Diving into

JTAG

Aliaksandr Kavalchuk

Diving into JTAG

A Comprehensive Guide to Debugging, Testing, and Securing Embedded Systems with JTAG Protocol

Aliaksandr Kavalchuk

This book is available at https://leanpub.com/divingintojtag

This version was published on 2024-11-09

* * * * *

* * * * *

Table of Contents

Table of Contents

Intro

This book is a collection of my articles originally published in the blogs Interrupt by Memfault and PlatformIO with some additional edits and enhancements.

Who Is This Book For?

This book is designed for engineers, developers, and technology enthusiasts who are eager to explore the intricacies of JTAG (Joint Test Action Group) technology. Whether you're a seasoned professional or a curious beginner, this book will provide valuable insights and practical knowledge.

Here's a breakdown of the target audience:

- **Embedded Systems Engineers**

  If you work with microcontrollers, processors, or SoCs and need to debug, test, or program these devices, this book will serve as an essential resource.

- **Hardware Engineers**

  For those designing PCBs or testing hardware prototypes, understanding JTAG can simplify the troubleshooting process and ensure robust designs.

- **Firmware Developers**

  If your responsibilities include low-level programming or debugging firmware, this book will help you unlock the full potential of JTAG for development and debugging tasks.

- **Test Engineers**

Professionals involved in manufacturing and quality assurance will benefit from learning how to use JTAG for boundary scan testing and production testing of electronic devices.

- **Educators and Students**

  For academics and learners studying embedded systems, computer architecture, or hardware design, this book offers foundational knowledge and hands-on exercises to deepen your understanding.

- **Hobbyists and Makers**

  If you enjoy tinkering with hardware, reverse engineering, or creating custom electronics, JTAG can open new possibilities for exploration and creativity.

- **Security Researchers**

  For those delving into hardware security, JTAG can serve as a powerful tool for examining vulnerabilities and ensuring device integrity.

This book assumes a basic understanding of electronics and programming but does not require prior experience with JTAG. The material is structured to guide readers from fundamental concepts to advanced applications, making it a versatile resource for a wide range of skill levels.

How to read this book

A unique feature of this book is the extensive use of GIF animations and videos to visually demonstrate key concepts and processes. However, since not all EPUB readers support GIF and video playback, each animation and video is accompanied by a link to an external resource. Readers can follow these links to view the corresponding content and enhance their understanding.

Repo with all GIF animation: https://github.com/Zamuhrishka/diving-into-jtag-book-animations

Feedback

This book is complete, but as with any technical work, it may still contain errors or sections that could be clarified further. While I have made every effort to ensure accuracy and readability, English is not my native language, and some imperfections may remain.

If you encounter any mistakes or areas that are unclear, or if you have suggestions for improving the content, I would greatly appreciate your feedback. Your insights will help ensure that this book remains a valuable resource for everyone exploring JTAG technology.

You can reach me on Linkedin

Support

If you want to help me, you may provide any feedback or review using your favorite social network or blog.

1. Overview

JTAG (Joint Test Action Group) is a specialized hardware interface based on the IEEE 1149.1 standard. This interface is designed to connect complex chips and devices to standard test and debugging hardware.

Nowadays JTAG is mainly used for:

- Output control of microcircuits
- Testing of printed circuit boards
- Flashing of microchips with memory
- Chip software debugging

The testing method implemented in the standard is called **Boundary Scan**. The name reflects the idea of the process: Functional blocks within the chip are isolated, and specific signal combinations are applied to their inputs. The state of each block's output is then evaluated. The whole process is performed by special commands via the JTAG interface, and no physical intervention is required.

1.1 Test Access Point (TAP)

The Test Access Port (TAP) is one of the key elements of the JTAG protocol designed to control and configure chips connected to the JTAG chain.

The TAP operates as a simple finite-state machine that is controlled by the `TMS` (Test Mode Select) signal. It allows access to the internal registers of microcontrollers and other devices through JTAG commands.

Each device connected to the JTAG chain has its own TAP, which consists of the `IR` (Instruction Register) and DR (Data Register) registers. The `IR` the register is used to select the instruction to be executed on the device and the `DR` the register is used to transfer data.

1.2 Control signals

The TAP comprises four mandatory signals (TCK, TMS, TDI, TDO) and one optional signal (TRST).

- TDI (Test Data Input) — test data input. The commands and data are inserted into the chip through this pin on the rising edge of the signal TCK.
- TDO (Test Data Output) — serial data output. Commands and data are output from the chip through this pin on the falling edge of the signal TCK.
- TCK (Test Clock) — clock input.
- TMS (Test Mode Select) — controls the transitions between states of the finite state machine TAP.
- TRST (Test Reset) — reset signal of the TAP finite state machine.

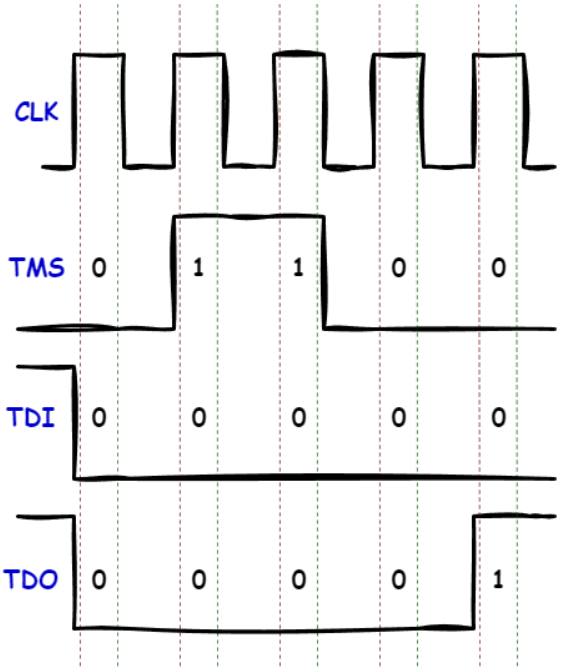


Figure 1.1 - JTAG Control Signals

The standard is that the JTAG module reads data from the TMS and TDI lines on the rising edge of the TCK line. The JTAG module in any chip must also change the logic value on the TDO line on the falling edge of TCK. In the diagram below, the moments of reading

data by the JTAG module are shown with the red dotted line and the moments of writing data are shown with the green dotted line.

1.3 Registers

The TAP state machine allows access to two special registers, the `IR` (Instruction Register), and a symbolic register called `DR` (Data Register).

The instruction registers store the current instruction to be executed. The value of this register is used by the TAP controller to decide what to do with incoming signals. The most commonly used instruction specifies which data register the incoming data should go into.

The Data Register is a placeholder for the register that is currently selected with the current content of the `IR`. Thus, `IR` is an index into a number of registers, and `DR` is the currently selected register.

There are three main types of data registers:

- `BSR` (Boundary Scan Register) — the main register for testing. It is used to transfer data to and from the pins of the chip.
- `BYPASS` — single-bit register that transfers data from `TDI` to `TDO`. It allows testing other chips connected in series with minimum delays.
- `IDCODES` — stores the ID code and revision number of the chip.

In the picture below you can see an approximate illustration of the principle of operation of the `DR` register: the switchers **SW3** and **SW4** choose the current register depending on the instruction in `IR`.

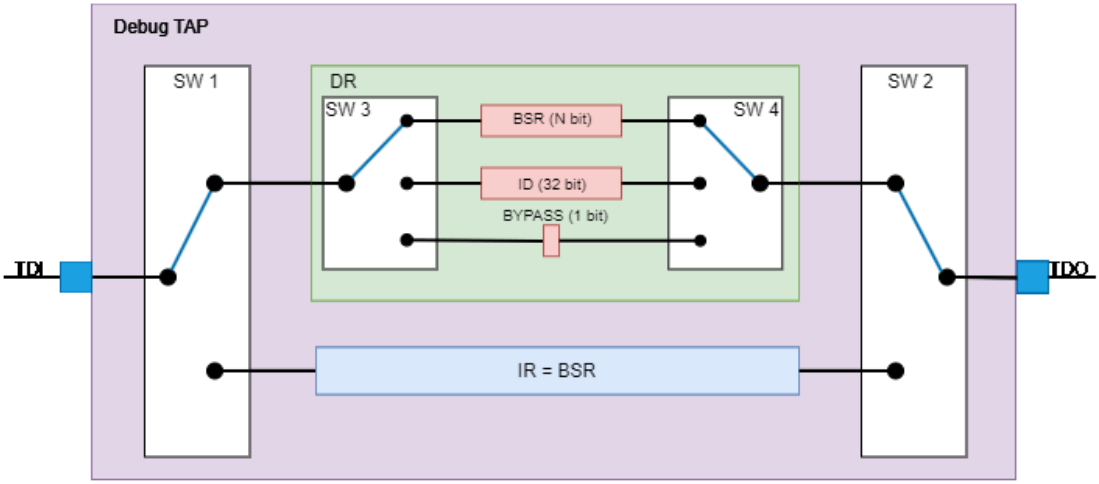


Figure 1.2 - Selection DR by Code in IR

The size of the `IR` is implementation specific and usually varies between 4 and 32 bits. Since the selected register is directly accessed when scanning `DR`, the size `DR` depends on the current instruction.

JTAG registers are an important part of the microcontroller debugging process because they allow you to control and monitor the state of the microcontroller during program execution. Each microcontroller manufacturer may use its own JTAG registers, so you should consult the documentation of your particular microcontroller for details on the JTAG registers it supports.

1.3.1 Shift register

Data transfer (read/write) in the JTAG protocol is performed by the shift register principle. In a shift register, the data is transferred sequentially, bit by bit, one per clock cycle.

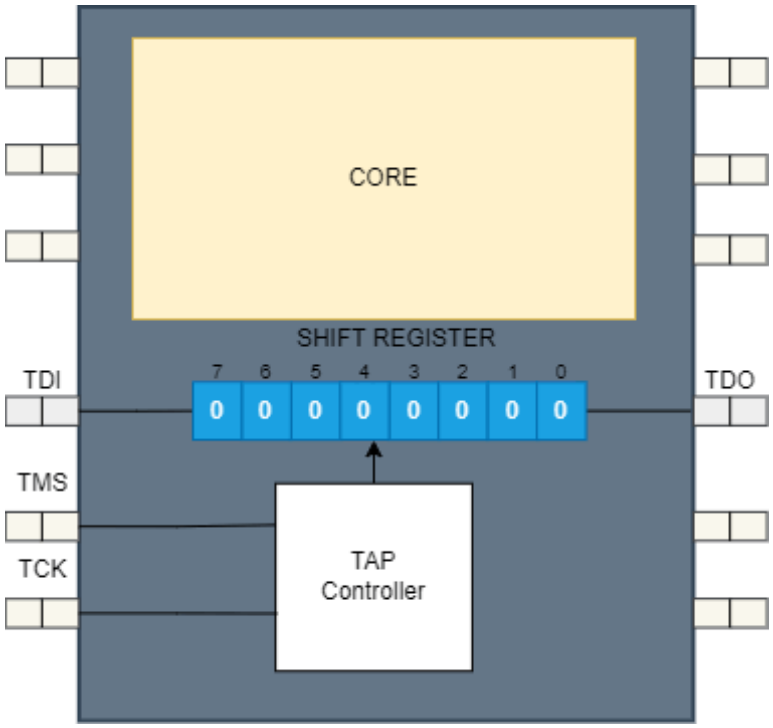


Figure 1.3 - Shift Register

This register is located between the `TDI` and `TDO` pins and is used to receive information from the `TDI` pin and output information to the `TDO` pin. Every time you want to write something to the TAP via JTAG protocol - you set the necessary signals to the `TDI` pin - these signals are synchronously written to the shift register starting from the highest bit and gradually moving to the lowest bit of the register with each new clock, and the value of the lowest bit of the shift register with each clock is moved to the `TDO` pin, from which we can read it.

In fact, JTAG can indeed be considered as a slightly modified SPI protocol using a daisy-chain configuration. In JTAG, `TDI` corresponds to `MOSI`, `TDO` to `MISO`, and `TCK` serves as `CLK`. The main differences are the presence of the `TMS` signal, which controls the TAP state machine, and the absence of a Chip Select signal.

By the way, if you want to dive deeper into various wired protocols used in embedded systems, such as 1-Wire, UART, SPI, I2C, I3C, CAN, USB, and more, I recommend my book Wired Protocols in Embedded Systems

1.4 JTAG Instruction

JTAG instructions are commands that interact with the TAP, enabling test, debug, programming, and configuration functions.

Let's look at some of the most common instructions.

1.4.1 The `IDCODE` instruction

The `IDCODE` instruction in JTAG is used to get the unique identifier of the device connected to the JTAG circuit. Each device that supports JTAG has its own unique **ID** code, which can be read using the `IDCODE` command. This can be useful to identify the device type, manufacturer, and version.

This identifier is 32-bit in size and consists of next fields:

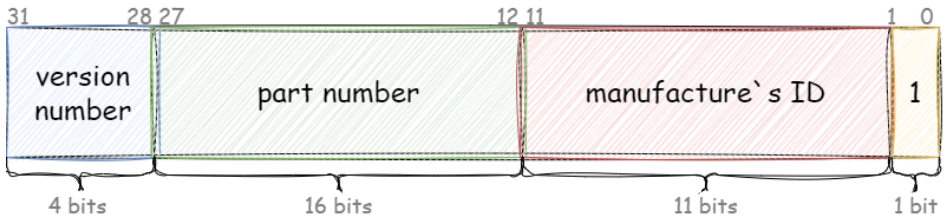


Figure 1.4 - Chip ID Format

- **Manufacturer ID**: Defines the manufacturer of the chip. The manufacturer code is assigned according to the JEDEC standard.
- **Part Number**: A unique identifier for the chip model.
- **Version Number**: Used to specify the version of the device or the chip revision.

So when you load the `IDCODE` instruction into the `IR` register, it will force the `IDCODE` register to be selected as the data register. However, after power-on, it is not even necessary to load the `IDCODE` instruction into the `IR` register, as it will already be loaded by default. Consequently, the register containing the ID code of the chip is selected as the `DR` register immediately after power-on.

1.4.2 The Boundary Scan instructions

This is a set of commands that includes the following:

- `EXTEST`: Used to test connections between chips on a board.
- `INTEST`: Used to test the internal logic of a chip.
- `SAMPLE`: Allows reading the current states of pins without modifying them.
- `PRELOAD`: Used to load values before activating other commands, such as `EXTEST`.

These commands are used to control and test the inputs/outputs of a chip via a special register called the **Boundary Scan Register (BSR)**.

The **Boundary Scan Register (BSR)** is a chain of flip-flops associated with each pin of the chip.

This register allows you to:

- Read the current states of the pins.
- Control the states of the pins for testing purposes.

In very simplified terms, the process of working with Boundary Scan commands can be described as follows: if you want to read the state of a chip's pins, you need to load the `SAMPLE` command into the `IR` register. After that, the boundary scan register will connect to the scan chain as the data register, and you can read this register, where each bit will reflect the state of the corresponding pin of the chip.

1.4.2 The `BYPASS` instruction

The **BYPASS** command is used to simplify interaction with the JTAG chain when one or more devices are not involved in the current testing or programming process. This command minimizes delays by skipping unnecessary data registers (Data Registers, DR) through a special one-bit register.

When the BYPASS instruction is passed to a JTAG chain, it skips the targeted device and passes control to the next device in the chain. Thus, the BYPASS command avoids addressing a device that cannot be scanned by the JTAG protocol and allows scanning of devices further down the chain.

