighted in bold. This is a clear violation of the JTAG standard, which stipulates that test logic should propagate data along the scan chain only on the rising edge of TCK when the TAP controller is in the SHIFT state. For example, if we repeat the test in Figure 3.11 on the functioning scan chain used in Section 3.1.2, the highlighted bit 1 never appears on the output line. Nevertheless, the failure of the enumeration does not influence the rest of the autodetection. ARM DS behaves as if there is only one DAP on the chain. However, such message explicitly appears only in the faulty autodetection log (Line 22 in A.1) and not in the working one (A.2).

The second failure occurs due to an unsuccessful memory read through the second Access Port in the DAP. We wanted to further investigate the faulty Access Port and the DAP in general. Since both Platform Cable USB II and our STM32 adapter operated on the level of the JTAG protocol, we needed a tool that specifically worked with the DAP.

### 3.1.6 CoreSight Access Tool

The CoreSight Access Tool (CSAT) [9] is a command-line software designed to test CoreSight systems connected via DSTREAM. Since CSAT interacts

### 3. IMPLEMENTATION

---

```
1 %>connect TCP:dstream1.ethz.ch
2 Attempting to connect to TCP:dstream1.ethz.ch
3 Connected to:DSTREAM
4 . . .
5 %>chain dev=auto clk=20000
6 Jtag clock set to 20000
7 ID:0 ARMCS-DP
8 %>devopen 0
9 Open connection to device ID :
10 0x1A101399, version 0x00000006
11 Msr returned with RVMOpenConn: ARM-DP Template using Rv-Msg
12 %>dapenum
13 Enumerated 2 APs
14     0 : APB-AP
15     1 : APB-AP
```

---

**Listing 3.7:** CSAT connection

with CoreSight infrastructure, its interface operates at the level of the DAP protocol, abstracting away underlying JTAG communication. Its byte level reads and writes to the address space of the CoreSight DAP, making it a valuable tool for manual exploration of a DAP. Apart from debugging, CSAT also exposes to the user an extensive tracing interface of DSTREAM. It is shipped with ARM Development Studio. The executable is in the binary directory of an ARM DS installation.

The following steps describe the way to establish a connection to a DAP that is interfaced with the DSTREAM probe:

1. Connect to DSTREAM either via TCP (`connect TCP:<IP address>`) or USB (`connect USB`)
2. Directly specify the devices on the chain (`chain dev=ARCMS-DP clk=10000000`) or use the auto-detect option (`chain dev=auto clk=A`) to get the list of the devices.
3. Select one of the DAPs on the chain with the device open instruction (`devopen 0` selects the first device in the list)
4. (Optional) Enumerate the Access Ports (APs) of the DAP (`dapenum`)

Listing 3.7 shows the result of the commands for ThunderX's DAP. We opted to use the slowest possible frequency for the auto-detect because, otherwise, CSAT often fails. On the other hand, the manual setup of the scan chain works without a problem. We tested it up to 1MHZ. CSAT correctly detects the single Debug Port on the scan chain and two associated Access Ports.

CSAT supports two types of I/O operations:

- Read/Write to Debug Port (DP) and Access Port (AP) registers configure the ports and run the DAP protocol. The operations are of the form `dregread <port type>.<register>` and `dregwrite <port type>.<register name/offset>` value where the `<port type>` is either `dp` or `ap` and the `<register>` is a register name (e.g., `csw` for control status word register) or a numerical register offset.
- Read/Writes to the DAP memory space. The operations are of the form `dpmemread <ap number> L|l:<address>[.b|B|h|H|l|L] <no.items>` and `dpmemwrite <ap number> L|l:<address>[.b|B|h|H|l|L] <data> [<data>]*`. `<ap number>` is an index into the list of Access Ports provided by `dapenum` instruction, the `<address>` is by default 32-bit or 64-bit if the optional argument `L`: is present. Both instructions can initiate multiple sequential memory accesses. For the read operation, `<no.items>` specifies the number of accesses, while for write operations, the data to be written is provided as a sequence at the end of the instruction.

Using CSAT, we can fully configure DP and AP registers and inspect the memory:

- Reading the Debug Port Identification Register DPIDR

---

```

1    %>drd dp.0 #CSAT does not provide a name for the DPIDR. We
      have to use numerical offset
2    dp.0 => 0x0A111399

```

---

0x1 in the version field [15:12] of the identification register means that the DAP implements DPv1. Note that DPIDR 0x0A111399 and IDCODE 0x1a101399 of ThunderX's DAP (that we determine later in Section 3.2.1) are the same up to version fields [31:28] and the MIN bit [16] in DPIDR.

- Reading the Access Port Identification register ID

---

```

1    # select the first Access Port by writing index 0 to the
      APSEL field of the AP select register. Also select the
      last register bank on the AP (0xF in the APBANKSEL field)
2    %>drw dp.addr 0x000000F0
3    %>drd ap.3 # Read the contents of the last register in the
      bank.
4    ap.3 => 0x03990003

```

---

The 4 least significant bits of the instruction register that identify the memory bus the Access Port connects to contain a reserved entry 0x3 that does not correspond to any standard AP. Thus, the autodetection tools may fail to determine the bus type of ThunderX's APs (we en-

counter it later in Section 4.2.2). IDR of the second AP 0x03990013 differs only in the variant field, so the manufacturer could differentiate between APB and CVM Access Ports in ThunderX.

- ThunderX provides access to system registers on the processor's cores via a Cavium-specific access port. The most recent version of the manual [13] states that to start the transaction, one must configure the Debug Address Register (DAR) according to the format in Table 21-2. The DAR is not mentioned anywhere else in the manual; we assumed it is the same as the Transfer Address Register (TAR) of the DAP in ARM's terminology. The address format includes the core number and the opcodes of the system registers listed in Table 2-17 of the manual. Unfortunately, when we tried to read the AP\_MIDR\_EL1 register of the fifth core, the memory reads through the first and second Access Ports failed (`dmr 0 0x00450000 0` and `dmr 1 0x00450000 0`).

On the other hand, according to the older version of the manual [12], one can access the system registers through the CVM-AP by simply supplying their I/O physical address. This approach worked. For example, we were able to read NIC Receive Ethertype Register `NIC_PFRX_ETYPE` by direct memory access through CVM-AP. Its address is on page 773, and the contents are on page 785 of ThunderX's manual [13]. The example also confirms that the CVM-AP is the second Access Port in the DAP.

1	# APB-AP	1	# CVM-AP
2	%>dmr 0 L:0x843000000500 1	2	>dmr 1 L:0x843000000500 1
3	0x00000500 : 0x00000000	3	0x00000500 : 0x00148100

- CSAT does not have a separate instruction that displays ROM tables. It has to be done via a manual read. The first two entries of the APB-AP ROM's table point to the debug register file of the first core in ThunderX and the cross-trigger interface file. (cf. Lines 28 and 32 in A.2)

1	%>dmr 0 0x80000000 4
2	0x80000000 : 0x88000003 0x88010003 0x88020003 0x88080003

## 3.2 OpenOCD

### 3.2.1 Motivation for OpenOCD

After correctly forwarding the TRST signal to ThunderX's DAP, we successfully established the debug session from the ARM Development Studio using DSTREAM. The major downside of this solution is that it requires physical



access to the machine. It is impractical for most of Enzian users because they connect to the machines remotely. Furthermore, it caps the number of simultaneous debug sessions based on available DSTREAMs.

The onboard programming module JTAG-SMT3-NC [15] mentioned in Section 2.4 could remedy these issues. Currently, it operates on Enzian as a programming module for the FPGA with the other devices on the daisy chain setup in the bypass mode. The module is accessed through the Xilinx software such as Vivado or XSDB. If we make the CPU visible on the daisy chain by setting the jumpers in normal mode, the chain becomes unresponsive and keeps the output line at logic 1 (Listing 3.8).

---

```

1 xsdb% jtag targets
2   . . . . .
3 7 Digilent JTAG-SMT3 210357A7CB8FA (error DR shift output all ones)
4   . . . . .

```

---

**Listing 3.8:** jtag targets with TRST unset

This is because JTAG-SMT3-NC does not supply the TRST signal to the daisy chain (page 118 of the mainboard schematics [20]), so we face the same problem we had with the DSTREAM unit. To fix this, we connect the reference voltage from pin 13 of the CPU's ARM JTAG 14 connector to the TRST pin 3. Also, we make sure that the jumper forwards the incoming TRST signal from the CPU's JTAG header to the DAP. Now, ThunderX's DAP appears on the jtag targets list of XSDB (Lines 3-5 in Listing 3.9). Unfortunately, XSDB fails to recognize the TAP model and does not identify it as a DAP (Line 8). As we discussed in Section 3.1.6, this is because ThunderX features a DAP with a non-standard IDCODE. Note that XSDB successfully identifies the DAP on the other computer visible to the hardware server (Line 10) since it has a standard identification number 4ba00477 [11]. The Xilinx software lacks ways to assign the correct device identity manually. Thus, we had to turn to other alternatives.

---

```

1 xsdb% jtag targets
2   . . . . .
3 7 Digilent JTAG-SMT3 210357A7CB8FA
4 8 unknown (idcode 1a101399 irlen 4)
5 61 xcvu9p (idcode 14b31093 irlen 18 fpga)
6 9 Digilent JTAG-SMT3 210357A7CF8CA
7 10 arm_dap (idcode 4ba00477 irlen 4)
8 11 xc7z015 (idcode 0373b093 irlen 6 fpga)
9   . . . . .

```

---

**Listing 3.9:** jtag targets with TRST set

#### 3.2.2 Brief Overview

The Open On-Chip Debugger (OpenOCD) is a software for debugging, flash programming, and boundary-scan testing of embedded systems. Initially created by Dominic Rath as part of his diploma thesis [30], it has since become a fully-fledged open-source project [28].

We now give a brief overview of the tool as it is described in the User's Guide [29]. Thanks to its extensive configuration options, OpenOCD is a highly versatile tool. It provides drivers for dozens of debug adapters that employ different transport protocols (JTAG, SWD, SPI) and supports a wide range of debug targets, from 8-bit STM8 microcontrollers to 64-bit ARM CPUs. We worked with the most recent version of OpenOCD (0.12.0) compiled from the source. OpenOCD operates in two stages: configuration and execution. During the execution stage, OpenOCD starts up the server. Clients can communicate with the server via telnet or by attaching gdb. The stages are divided by the `init` command in the configuration file. After this command, the file may contain server related instructions.

OpenOCD is written in C, but the configuration is done in TCL. The configuration is modular. By convention, the adapter (called interface) and the target are configured as separate TCL scripts, which can later be sourced in a single board file. The scripting is mostly linear, but we can register event handlers for predefined events to deal with any asynchrony. Valid events include gdb connection (`gdb-attach`), halting of a target (`halted`), assertion of the system reset `SRST` (`reset-assert`), and many more.

In the interface file, we usually select one of the adapter drivers that come with OpenOCD and specify which optional JTAG reset signals are implemented by the adapter. However, sometimes, device identification and further configuration may be needed. This was the case with the JTAG controller on Enzian.

The target configuration is more involved. It requires explicit enumeration of all visible TAPs on the scan chain and the associated DAPs. Also, each CPU core has to be declared separately. If the configuration is successful, one can issue various generic and microarchitecture-specific commands. Most of the former are memory and register accesses, while the latter vary wildly depending on the architecture. They include ARM Cross-Trigger Interface configuration, instruction disassembly, cache inspection, and trace collection.

#### 3.2.3 Simple setup

First, we show how to set up a simple connection. We demonstrate it on the STM32 Nucleo-G071RB. `openocd` command starts up the server. If no argument is supplied, the program looks for the configuration file `openocd.cfg` in the current working directory. OpenOCD is shipped with a script library

for dozens of adapters and debug targets. If we can use one of the files, we can skip the setup process by referencing the configuration file in the command: `openocd -f <file name>`.

The MCU on STM32 Nucleo-G071RB is STM32G071RB, accessible from the outside via the STLINK/V2-1 debugger adapter. OpenOCD of version 0.12.0 has configuration files for the microcontroller and adapter. The files are located in the interface and target directories of the script library. They are sourced in a single board configuration `st_nucleo_g0.cfg`. Thus `openocd -f board/st_nucleo_g0.cfg` successfully connects to the Nucleo board, and we can attach the cross-platform gdb from the `gdb-multiarch` package running on the x86 Ubuntu host. Another option would be to use the `gdb-arm-none-eabi` version of gdb, built specifically for ARM targets. As we can see in A.5, the gdb successfully attaches to the microcontroller, and we can control its execution with the conventional debugging functionality.

### 3.2.4 Interface file for the onboard adapter

Although JTAG-SMT3-NC does not have a configuration file, previous generations of the module do. First, we studied the configuration file of the direct predecessor JTAG-SMT2-NC [16] and subsequently adjusted it for compatibility with JTAG-SMT3-NC.

**JTAG-SMT2-NC** The core of JTAG-SMT2-NC is FTDI FT232HQ.<sup>1</sup> FT232HQ is a Single Channel Hi-Speed USB to Multipurpose UART/FIFO converter [22]. FTDI's Multi-Protocol synchronous Serial Engine (MPSSE) interfaces different types of synchronous serial devices (SPI, I2C, JTAG, SWD) to a USB port [21]. JTAG-SMT2-NC schematics are not publicly available. However, we assume that when MPSSE operates in JTAG mode, FT232HQ converts data from OpenOCD into signals conforming to the JTAG protocol. These signals are, in turn, forwarded to the output pins of JTAG-SMT2-NC. OpenOCD implements a driver for FTDI chips that utilize MPSSE. JTAG-SMT2-NC's file uses that driver to configure the adapter:

---

```
1 adapter driver ftdi
2 ftdi device_desc "Digilent USB Device"
3 ftdi vid_pid 0x0403 0x6014
4 ftdi channel 0
5 ftdi layout_init 0x00e8 0x60eb
6 reset_config none
```

---

**Listing 3.10:** Interface configuration file for JTAG-SMT2-NC

---

<sup>1</sup>The reference manual does not state it directly. We can determine the converter's name from the picture of JTAG-SMT2-NC on page 9 of the reference manual [16].

The adapter's name, its vendor and product IDs are specified in (2) & (3). It serves as a sanity check to confirm that the program connects to the correct adapter. (4) selects the channel of the FTDI chip for MPSSE operations (note that FT232HQ is a single-channel device). (6) states that the adapter supports neither TRST nor SRST signals. The first hexadecimal number in (5) specifies the initial values of the 8-bit register banks of the channel. The eight least significant bits correspond to the low bank pins ADBUS0-7; the rest are the high bank pins ACBUS0-7. The pinout diagram of the converter is depicted on page 4 of the datasheet [22]. The second number sets the direction of the pins. Logic 1 and 0 correspond to output and input, respectively (in the point of view of the module.) According to Section 3.5.5 "FT232H Pins used in an MPSSE" of the datasheet [22], in the JTAG mode, pins 13, 14, 15, and 16 (i.e. pins ADBUS0-3) contain four standard JTAG signals. They are, in turn, forwarded to TCK, TDI, TDO, and TMS output pins of JTAG-SMT3-NC [26]. For unknown reasons, JTAG-SMT2-NC duplicates the directional settings of FTDI pins. Output enable pins OETMS, OETDI, and OETCK connected to ADBUS5-7 must be driven to 1 for TMS, TDI, and TCK pins of JTAG-SMT3-NC to become outputs.

From the above, we deduce the least significant bits of `ftdi layout.init` parameters: The default value 11101000 or 0xe8 in [ADBUS7:0] occurs in 0x00e8 if we replace the irrelevant values with 0. Similarly for the direction value, 11101011 or 0xeb appears in 0x60eb (Table 3.1). Because we do not know how JTAG-SMT2-NC uses the high bank pins (ACBUS0-7), we cannot explain the corresponding default settings, especially logic 1 direction pins for ACBUS5 & 6 in 0x60eb.

**JTAG-SMT3-NC** follows the same structure as its predecessor, with an embedded FTDI FT2232HQ module acting as a protocol converter.<sup>2</sup> Although FT2232HQ is dual channel, all the relevant JTAG signals are mapped to channel 1 (see Section 3.1.4.5 "FT2232H pins used in an MPSSE" of the reference manual [23]). Furthermore, JTAG-SMT3-NC removed unnecessary directional pins leaving a single JTAG enable pin [25]. Thus, similarly to the previous case, we can derive the initialization values for the `ftdi` driver:

Again, we substitute the irrelevant values with 0 and get the following least significant bits of the configuration: default value 10001000 or 0x88, direction value 10001011 or 0x1b. The pinout mapping also contains support for a system reset signal. The pins are mapped to the high GPIO register ACBUS of the first channel. In particular, the SRST line is mapped to ACBUS5 and OESRST to ACBUS4. We decided to not enable the system reset functionality as it simplified the configuration.