Let's consider the usefulness of this command with an example. Suppose there are two microcontrollers in the JTAG chain: DEV1 and DEV2. We need to test microcontroller DEV1. To do this, we write one of the Boundary Scan commands to the IR register of DEV1, thereby connecting the BSR register as the data register. Meanwhile, microcontroller DEV2 still has its default data register (ID) active:

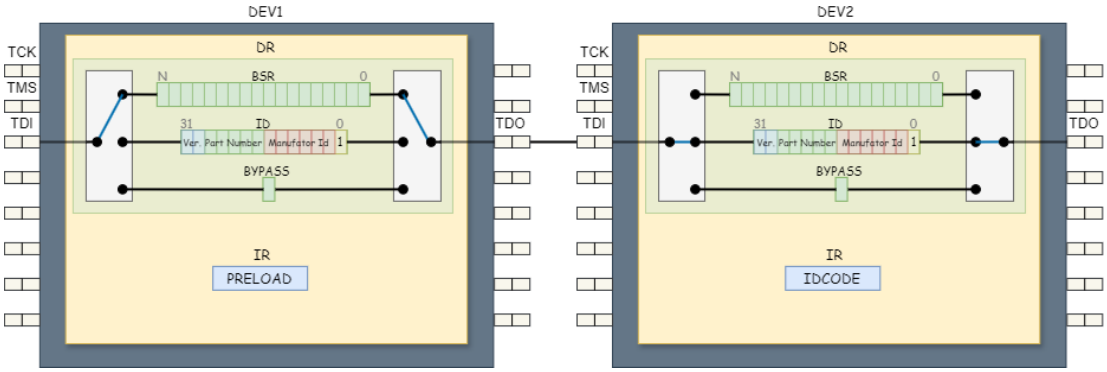


Figure 1.5 - The JTAG chain without the BYPASS usage

In this case, the scan chain consists of the BSR register, whose size roughly equals the number of pins of the microcontroller (let's assume 64 bits), and the ID register, which is 32 bits wide. Without the BYPASS command, to read the BSR register from microcontroller DEV1, you would need to read 64 + 32 = 96 bits from the JTAG chain. However, thanks to the BYPASS command and its corresponding 1-bit data register, the number of bits to read can be reduced to 1 + 64 = 65. To achieve this, the BYPASS command should be written to the IR register of microcontroller DEV2:

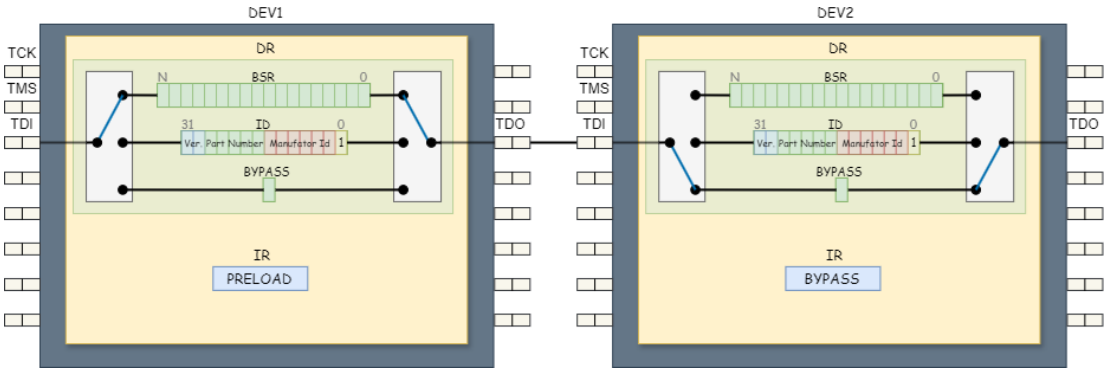


Figure 1.6 - The JTAG chain with the BYPASS usage

Thus, the `BYPASS` command is extremely useful when working with a JTAG chain that contains multiple TAPs.

1.5 TAP State Machine

The JTAG protocol's finite state automaton comprises a set of states that the TAP can assume, depending on the signals received at its inputs. Each state corresponds to a specific combination of signal values for the `TMS` and `TDI` inputs.

The transitions between states depend on the `TMS` signal at the moment of rising level of `TCK`.

The initial state after resetting is `Test-Logic-Reset`. As defined by the standard the LSB is pushed in and pulled out first for all shift registers.

The State Machine is quite simple and has two ways of working:

- *Instruction register selection* (blue blocks) is used to select the current command.
- *Data register selection* (green blocks) is used to read/write data into the data registers.

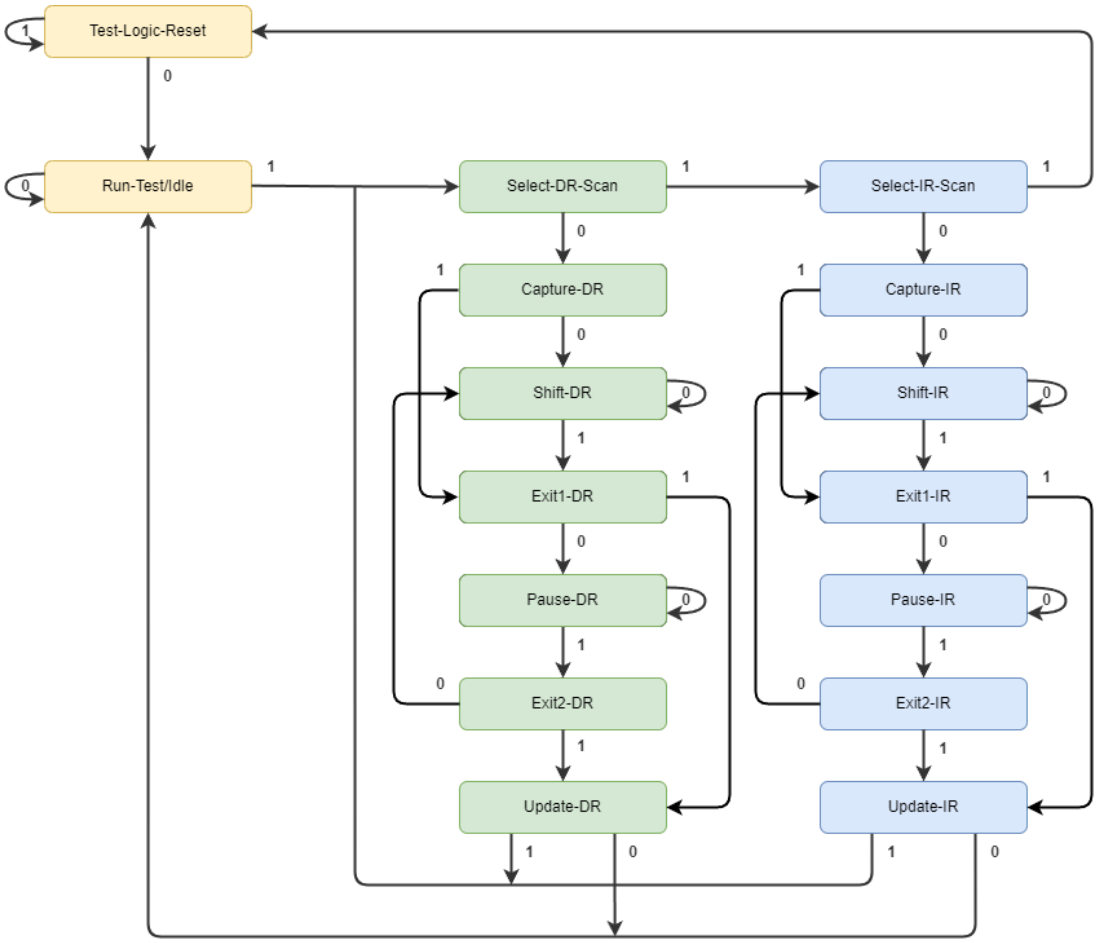


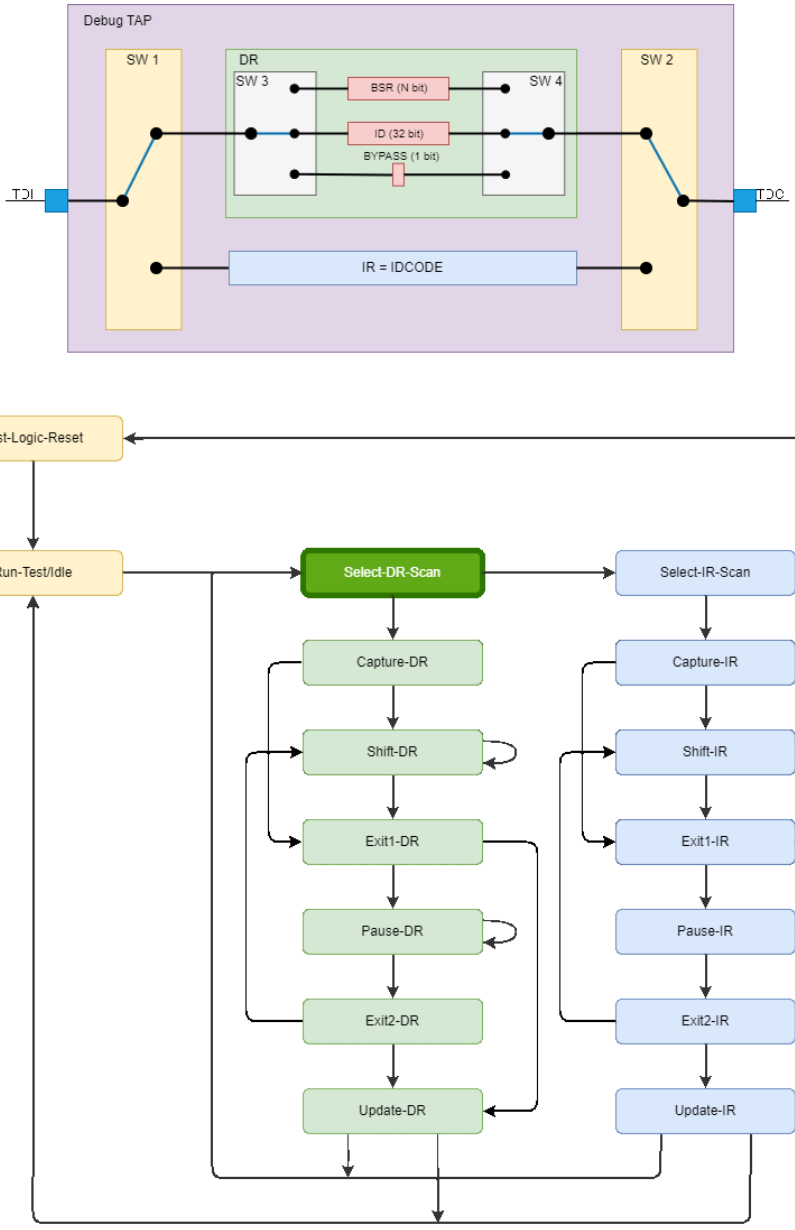
Figure 1.7 - TAP State Diagram

All states have two outputs, and transitions are arranged so that any state can be reached by controlling the dispenser with a single `TMS` signal (synchronized by `TCK`). There are two distinct sequences of states: one for reading or writing to the data register and one for working with the instruction register.

Let's describe the most important states. But since `IR` path and `DR` path have identical states, I will describe these states for both paths at once specifying the differences if necessary.

- **Test-Logic-Reset** — all test logic is disabled, chip behaves normally.
- **Run-Test/Idle** — first state to initialize test logic and default idling state

- **Select-DR/IR-Scan** — this state is necessary to select the current path: data or instruction. I think this can be visualized as the operation of the switches: `SW1`, `SW1`, `SW3`, `SW4`. When the `Select-DR-Scan` the state is hit, the switches `SW1`, `SW2`, `SW3`, `SW4` are switched to the corresponding `DR` register. When the `Select-IR-Scan` state is reached - switches `SW1`, `SW2` are switched to the `IR` register.



[Figure 1.8 - Choose of Data or Instruction Path]

- **Capture-DR/IR** — In this state, there is a parallel loading of the value stored in the selected `DR` register into the shift register if you follow the `Select-DR-Scan` state branch and loading of a special pattern if we follow the `Select-IR-Scan` state path, the value `0x01` is usually selected as the pattern.

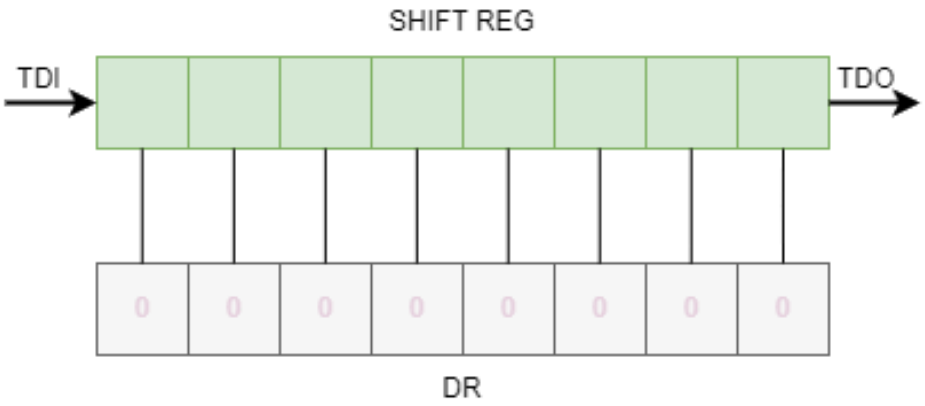


Figure 1.9 - Parallel Data Load into Shift Register

Some clarification about `Capture-DR/IR` state. During the `Capture-DR/IR` stage, the data stored in the register corresponding to this shift register is copied into it. This situation can be considered as follows: for example, you want to read the chip ID. This ID is stored in a special `IDCODE` register somewhere in the chip's internal memory, but this register does not participate in the TAP operation; it merely stores the ID bits. Inside the TAP, there is a shift register whose bit length corresponds to the bit length of the `IDCODE`, and it is this register that is integrated into the JTAG chain between `TDI` and `TDO`; let's call it, for example, `SR_IDCODE`. However, this `SR_IDCODE` does not store the chip ID bits; it serves only for shifting. And during the `Capture-DR` stage, information from the `IDCODE` register is copied into this shift register `SR_IDCODE`, which is then transmitted externally in the `Capture-DR/IR` state. The same thing happens, but in reverse order, for the `Update-DR/IR` state.

- **Shift-DR/IR** — register shifts data from `TDI` one step forward `TDO`. The `Shift-DR` and `Shift-IR` states are the main states for serial-loading data into either data registers or the instruction register.

Figure 1.10 - Shift data in Shift Register

- **Update-DR/IR** — the state in which the data in the shift register is written to the corresponding register in the chip. The `Update-DR` and `Update-IR` states latch the data into the registers, setting the data in the instruction register as the current instruction.

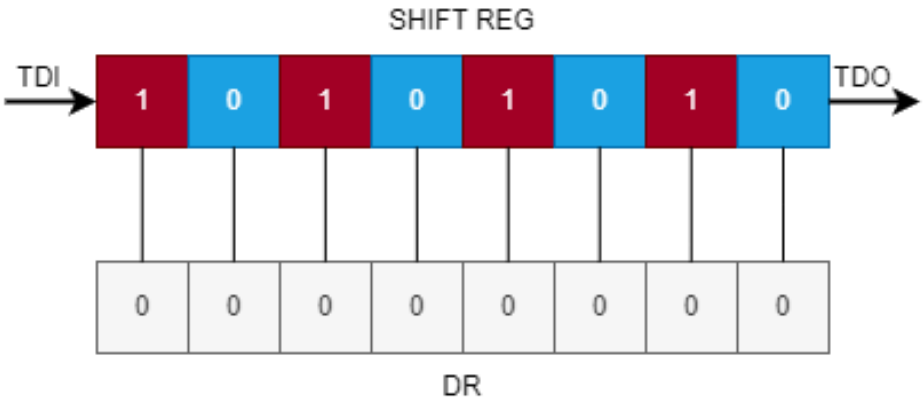


Figure 1.11 - Parallel Data Load from Shift Register

- `Pause-DR/IR` - Temporarlly halt data shifting from `TDI` to `TDO`;

Regardless of the initial state of the TAP controller, the `Test-Logic-Reset` state can always be entered by holding the `TMS` in logic 1 for 5 clock cycles of `TCK`.

---

2. Debugging

As noted in my previous chapter JTAG was initially developed for testing integrated circuits and printed circuit boards. However, over time, its potential for debugging was realized, and now JTAG has become the standard protocol for microcontroller debugging. Many Firmware and Embedded engineers first encountered it in this particular context.

However, there's a catch: while the use of JTAG in testing is fairly standardized, when it comes to debugging, each processor architecture has its unique nuances. With that in mind, this article will focus on debugging using JTAG on the ARM Cortex-M architecture, specifically with the STM32F407VG microcontroller.

2.1 JTAG Access to the STM32F407VG Controller

Before connecting to the STM32F407VG controller via the JTAG protocol, we need to determine the following: the length of the `IR` register and the number of TAP controllers connected to the JTAG chain. The official documentation, specifically the RM0090: Reference manual, can assist us in this.