---

<sup>2</sup>See the picture of the module on page 1 of the reference manual [15].

FTDI Pin	JTAG-SMT2-NC Pin	Default Value	Direction Value
ADBUS0	TCK	0	1
ADBUS1	TDI	0	1
ADBUS2	TDO	0	0
ADBUS3	TMS	1	1
ADBUS4	unknown	irrelevant	irrelevant
ADBUS5	OETMS	1	1
ADBUS6	OETDI	1	1
ADBUS7	OETCK	1	1

Table 3.1: FTDI to JTAG-SMT2-NC Pin Mapping

FTDI Pin	JTAG-SMT3-NC Pin	Default Value	Direction Value
ADBUS0	TCK	0	1
ADBUS1	TDI	0	1
ADBUS2	TDO	0	0
ADBUS3	TMS	1	1
ADBUS4	unknown	irrelevant	irrelevant
ADBUS5	unknown	irrelevant	irrelevant
ADBUS6	unknown	irrelevant	irrelevant
ADBUS7	OEJTAG	1	1

Table 3.2: FTDI to JTAG-SMT3NC Pin Mapping

```

1 adapter driver ftdi
2 ftdi device_desc
3 ftdi vid_pid 0x0403 0x6010 # Device descriptors are obtained
4                               # via lsusb -v
5 ftdi channel 0
6 ftdi layout_init 0x0088 x008b
7 reset_config none

```

Listing 3.11: Interface configuration file for JTAG-SMT3-NC

As the last step of the adapter setup, we added non-root access permissions to JTAG-SMT3-NC so that OpenOCD would work correctly in the user space. Our host computer was running Linux, so we made a new udev rule `digilent-jtag-smt3.rules` with the following contents: `SUBSYSTEM==usb, ATTRidVendor==0403, ATTRidProduct==6010, MODE=666`.

### 3.2.5 ThunderX's configuration file

Writing the target configuration file for the CPU is straightforward and follows the steps described in Section 3.2.2.

- (1)-(3) select JTAG as a transport protocol and set the frequency of

### 3. IMPLEMENTATION

---

```
1 transport select jtag
2 reset_config none
3 adapter speed 100
4
5 jtag newtap auto0 tap -irlen 4 -expected-id 0x1a101399
6 dap create auto0.dap -chain-position auto0.tap
7 cti create cti0 -dap auto0.dap -ap-num 0 -baseaddr 0x88010000
8 target create core0 aarch64 -dap auto0.dap -ap-num 0 -coreid 0 -cti
   cti0 -dbgbase 0x88000000
9 cti create cti1 -dap auto0.dap -ap-num 0 -baseaddr 0x88090000
10 target create core1 aarch64 -dap auto0.dap -ap-num 0 -coreid 1 -cti
   cti1 -dbgbase 0x88080000
11
12 target smp core0 core1
13
14 init
```

---

**Listing 3.12:** ThunderX's configuration file

#### JTAG-SMT3-NC

- (5)-(6) initialize ThunderX's TAP and bind its DAP. Note that all the devices on the chain that are set in non-bypass mode have to be declared in the order they appear on the scan chain, starting from the one closest to the adapter.
- (7) declares the Cross-Trigger Interface for the first core of ThunderX. It is a mandatory parameter for ARMv8 cores (page 78 of the User's Guide [29]). The Base Address in the memory space of the Access Port was taken from the ROM table section in the autodetection log (Line 31, A.2). Note that the CoreSight Debug infrastructure sits behind Access Port 0, as we found out in Section 3.1.6, hence the `-ap-num 0` option.
- (8) declares the first core, specifies its microarchitecture, assigns the CTI, and sets the starting address of the core's debug registers (Line 28, A.2). All other cores have to be configured separately.
- (12) binds both cores for SMP
- (14) signals the end of the configuration. OpenOCD enters the run stage.

We source both `thunderx.cfg` and `digilent-jtag-smt3-interface.cfg` in the board configuration file `enzian.cfg` (Listing 3.13). After this, we can start the server with `openocd -f enzian.cfg`, and attach gdb the same way as we did for the Nucleo board. (Section 3.2.3) Note that we still had to manually supply the TRST signal to the ARM DAP the same way as we did in Section 3.2.1.

```
1 source [find interface/digilent-jtag-smt3-interface.cfg]
2 source [find target/thunderx.cfg]
```

---

**Listing 3.13:** Enzian's configuration file

## Chapter 4

---

# Evaluation

---

In this chapter, we assess the tools we have developed. In particular, we will analyze the functionality of the STM32-based JTAG controller and evaluate its performance in relation to other JTAG adapters utilized in our work. We will also compare two ways we have established a debug session with ThunderX: OpenOCD through the onboard controller and ARM DS through the DSTREAM unit.

### 4.1 STM32

#### 4.1.1 JTAG level

When testing an implementation of the JTAG adapter, one has to show that it can successfully drive a JTAG scan chain. We have already conducted such experiments for our Nucleo board on the schibenstoll02's scan chain (Figures 3.6 - 3.8) and on the Enzian's DAP connected through the CPU's JTAG connector (Figures 3.10 - 3.12). The signal sequences demonstrate that the Nucleo board can accurately navigate the state machine of the TAP controller, properly configure the instruction register, generate the input data, and sample the output at the correct points in the clock period. By comparing the tests to their XSDB counterparts (Listings 3.3 to 3.1), we deduce that the Nucleo board completely replicates the functionality of the `jtag sequence` interface that drives the Platform Cable USB II except for the explicit `delay` command. However, the tool lacks user-friendliness: in contrast to TCL scripting, manually writing all signal sequences, including the state transitions, was cumbersome and error-prone.

#### 4.1.2 Data input

Next, we investigate how the two ways we can supply data to the Nucleo board differ in user ergonomics. (See TDI&TMS implementation in Section



3.1.3.) We rerun schibenstoll02's tests both with hardcoded signal sequences and using UART. The latter avoids recompilation between the tests, so it does not break the workflow. However, we have not implemented any way to edit the input on the UART console. Thus, to minimize errors, we mostly resort to pasting complete signal sequences into the console. That renders the console redundant, as the Python script can seamlessly replace the TDI and TMS arrays in the source code. Additionally, recompiling and flashing the program is not an issue, as it only takes around 10 seconds.

The downside of both implementations is that they do not allow us to interact with the execution dynamically. For example, even though the Nucleo board supplies TRST, there is no way to trigger it asynchronously.

### 4.1.3 DAP level

In our work, we only utilized the Nucleo board to check the responsiveness of the TAP controllers by conducting simple BYPASS and IDCODE tests. However, this is not the limit of the tool since it can issue arbitrary control and input sequences. We demonstrate that the Nucleo board can access the DAP infrastructure on ThunderX situated behind the TAP, i.e., it also functions on the level of the ARM Debug Interface and can configure the DAP. We conduct two experiments. First, we will read the identification register of the first Access Port of ThunderX's DAP. We will also access the memory space of the AP.

We connect the Nucleo board to ThunderX the same way we did it in Section 3.1.5. We then issue a 275-bit series of DPACC and APACC commands. They all follow the same structure described in Section 2.3.1. For each command, we first bring the state machine into a known TEST-LOGIC-RESET state (I). We then proceed into SHIFT-IR (II) and shift in 0b1010 for DPACC or 0b1011 for APACC (III). (We know that ThunderX's DAP has a 4-bit instruction register from the output of jtag targets in Listing 3.9). Subsequently, we move from UPDATE-IR to SHIFT-DR (IV) and input the data for the instruction: three configuration bits (V) and, in the case of the write operation, 32 bits of data (VI). We start by configuring the CTRL/STAT register that manages the operation of the DAP (Steps 1 and 2) and then read the contents of the APIDR register (Steps 3, 4, and 5). In more detail for each command:

1. **SELECT.DPBANKSEL 0 write.** To access CTRL/STAT, the DPBANKSEL field of the SELECT register must be set to 0. The control bits 001 (V) supplied by the TDI line initiate a DPACC write to the SELECT register. (Section 2.3.1 describes the addressing model in detail.)