According to this document, the STM32F407VG has two TAP controllers: the STM32 Boundary Scan and the CoreSight JTAG-DP. The first in the chain is the Boundary Scan TAP with an `IR` register size of 5 bits. Following it is the CoreSight JTAG-DP TAP with an `IR` register size of 4 bits.

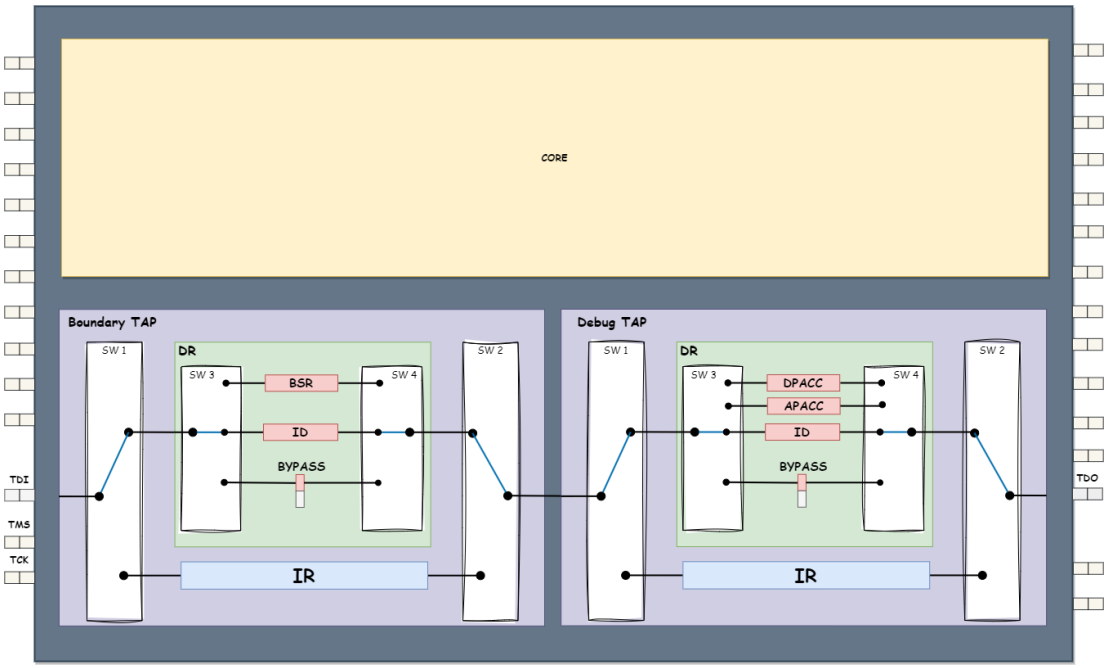


Figure 2.1 - TAPs of STM32F407VG chip

This knowledge is crucial for us because it determines how many bits to send in the JTAG chain and in what sequence to do so when accessing different TAP controllers. According this information, we can connect to the CoreSight JTAG-DP TAP controller and attempt to read its `IDCODE`.

2.1.1 Read of IDCODE

The sequence of bits for reading the `IDCODE` of the CoreSight JTAG-DP TAP controller will look as follows:

```
TMS: 11111011000000000011010000000000000000000000000000000110
TDI: 00000000000111111110000000000000000000000000000000000000
TDO: 00000000010001000000000011101110001000000000010111010010000
```

Figure 2.2 - Bits sequence for IDCODE reading

Let’s analyze this sequence in detail:

1. So the first step is to reset the TAP State Machine to the initial state `Test-Logic-Reset` in order to be sure that we are in a known state. To achieve this, simply hold the `TMS` in logic 1 for 5 clock cycles of `TCK`. The pink bit sequence is responsible for this step:

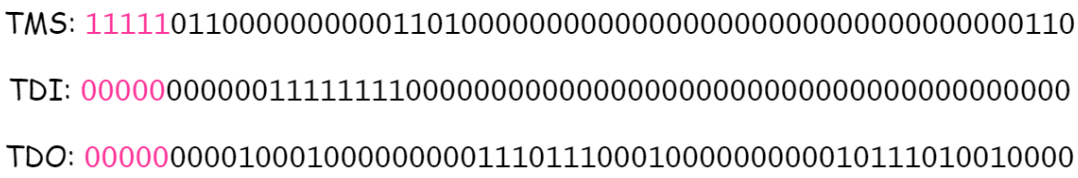


Figure 2.3 - JTAG TAP reset bits sequence

2. Next we need to load the `IDCODE` command code into the `IR` register, and to do this we put the TAP State Machine into the `Shift-IR` state. To do this, feed the `TMS` line with the sequence [0, 1, 1, 0, 0]. A sequence of blue bits is responsible for this step:

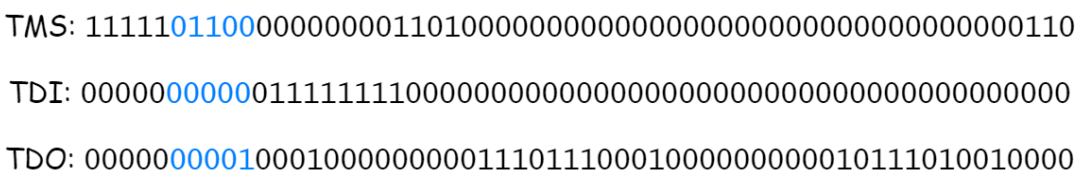


Figure 2.4 - Move to Shift-IR state

As you can see on the `TDO` line in the last measure there is a one. Let's find out where it came from. The thing is that going through the `Capture-IR` state, the TAP controller loads into the shift register a value of `0b0001` for CoreSight JTAG-DP TAP and `0b00001` for Boundary Scan TAP which start to move along the `TDO` line from this measure and LSB bits forward. So, over the next 9 clock cycles (including this one) we will get a `TDO` sequence of `0b100010000`.

3. The next step is to directly load the `IDCODE` command code into the CoreSight JTAG-DP TAP `IR` register and the `BYPASS` command code

into the Boundary Scan TAP `IR` register. We already looked at this kind of situation in the last chapter, but let's look at it again. As you remember there are two TAP controllers in our JTAG chain connected in series and we can not access the CoreSight JTAG-DP TAP registers without having to write something to the Boundary Scan TAP, and the size of the `IR` register for Boundary Scan is 5 bits, and the `DR` register is 32 bits, so to write a new data to the `DR` register of the CoreSight JTAG-DP TAP we need to write a 32 + 32 = 64 bits sequence of bits to the JTAG circuit. This is not very convenient. The `BYPASS` command will help us here. When this command is in the register `IR` the data register of the selected TAP becomes 1 bit long. So now we need to write a sequence of 33 bits Instead of 64. The `IDCODE` code is `0b1110`, and the `BYPASS` command code is `0b11111` total we need to push 9 bits: `0b011111111`. The `TMS` input must be held at logical 0 during the 8 LSB input in order for the machine to remain in `Shift-IR` state. The MSB bit of the instruction is pushed in when there is an exit from this state by setting `TMS` to logical 1. The red bit sequence is responsible for this step:

TMS: 11111011000000000110100000000000000000000000000000000110

TDI: 00000000000111111110000000000000000000000000000000000000

TDO: 00000000010001000000000011101110001000000000010111010010000

Figure 2.5 - Load Instructions

The `TDO` line continues to output the sequence `0b100010000` which was loaded in the `Capture-IR` state.

After this step we are in the `Exit1-IR` state.

4. Great, the necessary commands are set in the `IR` registers and now we need to return to the `Run Test/Idle` state for further work. At this point, thanks to the last 1 on the `TMS` line we are in the `Exit-1-IR` state and hence according to the state diagram we need to set

the following bit sequence on the TMS line to Run Test/Idle: [1, 0]. The orange bit sequence is responsible for this step:

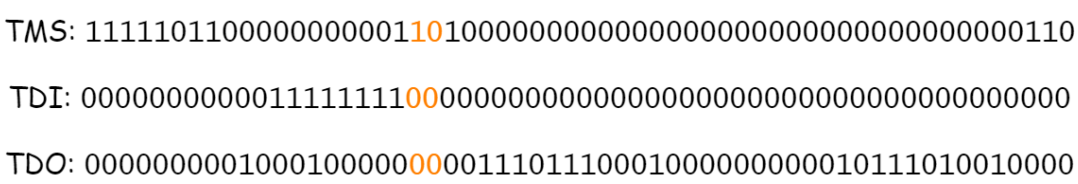


Figure 2.6 - Return to the RunTestIdle state

5. After setting the IDCODE command in the IR register we need to read IDCODE from the DR register. To do this the first step is to go to the Shift-DR state. To do this we set the following bit sequence [1, 0, 0] on the TMS pin considering that we are in the Run Test/Idle state at the moment. The yellow bit sequence is responsible for this step:

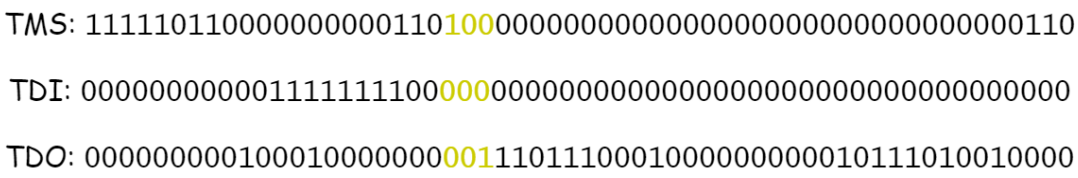


Figure 2.7 - Move to the Shift-DR state

On the TDO line we have the sequence: [0, 0, 1]. The last one is already the LSB of our chip ID. The fact is that passing through the Capture-DR state our chip ID is loaded into the shift register and begins to move outward in the next clock cycles.

6. In the Shift-DR state the remaining bits of the chip ID are unloaded. To remain in the Shift-DR state the TMS input must be held at loic 0 during the input of all bits except the MSB bit. The MSB data is pulled out when this state is exited by setting the TMS to a logical 1. The green bit sequence is responsible for this step:

TMS: 11111011000000000011010000000000000000000000000000000110

TDI: 00000000000111111110000000000000000000000000000000000000

TDO: 00000000010001000000000111011100010000000001011101001000 0

Figure 2.8 - Shifting the ID code

So we got a chip ID of the following kind:

```
11101110001000000000010111010010
```

Let's check this code according to the documentation:

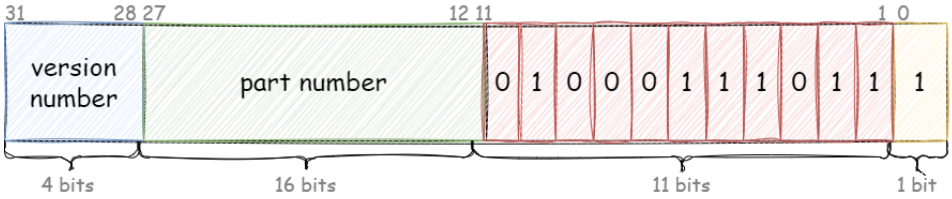


Figure 2.9 - Format of ID code for STM32F407VG microcontroller

Something is wrong here. What is wrong is that the LSB is in the usual position in the figure, but we took it in the inverted order. Let's correct this situation:

```
01001011101000000000010001110111
```

7. And the final step is to go back to the `Run Test/Idle` state. For this step is responsible for a sequence of bits of unknown color:

TMS: 11111011000000000011010000000000000000000000000000000110

TDI: 00000000000111111110000000000000000000000000000000000000

TDO: 00000000010001000000000111011100010000000001011101001000 0

Figure 2.10 - Return to the Run TestIdle state

2.2 Interaction with memory

Great, the JTAG connection works and we've read the `IDCODE`. Let's now proceed and try interacting with the microcontroller's memory by writing a value to it and reading it back. After all, reading/writing memory is one of the primary operations when debugging a microcontroller.

To interact with the microcontroller's memory, we need to understand the mechanism behind it. The JTAG protocol itself doesn't provide any specifics about this. Access to the internal resources of ARM architecture controllers (such as memory) is facilitated by a special module called DAP (Debug Access Port). As I said I've detailed this module extensively in the Appendix A. Unfortunately, without this knowledge, advancing further might be challenging. Therefore, I highly recommend reading that Appendix A or the official documentation IHI0031 - ARM Debug Interface

The first step for any access operation to the internal resources of an ARM architecture microcontroller with a DAP module is to request clocking and power for the debug module. This is done by setting the `CSYSPWRUPREQ` and `CDBGPWRUPREQ` bits in the DP `CTRL/STAT` register.

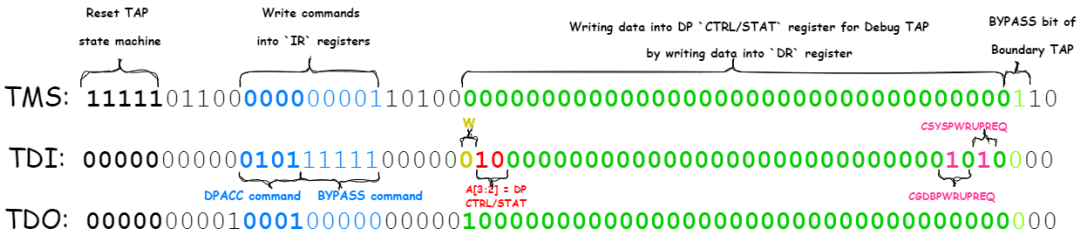


Figure 2.11 - Request clocking and power for the debug module

After this initial setup, we can begin our memory operations.

In the following two sections, I will be heavily referencing the Practical Part from the Appendix A. This section describes the algorithm for reading/writing memory through interaction with the DAP module. Additionally, I will supplement this algorithm with a sequence of bits, illustrating how this will appear from the perspective of the JTAG protocol.

2.2.1 Writing a variable to memory

Alright, let's write the value `0xAA55AA55` to the address `0x20000000`:

1. Write `0b1010` (the `DPACC` register code) into the `IR` register.
2. Some settings in the `AP` register `CSW` need to be configured. To access this register, we need to select the corresponding `AP` and register bank in the `DP SELECT` register. Write to the `DR` register:
    - DATA[31:0] = `0x00`
        - APSEL[31:24] = `0x00`
        - APBANKSEL[7:4] = `0x00`
    - A[3:2] = `0b10` (address of `SELECT` register)
    - RnW = `0b0`

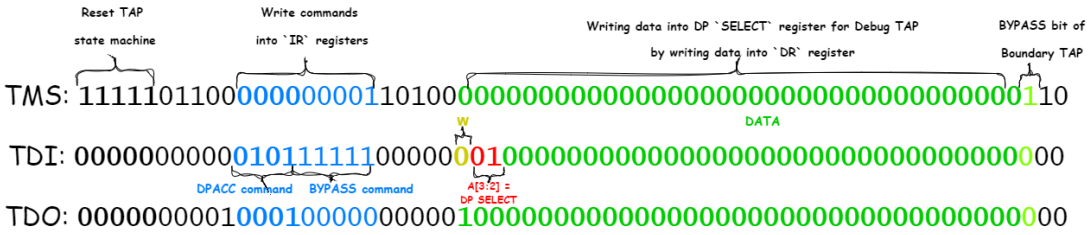


Figure 2.12 - Select the corresponding AP and register bank

3. Next, we need to write the `CSW` register. Since this is an `AP` register, we need to use the `APACC` register to access it:
    - Write `0b1011` (the `APACC` register code) into the `IR` register.
4. Form the data for the configuration: set the data size for writing to 32-bits (`Size[0:2] = 0b010`), disable the auto-increment function of the address (`AddrInc[5:4] = 0b00`). Write to `DR`:
    - DATA[31:0] = `0x00000002`
    - A[3:2] = `0b00` (address of `CSW` register)
    - RnW = `0b0`