#### 4. EVALUATION

---

```

TDI: 00000 00000 0101 0000 001 00000000000000000000000000000000
TDO: 11111 11111 1000 1111 100 00000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 00000000000000000000000000000001

      I       II      III     IV    V           VI

```

2. **CTRL/STAT write.** The data portion of the TDI signal contains the new value of the CTRL/STAT register. We make sure that all DAP domains are powered up by setting the System powerup request CSYSPWRUPREQ and Debug powerup request CDBGPWRUPREQ fields (CTRL/STAT discussion in Section 2.3.2). The bits are highlighted in the TDI signal sequence. We also set all control fields whose default value is not specified by the standard (see Section 2.3.2). Note that the received response bit field 0b010 on the TDO line indicates the success of the previous register access.

```

TDI: 00000 00000 0101 0000 010 000000000000000000000000000001010
TDO: 11111 11111 1000 1111 010 00000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 00000000000000000000000000000001

```

3. **SELECT.APBANKSEL 0xF write.** The identification register APIDR lies in the last register bank of the Access Port. We write the bank's offset value 0b1111 in the APBANKSEL field of the SELECT register. Since we access the first Access Port in the DAP, the APSEL field is set to 0b0000.

```

TDI: 00000 00000 0101 0000 001 0000111100000000000000000000000000000000
TDO: 11111 11111 1000 1111 010 00000000000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 00000000000000000000000000000000000001

```

4. **APIDR read.** The APIDR is the fourth register in the bank. The address bits [2:1] in the APACC test data register contain its offset 0b11. Bit [0] initiates the read access to APIDR.

```

TDI: 00000 00000 1101 0000 111 000000000000000000000000000000000000000
TDO: 11111 11111 1000 1111 010 00000000000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 00000000000000000000000000000000000001

```

5. Now, all that is left is to loop back to CAPTURE-DR and latch the output of the register read onto the scan chain (I). Note that we omit to go through TEST-LOGIC-RESET to preserve APACC in the instruction register (otherwise, the DAP would default back to IDCODE). If we

reverse the data field (III), we get the APIDR value 0x03990003 that we determined in Listing.

TDI:	0000000000000000	000	00
TDO:	1111111111111111	010	110000000000000001001100111000000
TMS:	100000000000100	000	001

I
II
III

The memory access experiment follows the same structure. The signal sequences are in A.3.

Register and memory accesses build a complete programming model for the DAP. Thus, the tests show that the DAP can be fully configured by the Nucleo board. The additional functionality of the DAP still needs testing (e.g., the transaction counter, variable memory size access, and the Large Physical Address Extension (for the CVM-AP). We omit it due to time constraints.

Of course, the user-friendliness issue remains. The CoreSight Access Tool replaces all the signal sequences that access the DAP registers with a simple interface (Listing 3.1.6), while for the memory access (A.3), the program delivers the same result with just one command `dmr 1 0x80000040 1`. Also, the static nature of our JTAG controller is still a problem. For example, the Nucleo board cannot react to memory access failures by polling on the control bits and reissuing the failed commands.

#### 4.1.4 Clock speed

The throughput of any JTAG adapter depends on two metrics: its clock speed and the length of its path through the state machine. Optimizing the latter provides limited speedup because the TAP controller spends most of its time looping in the SHIFT states. Thus, when measuring the performance of our implementation, we will only focus on the clock speed.

We generated the clock signal on the TCK pin both using the Pulse Width Modulation. Alternatively, we could have connected the GPIO pin directly to the MCU's internal timer. The former option offers more flexibility in terms of possible frequency (the internal clocks only have a limited set of prescaler options) and change in duty cycle. But the end result is the same. So in our testing we will use the PWM Mode. Note that neither method can hold the clock signals to effectively halt the TAP controller, a feature heavily utilized by the DSTREAM unit.

For our test sequence, we pick a simple IDCODE read performed on ThunderX's DAP (Figure 4.1.4).

#### 4. EVALUATION

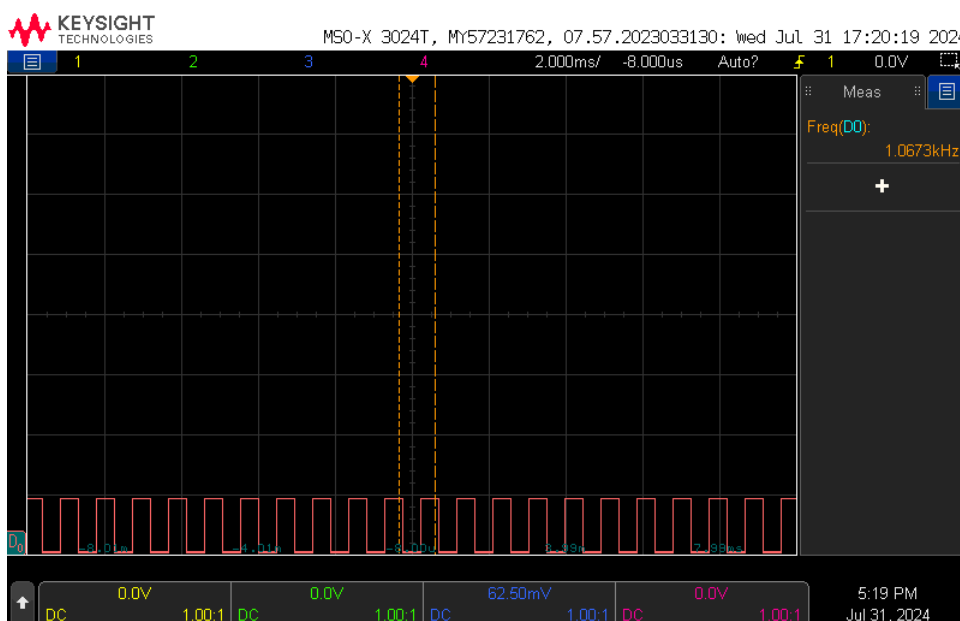
```

TDI: 00000 0000 000000000000000000000000000000000000 000000
TDO: 11111 1111 10011001110010000000100001011000 111111
TMS: 11111 0100 0000000000000000000000000000000001 111111

```

**Figure 4.1:** IDCODE test on Enzian

We aim to find the maximum operating frequency of our adapter running this test. We connect the Nucleo board to the ThunderX’s JTAG port in the manner described in Section 3.1.5. We set the timer’s clock source to its maximum value of 64 MHz, remove prescaling, and perform the binary search on the counter period value of the PWM based on whether the microcontroller correctly outputs the IDCODE of ThunderX’s DAP. The output is read from the UART console. The duty cycle of TCK is always set to 50% (i.e., the pulse value is half of the counter period). We start at the counter period value of 60000, which corresponds to 1.067kHz. Although we can determine the resulting clock speed from the source clock frequency and the counter period ( $64MHz/60000 \approx 1.067kHz$ ), we measure the frequency on the oscilloscope (Figure 4.2).



**Figure 4.2:** Clock speed measurement

We adjust the counter period until we are close to its precision limit. The resulting sequence is shown in Table 4.1.

Our implementation of the JTAG controller reaches its frequency limit when

Counter Period	TCK frequency	Result
60000	1kHz	OK
600	108kHz	OK
300	213kHz	FAIL: TDO is almost all 1
450	142kHz	OK
375	170kHz	OK
337	190kHz	OK
318	201kHz	FAIL: the TDO signal is skewed after bit 20
329	194.5 kHz	OK

Table 4.1: IDCODE speed estimation

the Nucleo board cannot keep up with the oscillations of TCK and, as a result, fails to correctly sample the output signal. For example, the frequency of 201kHz is close to the threshold since the TDO output is almost correct; it just fails to sample the signal in two clock periods, and that leads to a skew in the result:

```
correct TDO:  11111111110011001110010000000100001011000111111
TDO at 201kHz: 11111111110011001110100000001000101100011111111
```

Thus, for the IDCODE test, the threshold is between 195kHz and 200kHz. To be on the safe side, we further reduced it to 150kHz (counter period value 426) and tested our Nucleo board by reading the IDCODEs of schibenstoll02's scan chain (Figure 3.6) and rerunning APIDR (Section 4.1.3) and memory accesses (A.3) on ThunderX. The Nucleo board passed all of the tests.

Although the frequency of 150kHz is far lower than the maximum speed of the industry-grade JTAG adapters, it still falls within their operating range: 125kHz-30MHz for the Platform Cable USB II and 20kHz-180MHz for the DSTREAM unit.

## 4.2 OpenOCD

### 4.2.1 OpenOCD's limitations and boot problem

In this section, we evaluate the methods we used to establish the debug session with ThunderX, which is the primary goal of this thesis. The procedures are described in Section 3.1.4 for ARM Development Studio and at the end of Section 3.2.5 for OpenOCD.

In both bases, we first test the conventional debug functionality. We halt and resume the execution, walk through the code, disassemble the program, and

inspect the memory. ARM DS accomplishes all these tasks. Unfortunately, we encounter several issues when using OpenOCD. The gdb's `info registers` command fails (see A.4). Additionally, since we configured OpenOCD through the first Access Port, the tool is oblivious of the CVM-AP and cannot access the system registers of the core.

But the main problem occurs when we try to reboot Enzian. While it works with the DSTREAM unit plugged into the CPU's JTAG port, Enzian crashes on boot if the TRST pin is driven to logic 1. Recall that driving the pin is necessary since the onboard programming module JTAG-STM3-NC does not supply the TRST signal (Section 3.2.1). `0x0000000096000210` in the Exception Syndrome Register (ESR) reveals that a memory read caused the crash. The Fault Address Register (FAR) holds `0x000087e002000100`, the address of the DAP Debug Authentication Register (DAP\_IMP\_DAR) in ThunderX [13]. Its task is to configure the operation of ThunderX's DAP. For example, it can hide the trace unit or the memory buses used by Access Ports from the debug adapter. As we detail later in Chapter 5, one potential explanation is that there is a bus conflict between the DAP and the processor. We tried to solve it by resetting the DAP. Unfortunately, it did not work.

We can resume the execution by removing the jumper wire from the TRST pin and resetting the CPU via the Baseboard Management Controller (BMC) console. We can also program the Nucleo board to control the TRST line on ThunderX's JTAG port via the input from the UART console. When Enzian is booting, the Nucleo board should keep TRST low. We assert the reset line after the process is finished. Then, we can start up the OpenOCD server. Of course, this is impractical. Thus, we failed to reach our original goal for OpenOCD, which was to function as a remote debugging tool that did not require physical access to the machine.

### 4.2.2 JTAG and DAP API

The functionality of OpenOCD is not limited only to debugging. It exposes some parts of the JTAG and DAP API that function similarly to the Xilinx System Debugger and CoreSight Access Tool. We can use the API to turn JTAG-SMT3-NC into a low-level controller. We will briefly describe the commands' syntax, then test them on ThunderX's DAP.

For JTAG, OpenOCD supports navigation through the state machine using `pathmove` command<sup>1</sup>, halt of the execution with `runtest`, and I/O with `drscan` and `irscan` for JTAG data and instruction registers respectively. The commands can be issued through the telnet console or placed in the

---

<sup>1</sup>The state names used by `pathmove` are not covered in the manual. We were able to find them by inspecting the source code: `RESET`, `RUN/IDLE`, `DRSELECT`, `DRCAPTURE`, `DRSHIFT`, `DREXIT1`, `DRPAUSE`, `DREXIT2`, `DRUPDATE`, `IRSELECT`, `IRCAPTURE`, `IRSHIFT`, `IREXIT1`, `IRPAUSE`, `IREXIT2`, `IRUPDATE`.

execution stage of the configuration file. In that case, OpenOCD will issue the commands when it starts up the server.

We use the same setup as in Section 4.2.1 to test the interface. We connect to OpenOCD's server via the telnet port and issue the JTAG commands. In Listing 4.1, we manually set up the instruction register of ThunderX's DAP to run IDCODE and BYPASS instructions. Listing 4.2 replicates the Platform Cable test in Listing 3.1. It brings the state machine into the TEST-LOGIC-RESET state and then tries to output the contents of the test data register. OpenOCD passes the first test. It correctly prints the IDCODE of ThunderX's DAP (line 5). The input signal in BYPASS is delayed by a bit 0 on the output line (line 9). But the second test fails. Putting the state machine into RESET bugs OpenOCD. We suspect that OpenOCD expects the JTAG test logic to load the BYPASS operation by default upon reset, even if the IDCODE operation is available. Nevertheless, the JTAG interface of OpenOCD works, allowing us to interact with the DAP at a low level. If the boot problem (Section 4.2.2) can be resolved by configuring the DAP, this interface provides a viable way to achieve that.

---

```

1 ~$ telnet localhost 4444
2   . . . . .
3 > irscan auto0.tap 0xe
4 > drscan auto0.tap 64
   0x5555555555555555
5 555555551a101399
6 > irscan auto0.tap 0xf
7 > irscan auto0.tap 0xf
8 > drscan auto0.tap 8 0x55
9 aa

```

---

**Listing 4.1:** IDCODE and BYPASS

---

```

1 ~$ telnet localhost 4444
2   . . . . .
3 > pathmove RESET
4 > drscan auto0.tap 32
   0x00000000
5 Can't execute as the selected
   tap is in BYPASS
6 > irscan auto0.tap 0xe
7 > drscan auto0.tap 32
   0x00000000
8 1a101399

```

---

**Listing 4.2:** BYPASS failure

OpenOCD's auto-probing mechanism is similar to XSDB's `jtag targets` command. It runs automatically if the configuration file does not include the declaration of TAP devices on the scan chain. Unfortunately, the tool fails on Enzian's scan chain when multiple devices are set in normal (non-bypass) mode. Even though separately it correctly identifies the devices and their instruction register width.

Regarding the DAP, OpenOCD does not have an analog to CSAT's DAP enumeration. All DAPs on the chain must be declared manually with the `dap create` command, which assigns a DAP to a corresponding TAP controller. If the declaration is successful, one can inspect the ROM tables of Access Ports with `$dap_name info <ap index>`. (`$dap_name` is a placeholder for a name we assign to the DAP during its declaration) Additionally, the contents of various registers can be read with `$dap_name dpreg <register name>` and

## 4. EVALUATION

---

`$dap_name apreg <ap index> <register name>` where `<register name>` is one of the DAP registers such as debug base address (`baseaddr`), AP's ID (`apid`), or select AP (`apsel`). Alternatively, we can use numerical offsets into AP and DP register banks.

In Listing 4.3, we access the DAP registers (Lines 3-7) similarly to how we did it in Section 3.7 using the CoreSight Access Tool. We also print the contents of the ROM table for the first Access Port. OpenOCD fails to identify the type of the AP due to nonstandard type field entry in APIDR (Section 3.1.6). Using the DEVTYPE register fields (Section 2.3.4), OpenOCD correctly recognizes the page at address 0x88000000 as a debug register file of a processor core. However, it fails to identify ThunderX, possibly because its part number value in the PIDR's field is not publicly disclosed. (cf. Lines 25-28 of the ARM DS autodetection log in A.2). If we had not had access to the DSTREAM and ARM DS, we could have used the output of this command to set up the addresses in ThunderX's configuration file (Section 3.2.5).

---

```
1 ~$ telnet localhost 4444
2   . . . . .
3 > auto0.dap dpreg 0x0
4 0x0a111399
5 > auto0.dap apreg 0 0xfc
6 0x03990003
7 > dap info 0
8 AP # 0x0
9         AP ID register 0x03990003
10        Type is Unknown
11 MEM-AP BASE 0x00000003
12        Valid ROM table present
13   . . . . .
14        Component class is 0x1, ROM table
15        MEMTYPE system memory not present: dedicated debug bus
16        ROMTABLE[0x0] = 0x88000003
17        Component base address 0x88000000
18        Peripheral ID 0x03010cc20e
19        Designer is 0x1cc, Cavium Networks
20        Part is 0x20e, Unrecognized
21        Component class is 0x9, CoreSight component
22        Type is 0x15, Debug Logic, Processor
23        Dev Arch is 0x47706a15, ARM Ltd "Processor debug
24        architecture (v8.0-A)" rev.0
25   . . . . .
```

---

**Listing 4.3:** DAP interface

Note that compared to CSAT, OpenOCD does not permit write operations. Thus, if we want to configure the DAP in OpenOCD, we will have to use the JTAG interface.



## Chapter 5

---

# Conclusion

---

ARM DSTREAM, a debug adapter for ARM-based targets, was originally unable to interface with the debug infrastructure on ThunderX. That severely limited the debugging options for the Enzian processor. Our task was to locate and solve this problem. We started by interacting with the Debug Access Port on Enzian at the JTAG protocol level. For this, we initially employed the Platform Cable USB II, a JTAG adapter that exposes the JTAG communication interface to the user in the form of TCL commands. Aiming for more control over the execution, we designed our JTAG controller based on an STM32 Nucleo board. For simplicity, we implemented a static adapter that generated the signals according to sequences we passed to the Nucleo board. Still, the tool could drive the JTAG scan chains and communicate with the debug infrastructure behind the ARM DAP.