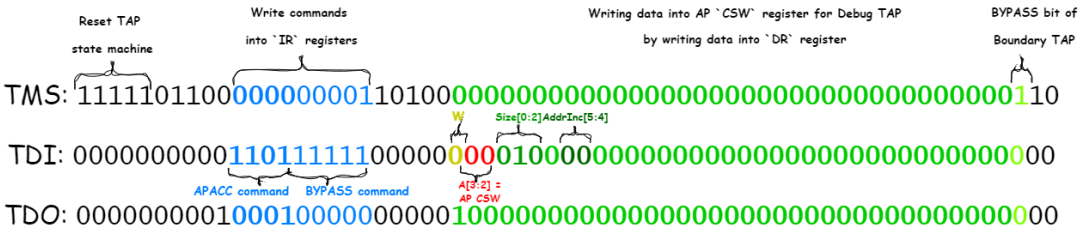


Figure 2.13 - Set the data size and disable the auto-increment function

5. Then we need to set the address of the memory cell to which we want to write the data. This is done through the `AP` register `TAR`. And since this register belongs to the same `AP` as `CSW` and is in the same bank, we can omit the reference to the `DP SELECT` register and immediately write the address value. Write to `DR`:
   - DATA[31:0] = `0x20000000`
   - A[3:2] = `0b01` (address of `TAR` register)
   - RnW = `0b0`

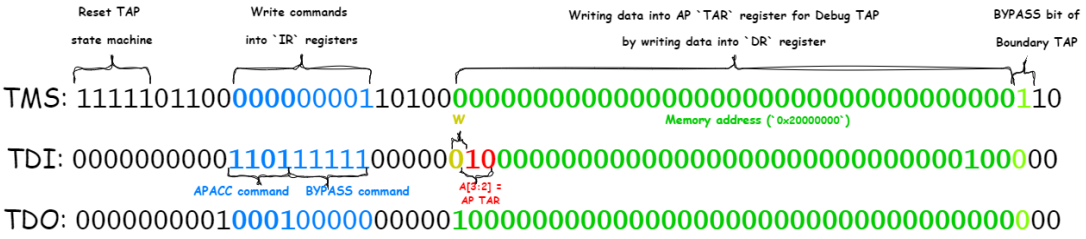


Figure 2.14 - Set the address of the memory cell

6. The last step is to actually write the data. To do this, we need to write them to the `DRW` register. Write to `DR`:
   - DATA[31:0] = `0xAA55AA55`
   - A[3:2] = `0b11` (address of `DRW` register)
   - RnW = `0b0`

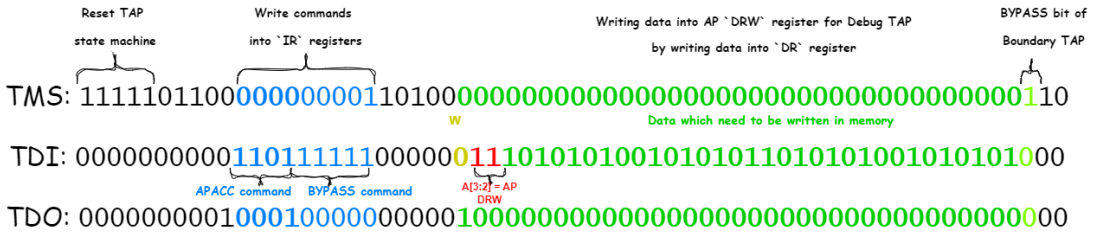


Figure 2.15 - Write the data

---

3. Boundary Scan

Boundary Scan is a technology for testing digital integrated circuits and printed circuit boards based on the IEEE 1149.1 standard. It enables diagnostics, testing, and debugging without requiring physical access to the chip's pins or the board.

Boundary Scan provides developers with the ability to control the state of the chip's pins and read their values through a special chain of flip-flops known as the **BSR (Boundary Scan Register)**.

3.1 The Principle of Boundary-Scan

The principle of operation is that special cells — scan cells — are inserted between the physical pins of the chip and its internal logic.

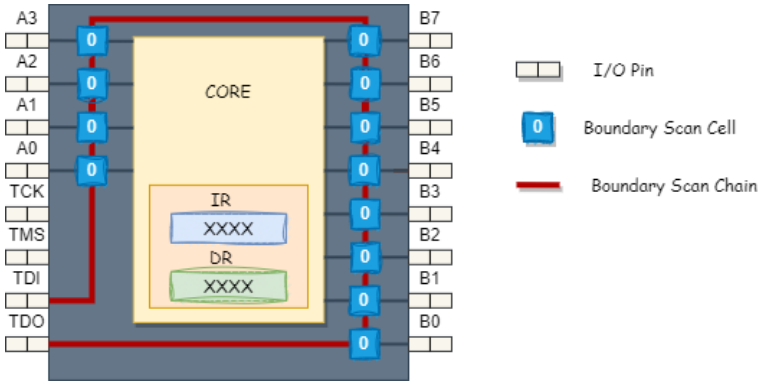


Figure 3.1 - Boundary Scan Architecture Overview

In normal mode, these cells are transparent and the core is connected to I/O ports. In boundary scan mode, the core is isolated from the ports, and the port signals are controlled by the JTAG interface.

The boundary scan cells are connected to a serial shift register, which is referred to as the boundary scan register (`BSR`). This register can be used to read and write port states.

In general, the principle of Boundary-Scan operation is that with the help of special JTAG commands, it is possible to set values in the scan cells and, through them, influence the chip pins.

Following boundary scan instructions are defined in the IEEE standard:

- `BYPASS` (mandatory): `TDI` is connected to `TDO` via a single shift register.
- `SAMPLE` (mandatory): Takes a snapshot of the pins of the IC.
- `PRELOAD` (mandatory): Loads data to the boundary scan register.
- `EXTEST` (mandatory): Apply preloaded data of the boundary scan register to the ports.
- `INTEST` (optional): Apply preloaded data of the boundary scan register to the core logic.
- `CLAMP` (optional): Apply preloaded data of the boundary scan register to the ports and selects the `BYPASS` register as the serial path between `TDI` and `TDO`.
- `HIGHZ` (optional): Places the IC in an inactive drive state (e.g. all ports are set to high impedance state) and leaves `BYPASS` register as the selected register.

The structure of the boundary scan chain and the instruction set are described with the Boundary Scan Description Language (BSDL). BSDL is a subset of the Very High-level Design Language (VHDL). The BSDL files are provided by the IC manufacturer. The BSDL language will be covered in the next article, so we will not describe it in detail here.

3.2 The Boundary Scan Cells

As mentioned above, scan cells are the core element that makes Boundary Scan possible. These cells can be programmatically configured to perform various functions, such as transmitting or receiving data, allowing you to test connections between chips without having to physically access the pins.

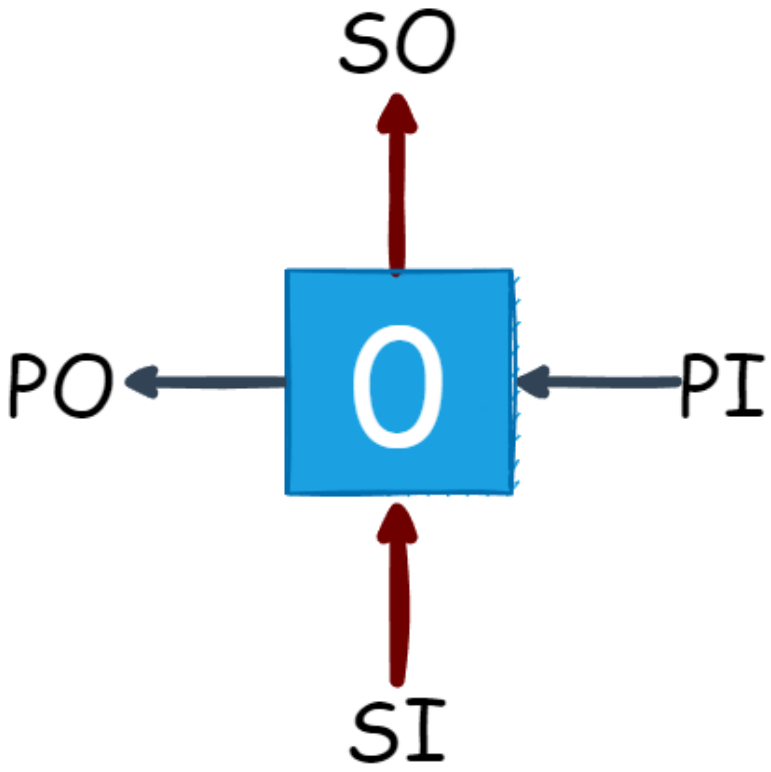


Figure 3.2 - Boundary scan cell inputs/outputs

Each boundary-scan cell can: - Capture data on its parallel input `PI` - Update data onto its parallel output `PO` - Serially scan data from `SO` to its neighbour's `SI` - Behave transparently: `PI` passes to `PO`

At the device level, the boundary-scan elements contribute nothing to the functionality of the internal logic. The boundary-scan path is independent of the function of the device.

The boundary-scan cells must be provided on all device digital input and digital output signal pins, except Power and Ground.

Scan cells can be categorized by functionality into the following types:

- **Input Cells**: used to monitor input signals.
- **Output Cells**: used to control the output signals.
  - `Output2`: This cell type does not support three-state logic. It can set the pin to state "`0`" or "`1`".

    - `Output3`: This cell type supports three-state logic, allowing a pin to be in the "`0`", "`1`" or "`Hi-Z`" (high impedance) state.
- **Bidirectional Cells**: can be used for both input and output. Cells of this type typically support three-state logic.
- **Control Cells**: used to control other cell types.
    - `Control`: This cell type can control the functionality of one or more other cells, for example, by switching them between input and output modes. This cell is not connected to the chip pins
    - `ControlR`: This is similar to the `Control` type, but with the additional ability to read the state of the controlled cells. This cell is not connected to the chip pins.
- **Cell on clock input**. A function that indicates that this cell is connected to the system clock frequency and this allows `INTEST` mode operation

Each scan cell typically consists of a small set of flip-flops and logic elements that allow it to perform various functions such as storing data, transferring data to other cells, etc.

In general, a scan cell can be represented by the following scheme:

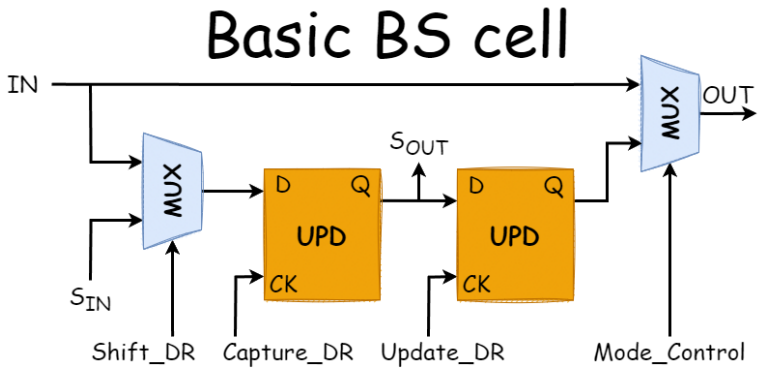


Figure 3.3 - Boundary scan cell scheme

A Boundary Scan cell's internal architecture can be highly different. In its version from 2001, the IEEE Std. 1149.1 describes ten different cell types (`BC_1` to `BC_10`):

- BC_1 – The basic Boundary Scan cell that can be used as an input cell, output cell, control cell, and internal cell. Supports all instructions.
- BC_2 – Boundary Scan cell that can be used as an input cell. Cell architecture is like BC_1 with the exception of a multiplexer in the signal patch at the entrance to the cell from the Parallel input.
- BC_3 – Cell only used for inputs or internal cells as it does not possess an Update latch, but it does support the INTEST instruction.
- BC_4 – Like the BC_3, this cell does not possess an Update latch. Also, a multiplexer has been removed from the system signal path. Removing the multiplexer removes some potential signal delay through the cell. This cell cannot be used on any input pin except a system clock.
- BC_5 – Cell can be used as a merged cell application. Merged cells act as an input cell, thus satisfying the requirement that an input pin have a cell and it can also serve to drive the enable of an output driver.
- BC_7 – Cell can provide data to the output driver and also monitor pin activity even when the output drive is driving the pin.
- BC_8 – Cell monitors only the pin driver output and therefore does not support the INTEST instruction.
- BC_9 – A self-monitoring cell for outputs that does support the INTEST instruction.
- BC_10 – A self-monitoring cell that does not support the INTEST instruction.

The number of cells need not necessarily match the number of chip pins. For example, if the chip pin is bidirectional (pin B2 in Figure 4), conceptually at least, three boundary-scan cells are required: one on the input side, one on the output side, and one to allow control of the IO status. In practice, the two IO scan cells are usually combined into a single multi-function cell called a BC_7.

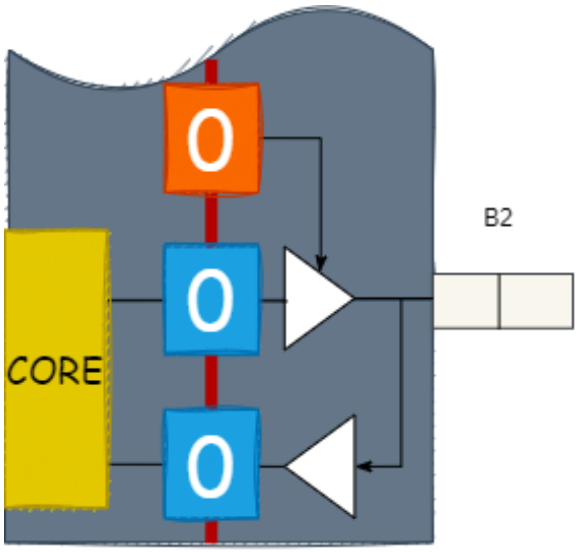


Figure 3.4 - Bidirectial Boundary scan cell example

3.3 The Boundary Scan Register

Boundary Scan Register (`BSR`) is one of the data registers (`DR`) consisting of a sequence of scan cells (Boundary-Scan Cells). Figure 5 shows how the scan cells are linked together to form the boundary-scan register. The order of linking within the device is determined by the physical adjacency of the pins and/or by other layout constraints. The boundary-scan register is selected by the `EXTEST`, `SAMPLE`, `PRELOAD`, and `INTEST` instructions.

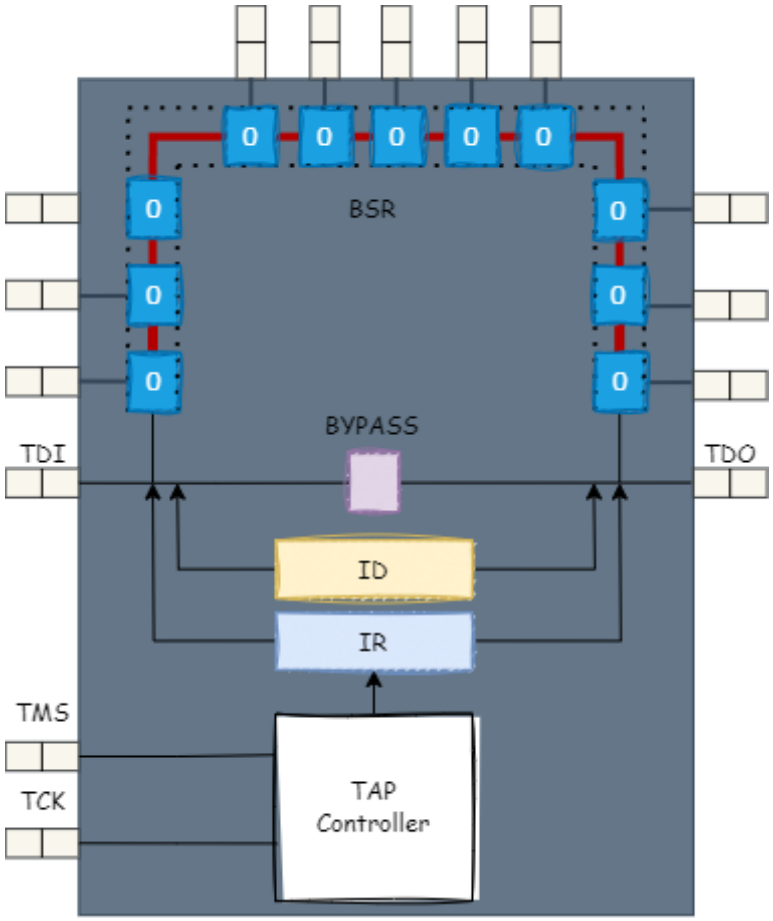


Figure 3.5 - JTAG Boundary Scan Register

During scan operations, data is shifted BSR in the direction from `TDI` to `TDO`. The values in the scan cells can be changed or read, allowing testing to be performed.

The size and format of this register determine the number and sequence of scan cells in a particular chip - i.e., before testing, it is necessary to find out the length and format of the `BSR` register. This information is supplied by the manufacturer in the BSDL file format for a particular chip. Here is an example of a truncated BSDL file for the STM32F407 microcontroller:

```
num         cell          port          function
405         BC_1,         *,            CONTROL,
404         BC_1,         PE2,          OUTPUT3,
403         BC_4,         PE2,          INPUT,
```

where, - `num`: Is the cell number. - `cell`: Is the cell type as defined by the standard. - `port`: Is the design port name. Control cells do not have a port name. - `function`: Is the function of the cell as defined by the standard. Is one of input, output2,output3, bidir, control or controlr.

We can see from this file that:

- The number of scan cells, and consequently the length of the `BSR` register is 405 pieces/bit.
- One I/O pin can have 3 scan cells which are divided according to the functionality to be performed: `CONTROL`, `OUTPUT3`, `INPUT`.

This information must be considered when forming the contents of the register `BSR`.

3.4 The Boundary Scan Instructions

3.4.1 SAMPLE Instructions

This command reads the current values from the scan cells and passes them to the `TDO` output. This is useful for reading the current state of the chip pins. This command causes `TDI` and `TDO` to be shorted to the `BSR` register. However, the chip remains in the normal operating state. During the execution of this command, the `BSR` register can be used to capture the data exchanged by the chip during normal operation. In other words, with this instruction, we can read signals from the microcontroller output without disturbing its operation.

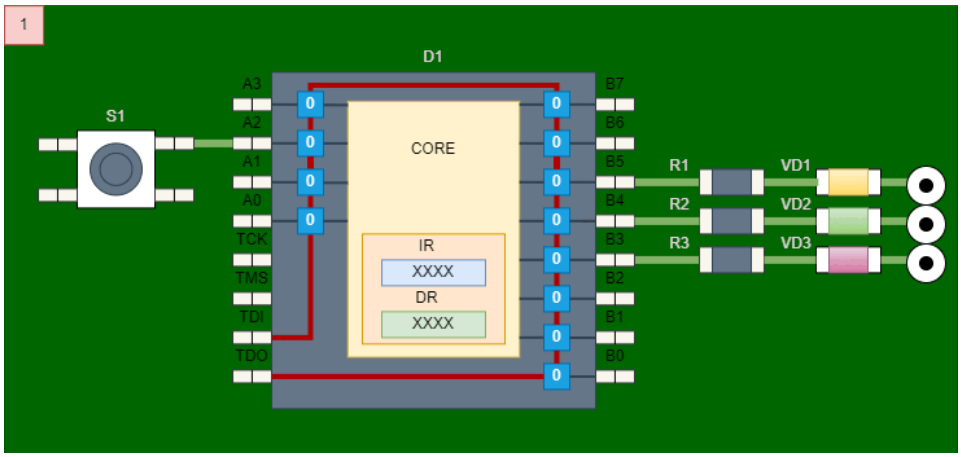


Figure 3.6 - SAMPLE instruction example

In step 3, when the `SAMPLE` instruction is loaded into the `IR` register, the signal is read from the pins to the scan cell. In the next steps, we move to the `Shift-DR` state and read the `BSR` register along with the value of the pins received in the previous step. Note that when the read unit passes through the scan cells bound to pins `B3`, `B4`, `B5` the LEDs connected to these pins do not light up, because at the moment the scan cells are not connected to the chip pins.

3.4.2 PRELOAD Instructions

This command allows you to preload certain values into Boundary-Scan Cells for later testing or other operations.

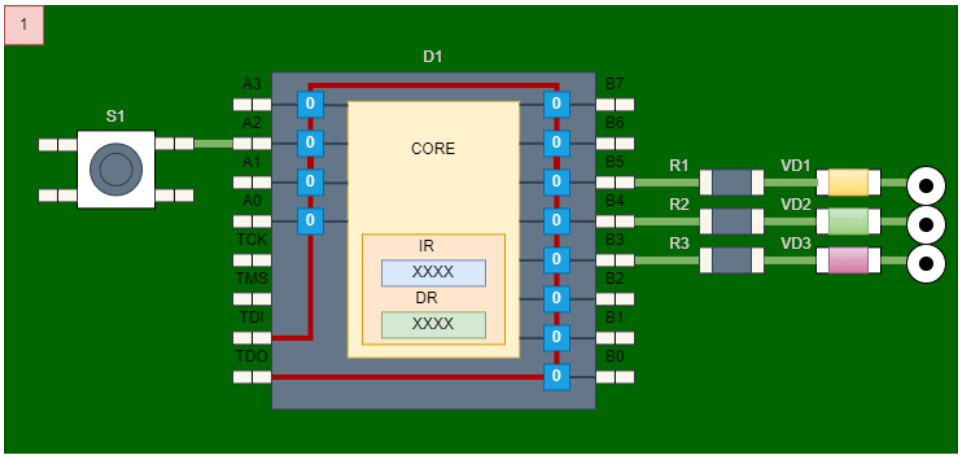


Figure 3.7 - PRELOAD instruction example

Here, everything is simple. After writing the command `PRELOAD`, we just need to write the values in the scan cells to the register `BSR` for the corresponding pins in the state `Shift-DR`. And again, note that after writing all the values to the scan cells, the LEDs connected to pins `B3`, `B4`, `B5` do not light up either.

3.4.3 SAMPLE/PRELOAD Instructions

Sometimes two commands: `SAMPLE` and `PRELOAD` are combined into one. When writing this command to the `IR` register, the values of the pins are read into the scan cells (`BSR` register) and then in the `Shift-DR` state we read these values and write new values for the pins into the scan cells (`BSR` register). Again, after the `SAMPLE/PRELOAD` instruction is completed, has no effect on the pins themselves.

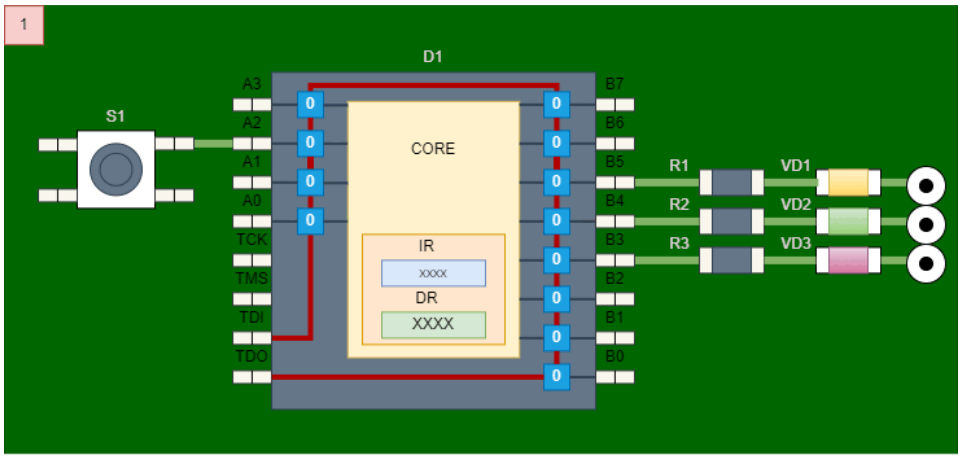


Figure 3.8 - SAMPLE/PRELOAD instruction example

Typically, the `SAMPLE/PRELOAD` command is the first command to be executed during Boundary Scan testing, and it serves as the basis for many other operations.

3.4.4 EXTEST Instructions

The `EXTEST` command in JTAG is used to test external circuits connected to the microcontroller pins. When the microcontroller is in `EXTEST` mode, all its function blocks are disabled and the microcontroller pins can be used to test the connected external circuits for short circuits, open circuits, etc. This command can be used to verify the pins and circuits of a microcontroller during the manufacturing process and to verify that external circuits are properly wired during development.

This command is the one for which we wrote values to the scan cells in the `PRELOAD` command. Because exactly `EXTEST` makes the scan cells transfer the values of the signals stored in them to the output.

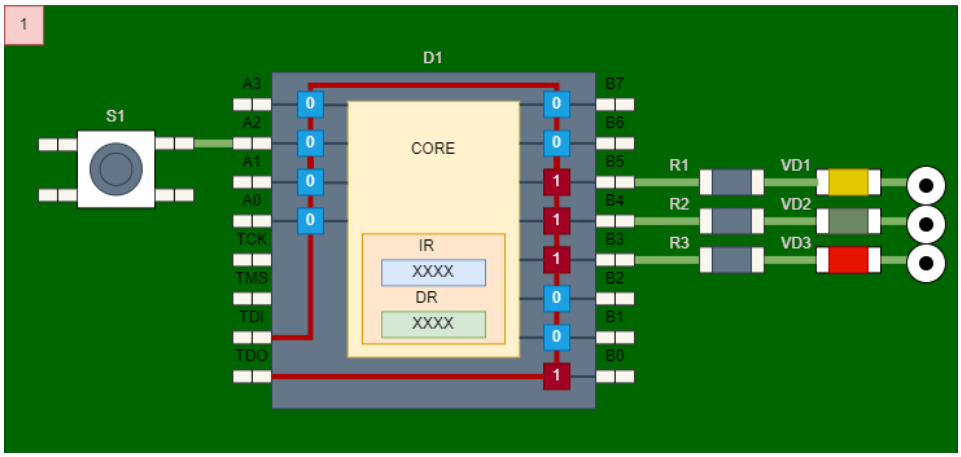


Figure 3.9 - EXTEST instruction example

After using this command, the I/O pins are disconnected from the internal logic of the microcontroller and you can no longer control them from the program.

3.4.5 INTEST Instructions

It is also possible to use boundary-scan cells to test the internal functionality of a device. This use of the boundary-scan register is called Internal Test, shortened to Intest. Intest is only really used for very limited testing of the internal functionality to identify defects such as the wrong variant of a device or to detect some gross internal defect.

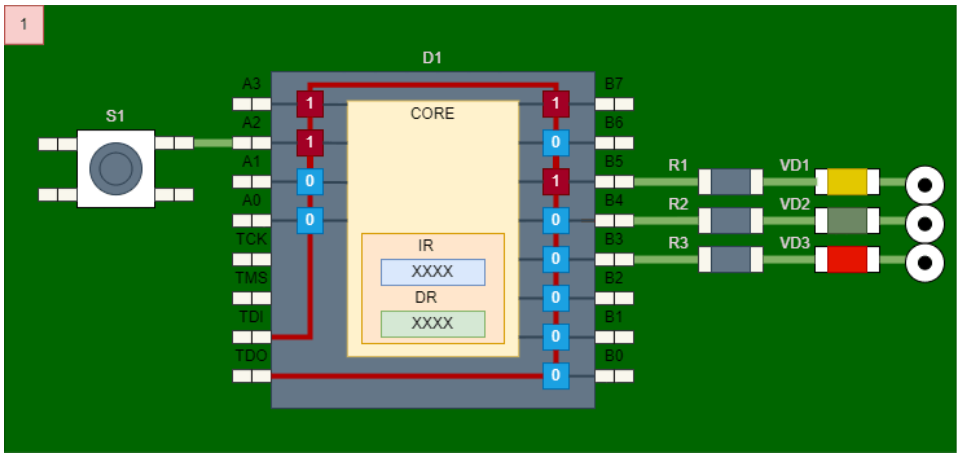


Figure 3.10 - JTAG INTEST instruction example

---

4. BSDL

In the previous chapter I briefly touched on how `.bsd` files written in Boundary Scan Description Language (BSDL) describe the structure of the boundary scan chain and the instruction set. In this article, we will examine this language's syntax more closely before seeing how `.bsd` files are leveraged in JTAG testing in the next article.

Files with the `.bsd` extension describe how to interface with a particular chip using the JTAG interface in the context of PCB and chip boundary-scan testing. The main functions of these files are:

1. **Chip Configuration Description:** file provides a detailed description of the chip pin configuration, including which pins support JTAG.
2. **Testing and Diagnostics Support:** These are used to support testing the PCB for shorts, open circuits, and other common problems. This is especially useful for verifying that components are properly mounted on the board.
3. **Test Automation:** allow you to automate the testing process by providing the necessary information to the software that controls the test equipment.
4. **Portability of tests:** Since BSDL is based on a standard, the files can be used with a variety of hardware and software that supports the standard, providing a high degree of test portability.

These files are a key component in the testing process of electronic devices, especially in complex systems where accurate and efficient component testing is important.

BSDL is a subset of VHDL, a hardware description language that describes how a particular JTAG device must be implemented for boundary scan. You'll find `.bsd` files available from most vendors.

One of the major uses of BSDL is as an enabler for the development of tools to automate the testing process based on IEEE 1149.1. Tools developed to support the standard can control the TAP if they know how the boundary-scan architecture was implemented in the device. Tools can also control the I/O pins of the device. BSDL provides a standard machine and human-readable data format for describing how IEEE 1149.1 is implemented in a device.

Let's take a look at the format and syntax of BSDL files.

4.1 Syntax

A BSDL description for a device consists of the following elements:

- Entity Descriptions
- Generic Parameter
- Logical Port Description
- Use Statements
- Pin Mapping(s)
- Scan Port Identification
- IDCODE Register Description
- Instruction Register Description
- Register Access Description
- Boundary Register Description

4.1.1 Entity Descriptions

The entity statement names the entity, such as the device name (e.g., `STM32F405_415_407_417_LQFP100`). An entity description begins with an `entity` statement and terminates with an `end` statement:

```
entity XYZ is
    {statements to describe the entity go here}
end XYZ
```

All of the rest of the attributes and descriptions that we will look at we can be located within this block. This structure allows the device

to be described for testing, debugging, and diagnostic purposes, providing the necessary information for tools and equipment utilizing boundary scan technology.

4.1.2 Generic Parameter

This keyword is used to define a parameter that can be changed when implementing the description in a particular device.

```
generic (PHYSICAL_PIN_MAP : string := "DW");
```

In this example, `PHYSICAL_PIN_MAP` is a generic parameter that defines the chip body type, which will default to the value “DW”.

4.1.3 Logical Port Description

The port description gives **logical** names to the I/O pins, and define their function as input, output, bidirectional, and so on.

```
port (
    OE    : in      bit;
    Y     : out     bit_vector(1 to 3);
    A     : in      bit_vector(1 to 3);
    GND   : linkage bit;
    VCC   : linkage bit;
    NC    : linkage bit;
    TDO   : out     bit;
    TMS   : in      bit
    TDI   : in      bit
    TCK   : in      bit
);
```

The `port` block contains both “useful” pins of the chip and the JTAG pins, as well as power, ground, and missing pins - pins that are not supposed to be connected to the chip.

If the port is a single port, it is marked as `bit`. If it is a bus, it is labeled as `bit_vector(X to Y)`. Inputs are `in`, outputs are `out`, and

bidirectional lines are `inout`. Power/ground lines, analog I/O, and pins not connected to the crystal are marked as `linkage`.

4.1.4 Use Statements

The use statement is used to indicate that a particular package or library will be used in the current context.

For example:

```
use STD_1149_1_1994.all;
```

Indicates that this file uses the IEEE 1149.1 1994 version of the IEEE 1149.1 standard library. In the BSDL context, this expression indicates that the definitions and constructs provided in the 1994 IEEE 1149.1 standard are used to describe the boundary scan behavior. This is to ensure that the description is compatible with tools supporting that standard.

4.1.5 Pin Mapping(s)

This element provides a mapping of logical signals from Logical Port Description section onto the physical pins of a particular device package. This is necessary because the logical signals of a chip may be the same, but the specific pins are different for different packages.

```
attribute PIN_MAP of XYZ : entity is PHYSICAL_PIN_MAP;

constant DW:PIN_MAP_STRING:=
    "OE  :  1," &
    "Y   :  (2,3,4)," &
    "A   :  (5,6,7)," &
    "GND :  8," &
    "VCC :  9," &
    "TDO :  10," &
    "TDI :  11," &
    "TMS :  12," &
```

```
    "TCK :  13," &
    "NC  :  14";
```

The distribution of single pins is quite transparent: one logic name - one pin:

```
"OE  :  1," &
```

The buses are described next way:

```
"Y   :  (2,3,4)," &
```

4.1.6 Scan Port Identification

The scan port identification statements define the device's TAP.

```
attribute TAP_SCAN_CLOCK of TCK : signal is (10.0e6, BOTH);
attribute TAP_SCAN_IN of TDI : signal is TRUE;
attribute TAP_SCAN_OUT of TDO : signal is TRUE;
attribute TAP_SCAN_MODE of TMS : signal is TRUE;
```

Values: `TAP_SCAN_CLOCK`, `TAP_SCAN_IN`, `TAP_SCAN_OUT`, `TAP_SCAN_MODE` - constants defining standard JTAG signals. Values: `TCK`, `TDI`, `TDO`, `TMS` are the port names defined in the **Pin Mapping(s)** section for a particular chip. The value `TRUE` indicates active use of this port.