Neither Platform Cable USB II nor the Nucleo board managed to get any response from ThunderX's DAP. The actual cause of the problem turned out to be easily fixable. A missing jumper on Enzian's PCB prevented the TRST line from reaching the DAP. Thus, the JTAG test logic did not react to the signals sent from the DSTREAM unit. After we had closed the line with the jumper, ARM Development Studio successfully established the bare-metal debug session with ThunderX. We also used our Nucleo board to investigate faulty parts of the autodetection procedure responsible for reconstructing the debug topology on the chip. We identified that Cavium's implementation of the BYPASS instruction does not adhere to the standard, which leads to the autodetection incorrectly counting the devices on the chain. Using the working DSTREAM connection, we further explored the DAP infrastructure with the help of the CoreSight Access Tool.

Finally, we investigated an alternative to DSTREAM, a programming module JTAG-SMT3-NC already present on Enzian's board. The software that usually interacted with the module was unable to connect to ThunderX. We resorted to an open-source alternative, OpenOCD. We attached gdb to ThunderX's

cores by writing our configuration files for the JTAG adapter and the processor. Unfortunately, the tool encountered several issues compared to ARM DS debugging. Mainly because, in contrast to ARM DS, it does not know about the specifics of ThunderX and treats it as a generic ARMv8 CPU. Still, the insights we gained about that part of the infrastructure and JTAG-SMT3-NC, in particular, can be helpful for future work with the module.

We propose several ideas for future projects that build upon our work or tackle the issues we were unable to solve due to time constraints.

**STM32 JTAG Adapter Optimization.** Since the adapter's speed was irrelevant to our work, we prioritized the program's simplicity and did not perform any optimizations. For example, we stored each bit of the input and control signals as a separate byte in the array. Employing a more compact way to store the signals should decrease the total amount of memory accesses and speed up the execution.

**Sequence compiler.** Even though writing signal sequences by hand was fine for simple scan chain tests, the example in Section 4.1.3 shows that anything more complicated dramatically increases the complexity of the sequences. We can simplify the process by implementing a compiler that would transform XSDB or CSAT-like scripts into bit sequences. The compiler could start by incorporating the commands similar to the `jtag` sequence interface. This interface can, in turn, be used to implement separate JTAG-level instructions (IDCODE, BYPASS, DPACC, and APACC). The last step would be to provide register and memory accesses of the DAP. One application of this interface could be halting a ThunderX's core in a manner similar to how CSAT does it for an Armv7 platform [5].

**Boot problem.** For any further work with the onboard JTAG controller, solving this issue is paramount. Without it, we cannot leave the CPU's jumper configuration header in normal mode on the daisy chain. Since the exception is thrown on the read access, one hypothesis is that the DAP locks the bus for itself and that the problem could be solved by resetting the DAP and then restarting the CPU from the BMC's console. We connect the Nucleo board to the CPU's JTAG header, cause ThunderX to crash, and try to reset the DAP by asserting the Debug reset request bit CDBGRESTREQ of the CTRL/STAT register (The reset procedure is described in Section B2.4 of the ADI Specification [8]). Unfortunately, it fails. When we read the CTRL/STAT register to get the response information, the Debug reset acknowledge bit CDBGSTACK is always 0 (Figure 5). According to the ADI specification, this strongly suggests that this reset procedure of the DAP is not supported by ThunderX.

**FTDI probe & ARM DS.** ARM Development Studio can work with third-party debug probes. In particular, it provides support for FTDI converters. ARM DS running on a computer connected to JTAG-SMT3-NC through the

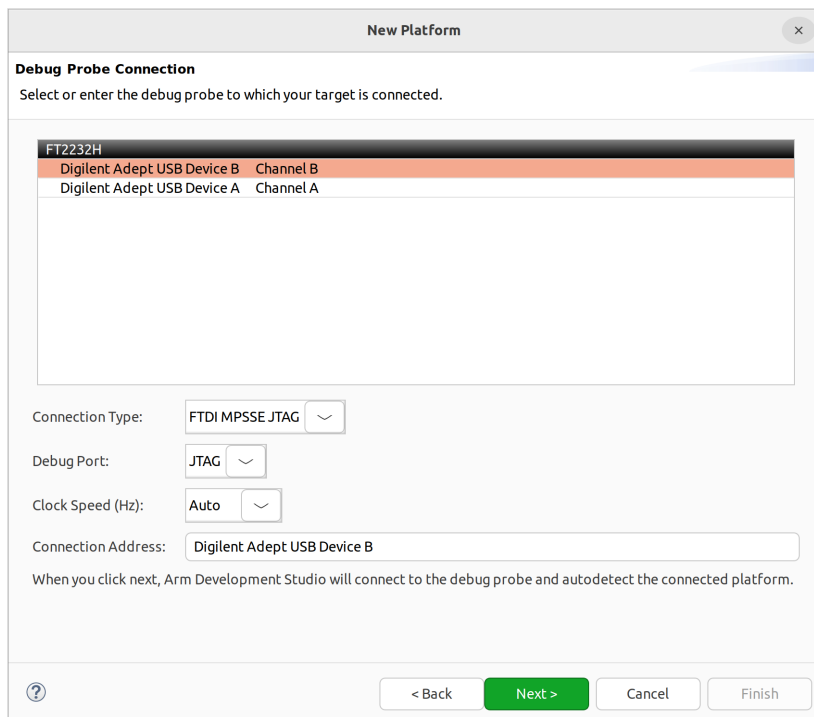
```

CTRL/STAT write: 00000000000000000000000000000101010
CTRL/STAT read: 00000000000000000000000000000101111

```

**Figure 5.1:** 32 bits of CTRL/STAT. The left most bit is the least significant bit in the register. The 27th bit is debug reset request CDBGIRSTREQ. The 28th bit is debug reset acknowledge CDBGIRSTACK (highlighted in bold)

USB cable successfully identifies the adapter in the Connection Browser window (Figure 5.2), and we can use it for the autodetection procedure. (Note that for Linux computers, you might need to load another FTDI driver; see Section 10.3 in the ARM DS Getting Started Guide [10]) Unfortunately, ARM DS fails to drive the adapter, and the JTAG signals coming from the JTAG-SMT3-NC module remain unchanged throughout the autodetection. We think the reason for this is that the JTAG-SMT3-NC module features additional signals that are not part of the standard FTDI device pinout. For JTAG-SMT3-NC, this is the JTAG enable OEJTAG signal that has to be set to 1 (Section 3.2.4). If there is a way to do this in ARM DS, it would eliminate the need for OpenOCD and resolve all debug session issues caused by OpenOCD’s lack of familiarity with ThunderX.



**Figure 5.2:** ARM DS identifies the FTDI converter



26 [22/07/24 11:22:37] This is likely to be related to the probe mode  
 being incorrectly set to SWD.  
 27 [22/07/24 11:22:37] Halting detection - bad AP configuration detected  
 when enumerating APs.

---

## A.2 Successful autodetection log

The session log was edited for length. Missing parts are indicated by five dots.

---

```

1 [17/04/24 14:53:13] --- --- ---
2 [17/04/24 14:53:13] Arm Development Studio v2023.1, build number
   202310907
3 [17/04/24 14:53:13] JTAG Clock Speed : Auto
4 [17/04/24 14:53:13] Beginning Autodetection
5 [17/04/24 14:53:13] --- --- ---
6 [17/04/24 14:53:13] Counting devices:
7 [17/04/24 14:53:13] DR Chain [136]:
8     1111111111111111111111111111111111111111111111111111111111111111
   1111111111111111111111111111111111111111111111111111111111111111
9 [17/04/24 14:53:13] Device Count: 0
10 [17/04/24 14:53:13] Failed to detect scanchain devices
11     . . . . .
12 [17/04/24 14:53:14] Enumerating AP devices for DAP at scanchain index
   0:
13 [17/04/24 14:53:14] Number of AP buses detected: 2
14 [17/04/24 14:53:14] AP types:
15 [17/04/24 14:53:14]   APB-AP
16 [17/04/24 14:53:14]   APB-AP
17 [17/04/24 14:53:14] --- --- ---
18 [17/04/24 14:53:15] --- --- ---
19 [17/04/24 14:53:15] Looking for ROM tables on AP0 (APB-AP)
20 [17/04/24 14:53:15] APB-AP ROM table base address detected as
   0x00000003
21 [17/04/24 14:53:15]   Reading ROM table for AP index 0, base address
   = 0x00000003
22 [17/04/24 14:53:16]           Component ID registers:
23 [17/04/24 14:53:16]           CID 0: 0x0d
24     . . . . .
25 [17/04/24 14:53:16]   Reading peripheral and component ID registers
   of device at address 0x88000000
26     . . . . .
27 [17/04/24 14:53:16]           Peripheral ID = 0x20e, JEP-106 code,
   including continuation = 0x34c, DEVTYPED = 0x15, DEVARCH = 0x6a15,
   Revision = 0x0
28 [17/04/24 14:53:16]   ThunderX-r2 found at address 0x88000000
29 [17/04/24 14:53:16]   Reading peripheral and component ID registers
   of device at address 0x88010000
  