Additionally, the `TAP_SCAN_CLOCK` attribute specifies the maximum frequency of the clock signal, in this example it is 10 MHz and that the signal is active on both clock edges (rising and falling). Only the `BOTH` parameter is used for the clock signal, but in general the following parameters can also be used for signals: `RISING` or `FALLING`.

4.1.7 IDCODE Register Description

The `IDCODE_REGISTER` attribute is used to define the `IDCODE` register of the chip. This attribute is part of the IEEE 1149.1 (JTAG) specification.

```
attribute IDCODE_REGISTER of <entity> : entity is
    "{specific attribute value}";
```

Example:

```
attribute IDCODE_REGISTER of STM32F405_415_407_417_LQFP100:
entity is
     "XXXX" &                   -- 4-bit version number
     "0110010000010011" &       -- 16-bit part number
     "00000100000" &            -- 11-bit identity of the
manufacturer
     "1";                       -- Required by IEEE Std 1149.1
```

4.1.8 Instruction Register Description

The Instruction Register description identifies the device-dependent characteristics of the Instruction Register.

```
attribute INSTRUCTION_LENGTH of XYZ : entity is 2;

attribute INSTRUCTION_OPCODE of XYZ : entity is
    "BYPASS (11), "&
    "EXTEST (00), "&
    "SAMPLE (10) ";

attribute INSTRUCTION_CAPTURE of XYZ : entity is "01";
```

The `INSTRUCTION_LENGTH` attribute defines the length of the `IR` register in bits.

The `INSTRUCTION_OPCODE` attribute defines the command codes.

The `INSTRUCTION_CAPTURE` attribute defines the value that will be loaded into the shift register when TAP enters the `Capture-IR` state, in other words, this attribute specifies what the instruction register is initialized with.

---

5. Usage Scenarios

The main examples of JTAG usage, such as debugging and testing of boards in production, we have considered in the previous parts. And for firmware/embedded developers only the first example - debugging - is the most useful. As far as I know, we almost never encounter the second option, although it can be very useful too. And in this article I want to look at two examples of JTAG Boundary Scan which I think can be very useful in everyday work of a firmware/embedded developer: Bring-up and Revers Enginnering.

5.1 Getting Started

To make working on this use case more convenient you need a specialized program. In this article, we will look at working with the program TopJTAG Probe and debugger SEGGER J-Link. These aren't the only tools suitable for this task, but I am a long-time fan of J-Link debuggers, and the TopJTAG Probe program turned out to be the simplest and the most accessible from a quick Google search.

While many people are familiar with SEGGER products, TopJTAG Probe requires an introduction.

The TopJTAG Probe program is software that works with the JTAG interface and is used for testing printed circuit boards (PCB). This software is commonly used to diagnose and repair PCBs and work with microcontrollers and other devices that support JTAG.

A boundary-scan (JTAG) based simple logic analyzer and circuit debugging software. It provides the ability to monitor pin values in real-time without interference with the normal operation of a working device and to interactively set up pin values to test board-level interconnects or on-chip internal logic.

To start working with TopJTAG Probe, we need to create a new project. To do this, go to the *File->New Project Wizard* menu.

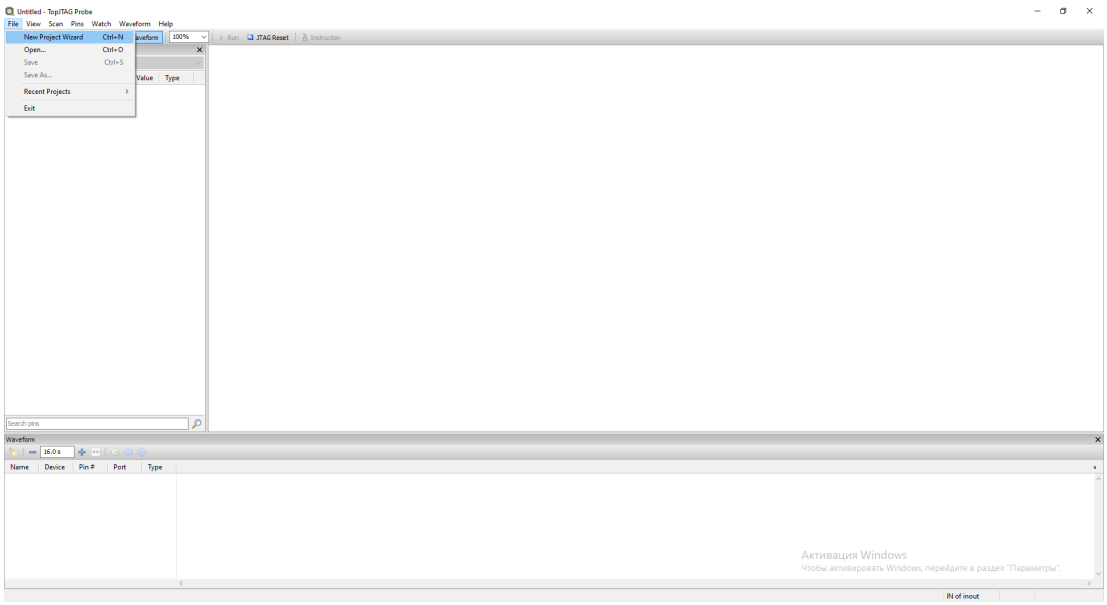


Figure 5.1 - The TopJTAG main screen

In the next window, select the required JTAG Probe and frequency. In my case, it is SEGGER J-Link and 12 MHz.

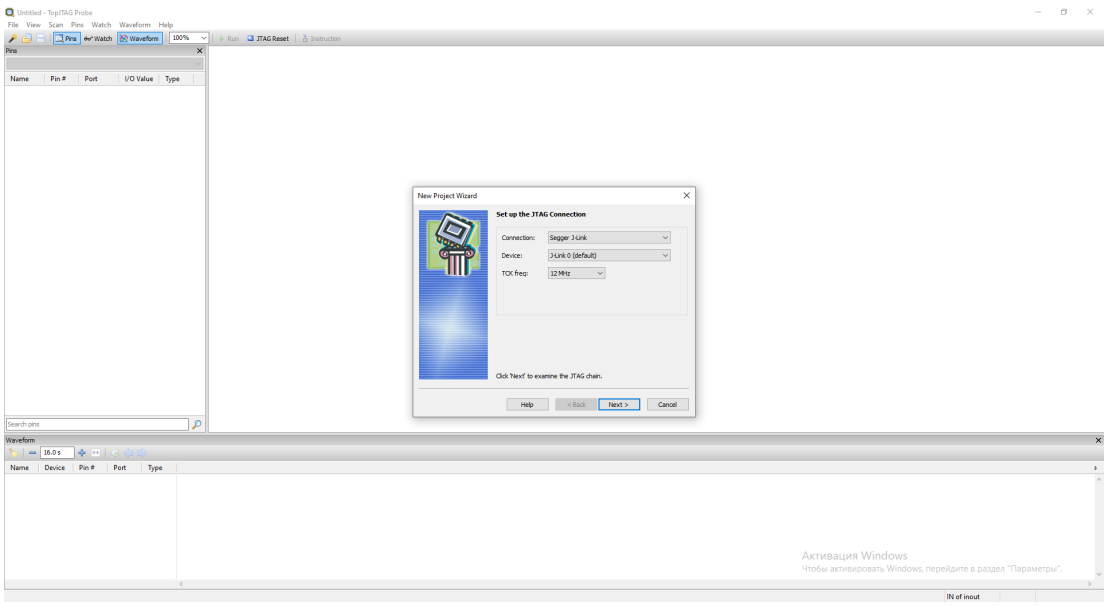


Figure 5.2 - The JTAG Probe and frequency screen

Click *Next*.

The program will scan our JTAG circuit and display a list of available TAP IDs.

You should already have a debugger connected to the board and the computer for this step to go well.

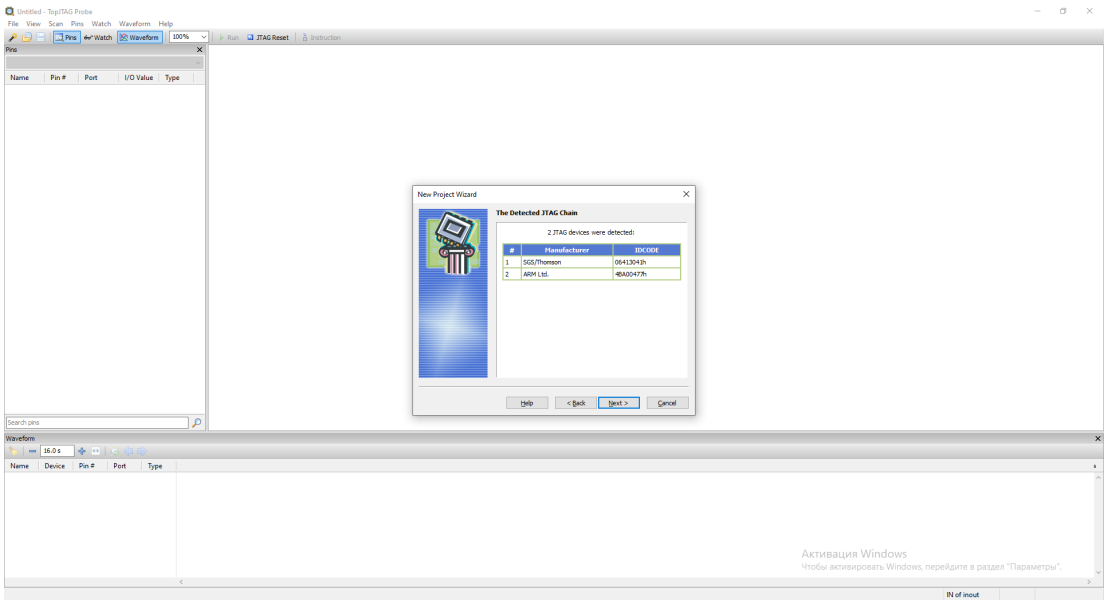


Figure 5.3 - List of available TAPs

Click *Next*.

At this point, you must specify .BSD files for each detected TAP. If you do not specify a .BSD file for a TAP, it will be put into the `BYPASS` state.

We specify STM32F405_415_407_417_LQFP100.bsd file for the first TAP - SGS/Thomson(`06413041h`) as it is responsible for Boundary Scan. We leave the second TAP in `BYPASS`. To select a file, click on the *CLICK HERE TO SET* link and select the desired file.

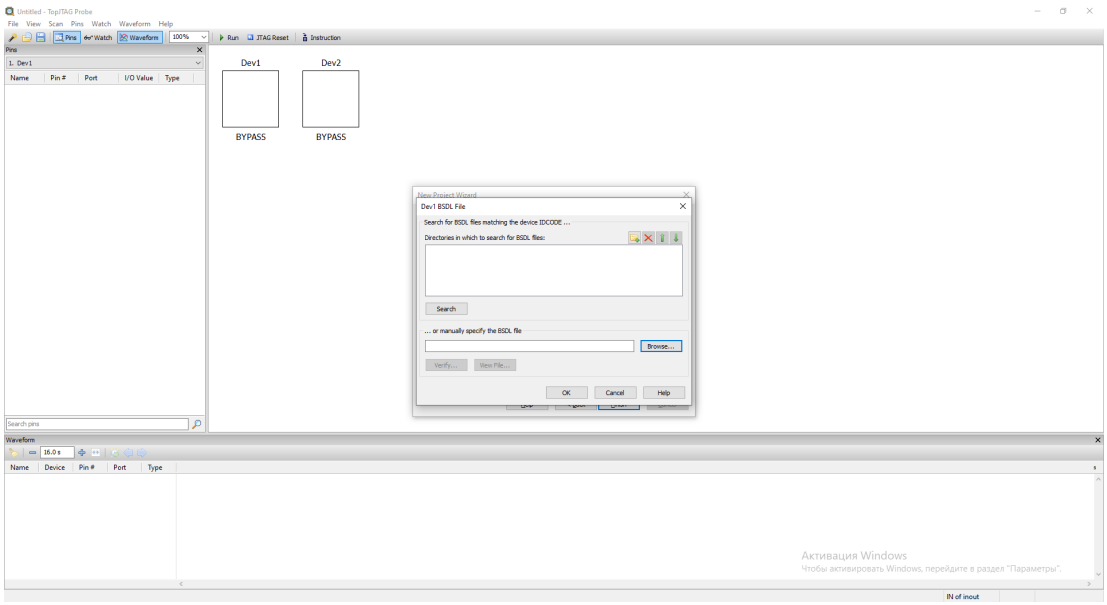


Figure 5.4 - The .bsd file choosing

After that we press *Finish* and now the program is ready to work.

Appearance of the program:

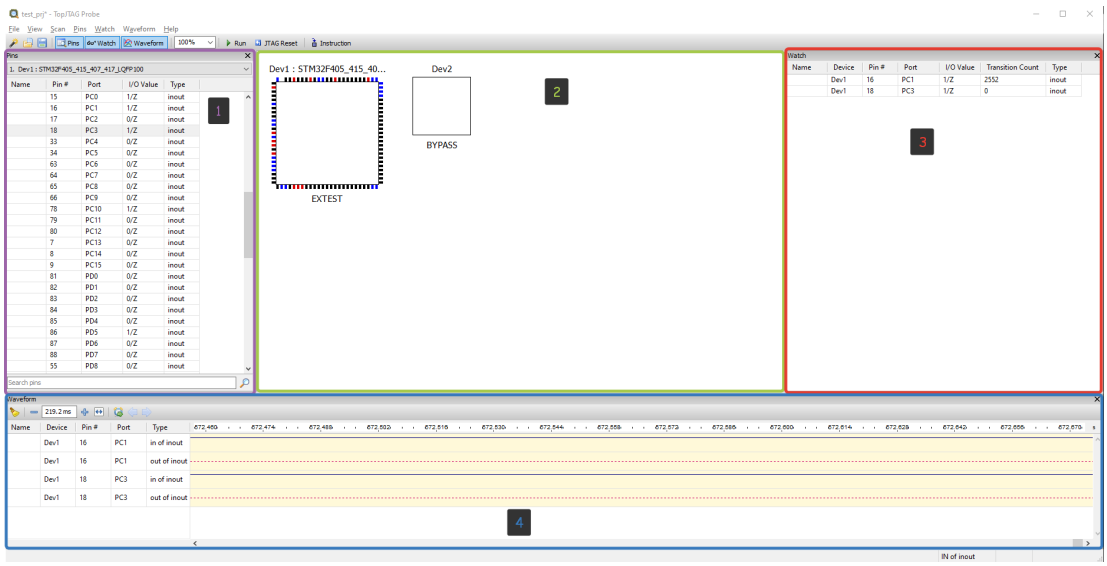


Figure 5.5 - The project screen

1 - Pins Window. The *Pins window* lists all pins belonging to the selected device. Any pin can be quickly located by typing its name, number, or

port name into the search box. The pin or bus name, PIN, port name, type, and current value are displayed for every pin.

2 - JTAG Chain (Main) View. The *JTAG Chain view* (main program's view) shows package outlines with color-coded pin values. These values are updated while the boundary-scan is running.

A device instruction is displayed below the package outline.

Pin values are color-coded. Black means 0, red — 1, brown — high-Z, and blue — linkage pin. For bidirectional pins it is possible to display either input or output values.

3 - Watch Window. The *Watch window* keeps pins of interest in one place to simplify monitoring and/or control.

4 - Waveform Window. The *Waveform window* records and displays waveforms. This is a wonderful tool to view and analyze a signal's history. It is particularly useful for fast changing signals.