```

## A. APPENDIX

---

```

30      . . . . .
31 [17/04/24 14:53:16] CSCTI found at address 0x88010000
32      . . . . .
33 [17/04/24 14:53:17] ThunderX-TRC found at address 0x8FFB0000
34 [17/04/24 14:53:17] End of ROM table
35 [17/04/24 14:53:17] --- --- ---
36 [17/04/24 14:53:17] Looking for ROM tables on AP1 (APB-AP)
37 [17/04/24 14:53:17] APB-AP ROM table base address detected as
38     0x00000003
39 [17/04/24 14:53:17] Reading ROM table for AP index 1, base address
40     = 0x00000003
41 [17/04/24 14:53:17] Failed to read ROM table: Failed to read 16
42     bytes from address 0x00000FF0 on CSMEMAP_1
43      . . . . .
44 [17/04/24 14:53:19] --- --- ---
45 [17/04/24 14:53:20] Autodetection Complete
46 [17/04/24 14:53:39] Creating database entry...
47 [17/04/24 14:53:39] Platform "Systems Group - Enzian 17 04 2024"
48     built successfully

```

---

### A.3 MEM-AP memory read

The setup of the experiment is the same as in Section 4.1.3. This time, we issue a 328-bit series of commands. Apart from the CTRL/STAT setup at the beginning of the execution (Steps 1 and 2), we also must configure the Control/Status Word register (CSW) register that manages the AP's memory transactions into the associated address space (Step 3). The rest is the memory access itself. We pass the address to the Transfer Address Register (Step 4), request the memory read through the Data Read/Write register (Step 5), and read the result (Step 6).

#### 1. SELECT.DPBANKSEL 0 write.

```

TDI: 00000 00000 0101 0000 001 00000000000000000000000000000000
TDO: 11111 11111 1000 1111 100 00000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 00000000000000000000000000000001

```

#### 2. CTRL/STAT write.

```

TDI: 00000 00000 0101 0000 010 0000000000000000000000000001010
TDO: 11111 11111 1000 1111 010 00000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 00000000000000000000000000000001

```

#### 3. CSW write. Every AP register we will access is located in the first

register bank. The APBANKSEL field of the SELECT register is already set to 0b0000 in the first step. All that is left to do is to change the offset value in the control bits of the APACC instruction. CSW is the first register in the bank (offset 0b00). The new content of the CSW register ensures that the device is on and sets the standard parameters for the memory transaction. For example, we choose 32-bit memory access (bits [2:0]) and enable the Access Port (bit [6]).

```
TDI: 00000 00000 1101 0000 000 010000100000000000000000000000000000000001
TDO: 11111 11111 1000 1111 010 000000000000000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 000000000000000000000000000000000000000001
```

4. **TAR write 0x80000040.** 32-bit TAR is the second register in the bank (offset 0b01). The data field contains our target address 0x80000040.

```
TDI: 00000 00000 1101 0000 010 000000100000000000000000000000000000000001
TDO: 11111 11111 1000 1111 010 000000000000000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 000000000000000000000000000000000000000001
```

5. **DRW read.** Recall that APACC read operations to the DRW register initiate a memory read from the address held in the TAR register. DRW is the fourth register in the bank.

```
TDI: 00000 00000 1101 0000 111 000000000000000000000000000000000000000000
TDO: 11111 11111 1000 1111 010 000000000000000000000000000000000000000000
TMS: 11111 01100 0001 1100 000 000000000000000000000000000000000000000001
```

6. We latch the result onto the scan chain by navigating the state machine into CAPTURE-DR. It is equal to CSAT's output 0x88290003 of the dmr 0 0x80000040 1 command.

```
TDI: 00000000000000 00 000 0000000000000000000000000000000000000000000000
TDO: 11111111111111 11 010 11000000000000001001010000010001
TMS: 10000000000001 00 000 0000000000000000000000000000000000000000000001
```

## A.4 Register read failure in OpenOCD

---

```
1 ~$ gdb-multiarch
2 GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
3 . . . . .
4 (gdb) target extended-remote localhost:3333
```

## A. APPENDIX

---

```
5      . . . . .
6 0xffff80008163d6dc in ?? ()
7 (gdb) display/2i $pc
8 1: x/2i $pc
9 => 0xffff80008163d6dc: nop
10    0xffff80008163d6e0: mov    x0, #0x0                // #0
11 (gdb) ni
12 core1 halted in AArch64 state due to debug-request, current mode: EL1H
13 cpsr: 0x600000c5 pc: 0xffff80008163d6dc
14 MMU: enabled, D-Cache: enabled, I-Cache: enabled
15 0xffff80008163d6e0 in ?? ()
16 1: x/2i $pc
17 => 0xffff80008163d6e0: mov    x0, #0x0                // #0
18    0xffff80008163d6e4: mov    x1, #0x0                // #0
19 (gdb) info registers
20 x0          0x0          0
21 x1          0x0          0
22      . . . . .
23 x30         0xffff80008163d73c 18446603338392000316
24 sp          0xffff800083213d50 0xffff800083213d50
25 pc          0xffff80008163d6dc 0xffff80008163d6dc
26 cpsr       0x600000c5      [ SP=1 EL=1 nRW=0 F I C Z ]
27 fpsr       0x10           16
28 fpcr       0x0            0
29 ELR_EL1    0x0            0x0
30 ESR_EL1    0x96000004      2516582404
31 SPSR_EL1   0x600003c5      1610613701
32 Could not fetch register "ELR_EL2"; remote failure reply 'EOE'
```

---



## A.5 OpenOCD to STM32 connection log

---

```

1  ~$ openocd -f
   board/st_nucleo_g0.cfg
2  . . .
3  Info : The selected transport
   took over low-level target
   control. The results might
   differ compared to plain
   JTAG/SWD
4  srst_only separate srst_nogate
   srst_open_drain
   connect_deassert_srst
5  Info : Listening on port 6666
   for tcl connections
6  Info : Listening on port 4444
   for telnet connections
7  . . .
8  Info : [stm32g0x.cpu]
   Examination succeed
9  . . .
10 Info : Listening on port 3333
   for gdb connections
11 Info : accepting 'gdb'
   connection on tcp/3333
12 [stm32g0x.cpu] halted due to
   debug-request, current
   mode: Thread xPSR:
   0x61000000 pc: 0x080043c6
   msp: 0x20008fa8
13 Info : device idcode =
   0x20006460 (STM32G07/G08xx
   - Rev B : 0x2000)
14 Info : halted: PC:
   0x080043c8000

```

---

**Listing A.1:** OpenOCD server pane

---

```

1  ~$ gdb-multiarch
2  . . .
3  (gdb) target extended-remote
   localhost:3333
4  Remote debugging using
   localhost:3333
5  warning: No executable has
   been specified and target
   does not support
6  determining executable
   automatically. Try using
   the "file" command.
7  0x080043c6 in ?? ()
8  (gdb) display/3i $pc
9  1: x/3i $pc
10 => 0x80043c6: ldr    r3, [r7,
   #24]
11    0x80043c8: adds   r3, #1
12    0x80043ca: beq.n  0x800446a
13  (gdb) ni
14  halted: PC: 0x080043c8
15  0x080043c8 in ?? ()
16  1: x/3i $pc
17  => 0x80043c8: adds   r3, #1
18    0x80043ca: beq.n  0x800446a
19    0x80043cc: bl     0x8001468

```

---

**Listing A.2:** GDB pane

---

## Bibliography

---

- [1] Arm Limited. *CoreSight Components Technical Reference Manual*, H edition, 2009. Available at <https://developer.arm.com/documentation/ddi0314/>.
- [2] Arm Limited. *ARM DSTREAM System and Interface Design Reference*, 2011. Available at <https://developer.arm.com/documentation/dui0499/>.
- [3] Arm Limited. White Paper: CoreSight Technical Introduction. A quick-start for designers. Technical report, 2013. Document Number: ARM-EPM-039795.
- [4] Arm Limited. *ARM DS-5 ARM DSTREAM User Guide*, K edition, 2015. Available at <https://developer.arm.com/documentation/dui0481/>.
- [5] Arm Limited. *Low Level Debug using CSAT on Armv7-based platforms*, 0100-02 edition, 2019. Available at <https://developer.arm.com/documentation/102715/0100/>.
- [6] Arm Limited. *How PCE identifies the CoreSight components on the target board*, 1.0 edition, 2021. Available at <https://developer.arm.com/documentation/102582/0100/How-does-PCE-detect-the-information>.
- [7] Arm Limited. *Arm Coresight Architecture Specification v3.0*, F edition, 2022. Available at <https://developer.arm.com/documentation/ih0029/>.
- [8] Arm Limited. *Arm Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*, G edition, 2022. Available at <https://developer.arm.com/documentation/ih0031/>.
- [9] Arm Limited. *CoreSight Access Tool (CSAT) User Guide*, 2.6.0 edition, 2022. Available at <https://developer.arm.com/documentation/epm051792/>.