5.2 Board Bring Up

Although assembled PCBs can be tested in production, as far as I know, it is not a mandatory procedure. So when these boards come to the Embedded department, especially when it's the first batch that firmware/embedded developers will work with, the boards should be tested to ensure they work properly before they start working on the firmware. Usually (at least in the case where I worked), this is done by the circuit designer. Still, since the microcontroller is empty, this check is limited to the test that the board does not burn up when turned on and that the power supply system provides all the necessary voltages.

Sometimes, a special test firmware can be prepared specifically for bring-up, which automatically or via the CLI interface can enable/disable board modules for testing. In any case, no matter how it happens, bring-up almost always requires interaction between the embedded and circuit designer and additional work to create test firmware.

However, JTAG Boundary Scan can facilitate this work by allowing the circuit designer to control the microcontroller pins and thus perform a better and deeper analysis of the board without depending on the programmer. It also saves the programmer from having to write special firmware for bring-up. You only need a test board, a .BSDL file for the corresponding microcontroller, a debugger, and a unique program to control the controller pins through GUI. Also, an additional advantage of this approach is universality, i.e., you don't need to write a new version of the test firmware for each new microcontroller. And if you remember that Boundary Scan allows you to check the signal integrity for a multiprocessor chain, where processors may have BGA cases and multilayer boards, and it may be very difficult to get to the contacts with probes.

Let's see what is needed for an elementary bring-up. To begin with, at least, you need to be able to control the microcontroller pins (set to logic one/logic zero) and read the state in which the microcontroller pin is located, and JTAG allows you to do this, and programs like TopJTAG Probe allow you to do it more or less conveniently.

As a board for Bring-Up I will use already familiar from the previous parts STM32F407G-DISC1 with microcontroller STM32F407VG.

5.2.1 GPIO Output Control

Let's control the LEDs on our board. Quite a workable task for bring-up. So we know that the LEDs are connected to pins: `PD12`, `PD13`, `PD14`, `PD15`. They are turned on by setting the pin to a logical one and turned off by a logic zero. To select the required level on a pin, it is necessary to find this pin in the *Pin* window and select the necessary actions from the context menu:

Video 5.6 - GPIO output control

5.2.2 GPIO Input State View

Another necessary operation for such a bring-up is to view the state of the chip pin. This can also be done using the JTAG Boundary Scan and TopJTAG application. You can view the output state either in the *Watch* window or in the *Waveform* window. Let's look at the state of the output to which the button is connected:

Video 5.7 - GPIO input state view

Note that as mentioned in the Chapter 3 there can be several scan cells per pin and here you can see that there are two of them: one for receiving and one for transmitting. And as you can see that when the controller transmits something, the receiving scan cell duplicates this signal, as seen in the video with LEDs, but if the output works only for receiving, the signal is present only on one cell, as seen in the video with the button.

---

6. Security

The JTAG interface is an important tool for debugging and testing embedded systems, providing low-level access to the internal workings of microcontrollers and other integrated circuits. However, this powerful interface also presents significant security threats. In the last chapter we will focus on security issues related to JTAG and the Debug Port.

Disclaimer 1: I am not a cyber-security expert and not deeply immersed in this topic, so this article does not aim to uncover any secrets or provide an in-depth exploration. It is a brief discussion from a standard firmware engineering perspective.

Disclaimer 2: This chapter is for informational purposes only and does not encourage hacking

6.1 Protection

Let's start by considering simple protection methods that can be applied to make life more difficult for an attacker who wants to gain unauthorized access to a device using the JTAG interface.

6.1.1 Board-Level

The simplest, most obvious, but probably least effective method of protection is to restrict access to the JTAG connector at the board level. This restriction is achieved by physically removing the JTAG connector from the board at the end of production, as shown in this picture, where you can see that the JTAG connector was removed:

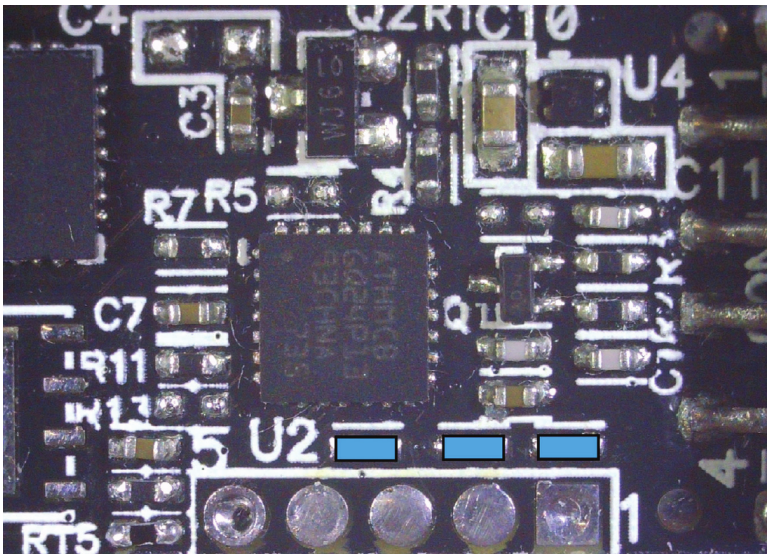


Figure 6.1 - Board with removed JTAG connector

Another protection option is suitable when there is no JTAG connector on the board at all and the JTAG pins are scattered across the board as test points. For example, in the next picture, you can see that the board contains JTAG pins as separate pads.

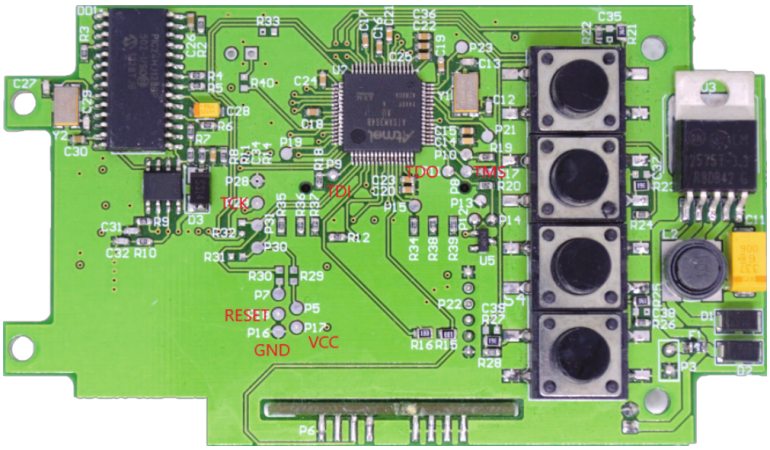


Figure 6.2 - JTAG pins as separate pads

However, as you well understand, these methods provide only the most basic protection, as soldering a JTAG connector is not a significant effort, nor is finding the necessary JTAG pins on the microcontroller, especially when its part number is written on the package, as shown in the photo. But even if the microcontroller is unknown, there are ways to

scan the pins of the microcontroller to discover the necessary JTAG pins (I will discuss these methods in more detail later).

6.1.2 Chip-Level

Debug interfaces are convenient during device development and debugging, but it is good practice to disable them in the release version of the device/firmware. There are configurable bits in the microcontroller's non-volatile memory to disable certain debug interface functions. The flexibility of configuring protection features depends on the specific core and microcontroller model.

For instance, access to flash memory can be disabled. Some microcontroller models support completely disabling the debug interface. The exact configuration of possible protection features can be found in the microcontroller's documentation. An example is the Flash Readout Protection (RDP) technology in STM32 microcontrollers. This technology allows for the protection of the contents of the microcontroller's embedded flash memory from being read through the debug interface.

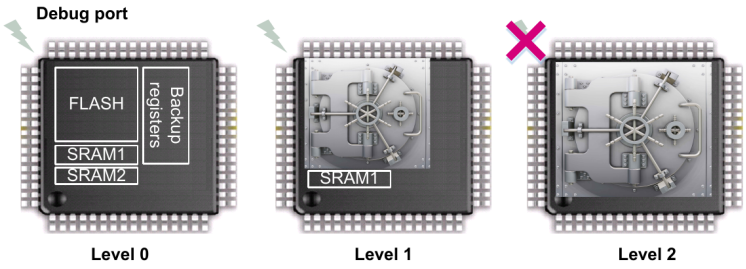


Figure 6.3 - RDP protection levels

RDP has three levels of protection (0, 1, and 2):

- **Level 0** (default RDP level) - The flash memory is fully open, and all memory operations are possible in all boot configurations (debug features, boot from RAM, boot from system memory bootloader, boot from flash memory). There is no protection in this

configuration mode that is appropriate only for development and debug.
- **Level 1** - Flash memory accesses (read, erase, program), or SRAM2 accesses via debug features (such as serial-wire or JTAG) are forbidden, even while booting from SRAM or system memory bootloader. In these cases, any read request to the protected region generates a bus error. However, when booting from flash memory, accesses to both flash memory and to SRAM2 (from user code) are allowed.
- **Level 2** - All protections provided in Level 1 are active, and the MCU is fully protected. The RDP option byte and all other option bytes are frozen, and can no longer be modified. The JTAG, SWV (single-wire viewer), ETM, and boundary scan are all disabled.

RDP can always be leveled up. Increasing the level is necessary in the following cases:

- **Intellectual property protection**: Increasing the RDP level prevents reading the contents of the flash memory, which protects the firmware and other important data from being copied or analyzed by third parties.

- **Prevention of unauthorized firmware modification**: At higher protection levels (e.g., Level 2), it becomes impossible not only to read but also to write to flash memory without completely resetting the microcontroller. This helps prevent the introduction of malicious modifications to the software.

- **Debugging protection**: When the RDP level is raised, the microcontroller disables debugging interfaces (e.g., JTAG or SWD), preventing attackers from using them to debug and analyze the program's operation.

- **Compliance with safety standards**: In some applications (e.g., automotive or medical technology), compliance with certain safety standards is required, which necessitates hardware-level data protection.

A level regression is possible with the following consequences:

- Regression from RDP level 1 to RDP level 0 leads to a flash memory mass erase and the erase of SRAM2 and backup registers. You may need to perform a level regression to update software already in flash memory or when you need to repair a production device that has been returned.
- In RDP level 2, no regression is possible.

In consumer products, RDP should always be set to at least level 1. This setting prevents basic attacks through the debug port or bootloader. RDP level 2 is mandatory for implementing applications with a higher level of security.

The RDP technology for STM32 is an example; similar protection technologies exist for other microcontrollers.

6.2 Attack

Naturally, if there are defense mechanisms against the use of debugging interfaces, there are also well-known vulnerabilities in these defense mechanisms. Let’s consider the main and most well-known of them.

In general, all attacks can be divided into the following types:

- Software attacks are carried out in particular by exploiting bugs, weaknesses in protocols or untrusted code fragments. Attacks on communication channels (interception or usurpation) fall into this category. Software attacks account for the vast majority of cases. Their cost can be very low. They can be widespread and repeatable with tremendous damage. It is not necessary to have physical access to the device. The attack can be performed remotely.
- Hardware attacks require physical access to the device. The most obvious one uses the debug port if it is not secured. In general, however, hardware attacks are complex and expensive. They are carried out with specialized materials and require electronics skills. There is a distinction between non-invasive attacks (carried out at the board or chip level without destroying the device) and invasive

attacks (carried out at the device and silicon level with the destruction of the package).

Attacks on the JTAG interface belong to the hardware none-invasive attack type.

6.2.1 Detecting JTAG pins

If there is absolutely no documentation for a given chip, the first thing an attacker would do is detect the actual JTAG pins. Special tools leverage some features of the JTAG protocol to automate the detection process. Let's dive into them.

6.2.1.1 IDCODE Scan

Feature one: very often the `IR` register is connected to the `IDCODE` register, which means that the intended pinout can be tested by a method called **IDCODE Scan**. This method is simple because there is no need to scan `IR` and the `IDCODE` register has a fixed length of 32 bits. The algorithm of the method is as follows:

1. First we select the pins we want to scan and put these pins in correspondence with JTAG lines (`TMS`, `TCK`, etc.), by luck, as you want. But, if there are too many pins to be tested, you can potentially reduce their number by doing some work with measuring the electrical parameters (resistance, voltage) of the tested pins and using these measurements to draw conclusions about the pin belonging to one or another JTAG signal, at least to determine GND and VCC and not to include these pins in further scanning. This procedure is shown very well in this video clip.
2. Generate the sequence `0b0100` on the selected `TMS` pin to move to the `Shift-DR` state.
3. Hold `TMS` in logic zero and generate 32 clock pulses on the `TCK` pin.
4. Read the bits to be advanced on `TDO`.
5. If more than one zero or one is read from `TDO`, this is a pretty good indication that the electrical JTAG connection is working properly. You can try googling the read `IDCODE` and with some luck, you will

be able to determine not only the JTAG pins, but also the type of microcontroller. If the TDO pin reads only ones or zeros, you have incorrectly assigned the pins in step 1, and you need to reassign the pins and repeat the whole procedure.

However, this method has a number of disadvantages:

- This method does not find the TDI pin.
- If the JTAG circuit contains more than one TAP, it may take more than 32 iterations to find IDCODE, which means that if after e.g. 64 iterations you do not get a TDO pattern similar to IDCODE on the TDO pin, it does not mean that you have selected the wrong pins; it is possible that the JTAG circuit contains many TAPs.

6.2.1.2 BYPASS Scan

This method uses the BYPASS register to scan the JTAG circuit.
The BYPASS register allows you to skip a logic element in the circuit and directly transfer data from input to output, minimizing latency.

1. The first step is the same as the first step in the IDCODE scan method.
2. Send the sequence 0b01100 on the selected TMS pin to move it to the Shift-IR state.
3. Send the BYPASS command. Since the size of the IR register is not known, you have to send a large number of ones (as many as 1024, for example).
4. Send the sequence 0b11000 to the selected TMS pin, thus moving to the Shift-DR state.
5. Using the TDI output - send some bit pattern that can be easily recognized.
6. Read the bits from the TDO pin. If a bit pattern is detected, it means that you have selected the pins correctly.

This algorithm is slower than the previous one, but it can detect all JTAG pins.

If you have a lot of candidate output, it is recommended to first reduce the number of candidates using `IDCODE Scan` and then refine `BYPASS Scan`

Of course every time to do all these steps manually is quite boring and tedious so smart people have long ago automated this process. Quite popular tools are: JTAGenum and JTAGULATOR.

If the highest level of RDP 2 protection has been applied to the microcontroller, JTAGulator and similar tools will not help

Examples of such attacks are described in more detail in the following sources:

- JTAG Scanning
- Hardware Debugging for Reverse Engineers Part 2: JTAG, SSDs and Firmware Extraction
- JTAGulator: Introduction and Demonstration
- 03 - How To Find The JTAG Interface - Hardware Hacking Tutorial

---

Appendix A: ARM Debug Access Port

For successful debugging, it's not enough to simply know the basics of JTAG. It's also essential to understand the workings of the ARM Debug Access Port, a key component that allows access to the controller's internal registers and memory. I've detailed this module in this Appendix.

The DAP is the primary "unit" of the ARM Debug Interface (ADI).

DAP is a module that contains a set of registers and logic required to provide microcontroller debugging via a debug interface. DAP consists of two main components: **Debug Port (DP)** and **Access Port (AP)**.

Currently, most processors are implementing ADIv5 (specified in Arm IHI0031E), while the newer ADIv6 (see Arm IHI0074C) is being slowly phased in. Because it is more popular, I'll be focusing here on the ADIv5 standard.

The **Debug Port** provides control of the DAP and establishes a connection between the debugger and the target device. DP performs functions such as initialization, reset, and operation status checking. This module is conditionally divided into two parts: common and specific to a particular debugging protocol. The common part contains registers used to configure and operate the DP module, and the specific part is responsible for working with external debugging protocols. According to the documentation, our microcontroller supports two debugging protocols: SWD and JTAG, but in this series of articles, I decided to choose JTAG for consideration. Therefore, in the future, unless specifically stated otherwise, I will describe the operation of DAP specifically in the context of the JTAG protocol.

The **Access Port** provides access to various resources of the microcontroller, such as memory. Access Port can have several data exchange interfaces, such as APB (Advanced Peripheral Bus), AHB

(Advanced High-performance Bus), and AXI (Advanced eXtensible Interface), which are used to access various device resources. That is, access to all modules connected to the AHB bus, for example, will be provided through a special AHB-AP.