- 
- [10] Arm Limited. *Arm Development Studio Getting Started Guide*, 2024.0-00 edition, 2024. Available at <https://developer.arm.com/documentation/101469/>.
- [11] Arm Limited. The JTAG IDCODE for a Cortex processor, KBA Article ID: KA001235, 2024. Online. Available at <https://developer.arm.com/documentation/ka001235/>. Accessed 2024-08-03.
- [12] Cavium. *Cavium ThunderX CN88XX Hardware Reference Manual*, 0.965E edition, 2015.
- [13] Cavium. *Cavium ThunderX CN88XX, Pass 2 Hardware Reference Manual*, 2.7P edition, 2017.
- [14] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: an open, general, CPU/FPGA platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 434–451, New York, NY, USA, 2022. Association for Computing Machinery.
- [15] Digilent. *JTAG-SMT3-NC Reference Manual*, 2021. Available at [https://digilent.com/reference/\\_media/reference/programmers/jtag-smt3/jtag-smt3-nc-rm.pdf](https://digilent.com/reference/_media/reference/programmers/jtag-smt3/jtag-smt3-nc-rm.pdf).
- [16] Digilent. *JTAG-SMT2-NC Reference Manual*, 2023. Available at [https://digilent.com/reference/\\_media/programmers/jtag-smt2-nc/jtag-smt2-nc\\_rm.pdf](https://digilent.com/reference/_media/programmers/jtag-smt2-nc/jtag-smt2-nc_rm.pdf).
- [17] DirtyJTAG. DirtyJTAG. JTAG probe firmware, 2022. Online. Available at <https://github.com/dirtyjtag/DirtyJTAG>. Accessed 2024-08-03.
- [18] Felix Domke. *Blackbox JTAG Reverse Engineering*. 2009.
- [19] Enclustra GmbH. *Mercury XU5 SoC Module User Manual*, 08 edition, 2021.
- [20] Enzian Team. The Enzian Research Computer; Schematics, April 2022. Available at <https://doi.org/10.5281/zenodo.6465908>.
- [21] FTDI Limited. *Application Note AN\_135 FTDI MPSSE Basics*, 1.1 edition, 2010. Available at [https://www.ftdichip.com/Documents/AppNotes/AN\\_135\\_MPSSE\\_Basics.pdf](https://www.ftdichip.com/Documents/AppNotes/AN_135_MPSSE_Basics.pdf).

- [22] FTDI Limited. *FT232H single channel hi-speed USB to multipurpose UART/FIFO IC Datasheet*, 2.1 edition, 2023. Available at [https://ftdichip.com/wp-content/uploads/2023/09/DS\\_FT232H.pdf](https://ftdichip.com/wp-content/uploads/2023/09/DS_FT232H.pdf).
- [23] FTDI Limited. *FT2232H Dual High Speed USB to Multipurpose UART/FIFO IC Datasheet*, 2.8 edition, 2024. Available at [https://ftdichip.com/wp-content/uploads/2024/05/DS\\_FT2232H.pdf](https://ftdichip.com/wp-content/uploads/2024/05/DS_FT2232H.pdf).
- [24] IEEE. IEEE Standard for Test Access Port and Boundary-Scan Architecture. *IEEE Std. 1149.1-2013*, 2013.
- [25] JColvin. Pin Mapping for JTAG-SMT3-NC. Digilent Forums, 2019. Online. Available at <https://forum.digilent.com/topic/17752-pin-mapping-for-jtag-smt3-nc/?do=findComment&comment=45305>. Accessed 2024-08-03.
- [26] jpeyron. Pin mapping for JTAG-SMT2-NC? Digilent Forums, 2017. Online. Available at <https://forum.digilent.com/topic/4745-pin-mapping-for-jtag-smt2-nc/#comment-19222>. Accessed 2024-08-03.
- [27] Arm Limited. How to configure debug and trace, Article ID: KA001452, 2024. Online. Available at <https://developer.arm.com/documentation/ka001452/>. Accessed 2024-08-03.
- [28] OpenOCD. Open On-Chip Debugger, 2023. Online. Available at <https://openocd.org/>. Accessed 2024-08-03.
- [29] OpenOCD. *Open On-Chip Debugger: OpenOCD User's Guide*, 0.12.0+dev edition, 2024. Available at <https://openocd.org/doc/pdf/openocd.pdf>.
- [30] Dominic Rath. Open on-chip debugger. Diploma thesis, University of Applied Sciences Augsburg, Augsburg, 2005.
- [31] STMicroelectronics. *Description of STM32G0 HAL and low-layer drivers, UM2319 User manual*, 2nd edition, 2020. Available at [https://www.st.com/resource/en/user\\_manual/um2319-description-of-stm32g0-hal-and-lowlayer-drivers-stmicroelectronics.pdf](https://www.st.com/resource/en/user_manual/um2319-description-of-stm32g0-hal-and-lowlayer-drivers-stmicroelectronics.pdf).
- [32] STMicroelectronics. How to use the STM32CubeIDE terminal to send and receive data, 2024. Online. Available at <https://community.st.com/t5/stm32-mcus/how-to-use-the-stm32cubeide-terminal-to-send-and-receive-data/ta-p/49434>. Accessed 2024-08-03.
- [33] STMicroelectronics. *STM32 Nucleo-64 boards (MB1360), UM2324 User manual*, 5th edition, 2024. Available at <https://www.st.com/resource/>

- [en/user\\_manual/um2324-stm32-nucleo64-boards-mb1360-stmicroelectronics.pdf](#).
- [34] STMicroelectronics. STM32Cube - discover the STM32Cube ecosystem, 2024. Online. Available at [https://www.st.com/content/st\\_com/en/ecosystems/stm32cube-ecosystem.html](https://www.st.com/content/st_com/en/ecosystems/stm32cube-ecosystem.html). Accessed 2024-08-03.
- [35] STMicroelectronics. *STM32CubeIDE user guide, UM2609 User manual*, 12th edition, 2024. Available at [https://www.st.com/resource/en/user\\_manual/um2609-stm32cubeide-user-guide-stmicroelectronics.pdf](https://www.st.com/resource/en/user_manual/um2609-stm32cubeide-user-guide-stmicroelectronics.pdf).
- [36] STMicroelectronics. *STM32CubeMX for STM32 configuration and initialization C code generation, UM1718 User manual*, 45th edition, 2024. Available at [https://www.st.com/resource/en/user\\_manual/um1718-stm32cube-mx-for-stm32-configuration-and-initialization-c-code-generation-stmicroelectronics.pdf](https://www.st.com/resource/en/user_manual/um1718-stm32cube-mx-for-stm32-configuration-and-initialization-c-code-generation-stmicroelectronics.pdf).
- [37] Michael Williams. *Low Pin-count Debug Interfaces for Multi-device Systems*. 2009.
- [38] Xilinx. *Platform Cable USB II Data Sheet*, 1.5.1 edition, 2018. Available at <https://docs.amd.com/v/u/en-US/ds593>.
- [39] Xilinx. *Xilinx Software Command-Line Tool (XSCT)*, UG1208 edition, 2018. Available at <https://docs.amd.com/v/u/2018.2-English/ug1208-xsct-reference-guide>.

## Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies<sup>1</sup>.
- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies<sup>2</sup>.
- I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies<sup>3</sup>. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

A tool for debugging JTAG

Authored by:

*If the work was compiled in a group, the names of all authors are required.*

Last name(s):

Memetov

First name(s):

Edem

With my signature I confirm the following:

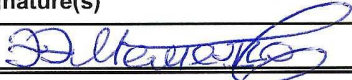
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date

Altendorf, 05.08.2024

Signature(s)



*If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.*

<sup>1</sup> E.g. ChatGPT, DALL E 2, Google Bard

<sup>2</sup> E.g. ChatGPT, DALL E 2, Google Bard

<sup>3</sup> E.g. ChatGPT, DALL E 2, Google Bard