The approximate structural diagram of DAP is shown in the following figure:

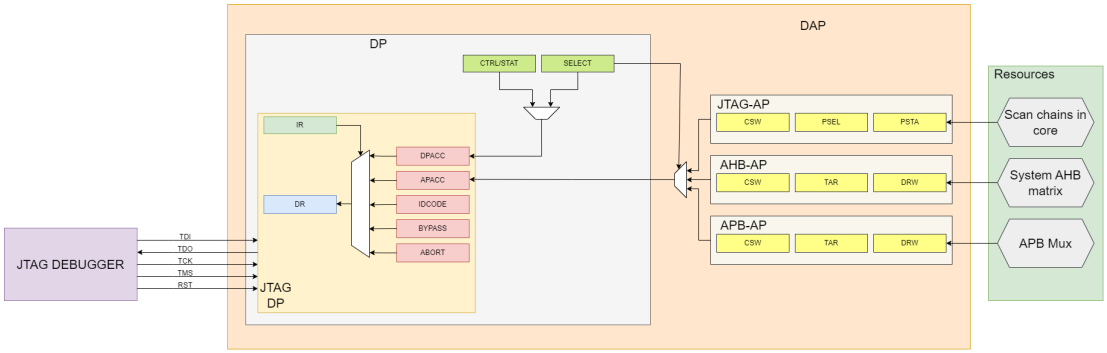


Figure A.1 - Debug Access Port (DAP)

A.1 The external interface, the Debug Port (DP)

A.1.1 JTAG Debug Port (JTAG-DP)

Since JTAG-DP must support the operation of the microcontroller via the JTAG protocol, external connection is made through the JTAG standard pins:

JTAG-DP pin	JTAG pin	Direction	Description
DBGTDI	TDI	INPUT	Debug Data In
DBGTDO	TDO	OUTPUT	Debug Data Out
TCK	TCK	INPUT	Debug Clock
DBGTMS	TMS	INPUT	Debug Mode Select
DBGTRSTn	TRST	INPUT	Debug TAP Reset

Table 1 - JTAG pinout

The JTAG-DP wire-level interface is through scan chains and comprises:

- a Debug TAP State Machine (`DBGTAPSM`) - This state machine is the cornerstone of the JTAG protocol and will be discussed in detail in the next article dedicated to this protocol.
- an Instruction Register (`IR`) and associated IR scan chain, used to control the behavior of the JTAG-DP and the currently-selected data register. The specific data register we want to work with is selected by writing various commands to this register.
- several Data Registers (`DR`s) and associated DR scan chains.

A.1.1.1 JTAG Registers

Instruction Register (IR)

A Debug TAP State Machine is implemented with a 4-bit instruction register (`IR`).

The Instruction Register contains the current command for the DAP controller. This register serves as a kind of entry point when working with DAP over the JTAG protocol. You need to write a command to this register that will determine the data register you want to work with.

The list of commands for the `IR` register is presented in the following table:

IR instruction value	Data register name	DR length	Description
b1000	ABORT	35	The JTAG-DP Abort Register
b1010	DPACC	35	The JTAG-DP DP Register
b1011	APACC	35	The JTAG-DP AP Register
b1110	IDCODE	32	Device ID Code Register
b1111	BYPASS	1	The JTAG-DP Bypass Register

Table 2 - List of JTAG commands supported by DAP

- **ABORT Instruction:**

  The `ABORT` command in JTAG-DP is used to interrupt the execution of the current debugging operation and return control to the debugger. It allows performing a so-called *debug reset* of the processor.

  During the debugging of a processor, the debugger establishes a connection with JTAG-DP and sends commands to read and write registers, execute instructions, etc. In some situations, there may be a need to interrupt the execution of the current debugging operation and return control to the debugger.

  For example, in case of program hang or when there is an error in the debugging script. In such situations, the debugger can use the `ABORT` register to perform a *debug reset* of the processor and return control to the debugger.

- **BYPASS Instruction:**

  The `BYPASS` command connects the TDI and TDO pins directly through a one-bit register called `BYPASS`. This command can be

used to bypass some components connected to the JTAG chain. It may not be very clear now, but this command is closely related to the operation principle of the JTAG protocol and it's difficult to explain it without discussing this principle. The next article will be dedicated to the JTAG protocol, so let's postpone the explanation of this command for now. It's not necessary for understanding the information in the current article.

- **IDCODE Instruction:**

  The `IDCODE` command is used to identify the microchip connected to the JTAG chain. This command connects the `IDCODE` register, which contains a unique 32-bit identifier for the microchip, as the data register.

  Reading the `IDCODE` register can be useful for verifying that the microchip corresponds to the expected model or version, as well as for checking the integrity and proper functioning of the JTAG chain.

- **DPACC Instruction:**

  The `DPACC` command connects the 35-bit `DPACC` (Debug Port Access) register as the data register, which is used to access some Debug Port registers.

- **APACC Instruction:**

  The `APACC` command connects the 35-bit `APACC` (Access Port Access) register as the data register, which is used to access registers of the selected Access Port.

Data Register (DR)

The Data Register is a JTAG register through which all communication with one of the five registers listed in Table 1 occurs. In other words, by specifying the code of one of the five registers in the `IR` register, you are connecting the `DR` register to the selected register, and any read or write operation to the `DR` register will be mapped to the selected register.

A.1.1.2 Debug Port Registers

The DP is responsible for configuring the debug module, controlling power and reset, and monitoring various errors that may occur during DAP operation. Special registers, such as `ABORT`, `IDCODE`, `CTRL/STAT`, and `SELECT`, is responsible for all of these tasks.

- **Abort Register, ABORT:**

  The Abort Register is always present on all DP implementations. Its main purpose is to force a DAP abort.

Bits	Access	Function	Description
[1:31]	RO	-	Reserved
[0]	WO	DAPABORT	Write 1 to this bit to generate a DAP abort. This aborts the current AP transaction. Do this only if the debugger has received WAIT responses over an extended period.

  Table 3 - DP ABORT register bits

  When working with this register within JTAG-DP, the only writable bit is the `DAPABORT` bit. Writing to the other bits of this register results in undefined behavior according to the documentation.

- **ID Code Register, IDCODE:**

  The Identification Code Register is always present on all DP implementations. It provides identification information about the ARM Debug Interface.

Bits	Access	Function	Description
[31:28]	RO	Version	Version code
[27:12]	RO	PARTNO	Part Number for the JTAG-DP - 0xBA00
[11:1]	RO	DESIGNER	JEDEC Designer ID, an 11-bit JEDEC code that identifies the designer of the ADI implementation - 0x23B
[0]	RO	-	Always 1

Table 4 - DP IDCODE register bits

- **Control/Status Register, CTRL/STAT:**

  The `CTRL/STAT` Register is always present on all DP implementations. It provides control of the DP, and status information about the DP.

Bits	Access	Function	Description
[31]	RO	CSYSPWRUPACK	System power-up acknowledge
[30]	R/W	CSYSPWRUPREQ	System power-up request
[29]	RO	CDBGPWRUPACK	Debug power-up acknowledge
[28]	R/W	CDBGPWRUPREQ	Debug power-up request
[27]	RO	CDBGRSTACK	Debug reset acknowledge
[26]	R/W	CDBGRSTREQ	Debug reset request
[25:24]			Reserved, RAZ/SBZP
[23:12]	R/W	TRNCNT	Transaction counter
[11:8]	R/W	MASKLANE	Indicates the bytes to be masked in pushed compare and pushed verify operations
[7]	RO	WDATAERR	Set if a Write Data Error occurs
[6]	RO	READOK	This bit is set to 1 if the response to the previous AP or RDBUFF read was OK
[5]	RO	STICKYERR	Set if an error is returned by an AP transaction.
[4]	RO	STICKYCMP	This bit is set to 1 when a match occurs on a pushed compare or a pushed verify operation

Bits	Access	Function	Description
[3:2]	R/W	TRNMODE	Transfer mode for AP operations: 00 = Normal operation, 01 = Pushed verify operation, 10 = Pushed compare operation, 11 = Reserved.
[1]	RO	STICKYORUN	Overrun error flag
[0]	R/W	ORUNDETECT	This bit is set to 1 to enable overrun detection

Table 5 - DP CTRL/STAT register bits

Let's take a closer look at the bits of this register.

Sticky flags and DP error responses

A read or write error might occur within the DAP, or come from the resource being accessed. In either case, when the error is detected the Sticky Error flag, `STICKYERR`, in the `CTRL/STAT` Register is set to 1. A read/write error also might be generated if the debugger makes an AP transaction request while the debug power domain is powered down.

If a debugger issue `APACC`s too fast, overrun may happen, the DP can be programmed so that if an overrun error occurs, the DP set the `STICKYORUN` flag in the `CTRL/STAT` register. But if overrun detection is on, the debugger must check for overrun errors after each sequence of APACC transactions, the DP will also no longer send FAULT response and WAIT response.

In any case, if a debugger receives a FAULT response from the DP, it must read the `CTRL/STAT` register to check the sticky flags.

The transaction counter

The DP includes an AP transaction counter, TRNCNT, which enables a debugger to make a single AP transaction request to generate a sequence of AP transactions, thus accelerating code download or memory fill operations. The TRNCNT maps onto the CTRL/STAT[23:12] bitfield, and writing a value N (N>=0) to this field generates N+1 AP transaction(s) . If TRNCNT is not zero, it is decremented after each successful transaction, but it is not decremented when there are any sticky flags set. When reaches zero, TRNCNT does not auto-reload.

Power control

This is to enable an external debugger to connect to a potentially turned-off system and power up as much as required to get a basic level of debug access with a minimal understanding of the system.

The DAP model supports multiple power domains; there can be three power domains:

- Always on domain
- System power domain
- Debug power domain

The DP registers reside in the always-on domain, and there are two control bits in the CTRL/STAT register:

- Bit [28], CDBGPWRUPREQ, used to request the system's power manager to fully power up and enable clocks in the debug power domain.
- Bit [30], CSYSPWRUPREQ, used to request the system's power manager to fully power up and enable clocks in the system power domain.

Both bits need to be set during initialization, to ensure the MCU is fully powered up and clocks are enabled.

Debug Reset Control

The DP CTRL/STAT register provides bits [27:26], for reset control of the debug domain. The debug domain controlled by these signals covers the internal DAP and the connection between the DAP and the debug components, for example, the debug bus. The two bits provide a debug reset request, CDBGRSTREQ, and a reset acknowledge, CDBGRSTACK., and the associated signals provide a connection to a system reset controller. The DP registers are in the always-on power domain on the external interface side of the DP. Therefore, the registers can be driven at any time, to generate a reset request to the system reset controller.

- **AP Select Register, SELECT:**

  The AP Select Register is always present on all DP implementations. Its main purpose is to select the current Access Port (AP) and the active four-word register bank within that AP.

Bits	Function	Description
[31:24]	APSEL	Selects the current AP (AHB-AP, APB-AP, JTAG-AP)
[23:8]	-	Reserved
[7:4]	APBANKSEL	Select the active four-word register bank on the current AP
[3:1]	-	Reserved
[0]	-	Reserved

Table 6 - DP SELECT register bits

AP selection

As DAP can contain multiple APs, we need to have the ability to select a specific AP with which we want to work at the moment. And it is the APSEL bits that are responsible for this selection. They specify the number of the AP that will be used for access.

Bank of registers selection

The APBANKSEL bits are used to select the register bank to which access will be made. Register banks are used to organize registers associated with specific functional blocks within the microcontroller. For example, a register bank may be allocated for the core processor debug registers, while another bank may be allocated for the peripheral device debug registers.

A.1.1.3 Accessing the DP registers

As mentioned earlier, to access the DP registers CTRL/STAT and SELECT, the DPACC register is used, and working with this register has its own peculiarities. Let's take a closer look at the algorithm for accessing the DP registers.

DP registers can be divided into two categories: control registers and access registers. Control registers contain information about the state of the debug logic, such as the status of command execution or the reset state. Access registers are used to read and write data from registers that control the operation of the debug logic.

In this context, we will focus on the DP access registers, which provide access to the debug logic. DP access registers can be addressed through the DPACC register and include control registers (CTRL/STAT and SELECT), access registers to the application logic (APACC), bypass register (BYPASS), and device identification register (IDCODE). Working with these registers has its own peculiarities and requires knowledge of the DPACC register format, which we will discuss further.

DPACC register format

The DPACC register has a size of 35 bits and its format depends on the operation (read or write) we want to perform.

DPACC register format:

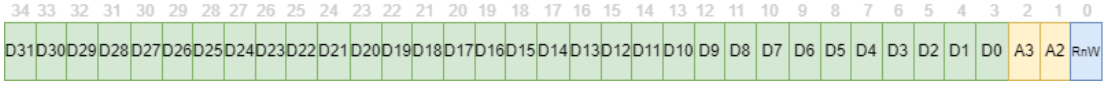


Figure A.2 - DPACC register write format

- D - these are data bits. That is, in these 32 bits, you store the value you want to write to the selected register.
- A - these are address bits of the DP register. These two bits determine the specific DP register you want to work with:

Address	A[3:2]	Register
0x04	[01]	CTRL/STAT
0x08	[10]	SELECT

Table 7 - A[3:2] bits for DPACC

`DR` register in JTAG-DP is like a common register for five data registers (`ABORT`, `BYPASS`, `IDCODE`, `DPACC`, `APACC`), and by writing or reading data from/to it, you write/read data to/from one of the five data registers selected by the command in the `IR` register. Similarly, for working with the `CTRL/STAT` and `SELECT` registers, when you select the `DPACC` register for operation with the `DPACC` command in the `IR` register, the `DR` register is mapped to the `DPACC` register, and the `DPACC` register is mapped to one of the shared `DP` registers: `CTRL/STAT` or `SELECT`. This results in a chain of registers: `DR --> DPACC --> CTRL/STAT` or `SELECT`, for example. Therefore, by writing/reading data to/from the `DR` register, you write/read data to/from the `CTRL/STAT` or `SELECT` register.

- RnW - this bit for the write operation has the value 0.

Format for reading:

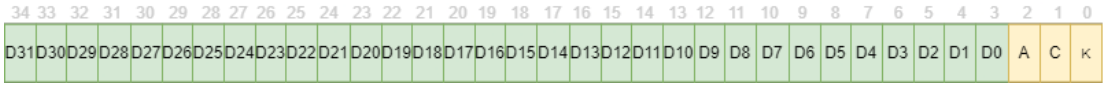


Figure A.3 - DPACC register read format

- D - these are data bits. I.e. in these 32 bits you get the value read from the register.
- ACK[2:0] - Acknowledge of read operation:

ACK	Value
0b010	OK/FAULT
0b001	WAIT

Table 8 - Acknowledge codes

Operation algorithm

Now let's step by step go through the algorithms for writing/reading a new value to/from the DP register:

Algorithm for writing:

- Write the value `0b1010` (the code for the `DPACC` register) to the `IR` register.
- Write 35 bits to the `DR` register according to the format described above:
  - DATA[31:0] - the data we want to write
  - A[3:2] - the selected register
  - RnW - `0b0`

Algorithm for reading:

- Write the value `0b1010` (the code for the `DPACC` register) to the `IR` register.
- Write a **request** to read the register to the `DR` register:
  - DATA[31:0] - `0x00`
  - A[3:2] - the selected register
  - RnW - `0b1`
- Read 35 bits from the `DR` register:
  - DATA[31:0] - the data read from the register
  - ACK[2:0] - Acknowledge code.

As you may have noticed, during the read operation, we access the DR register twice. The first time, we form a **request** to read data from a specific register, i.e., we inform the microcontroller that we would like to read data from a particular register. The actual reading of the data happens during the second access to the DR register.

A.2 The resource interface, the Access Ports (AP)

Access ports provide access to system resources such as debug registers, trace port registers, a ROM table, or system memory.

An ARM Debug Interface always includes at least one Access Port, and might contain multiple APs. The simplest ARM Debug Interface uses a single AP to connect to a single debug component.

Multiple APs can be added to the DAP, depending on the needs. ARM provides specifications for two APs :

- **Memory Access Port (MEM-AP)**. This AP provides access to the core memory and registers.
- **JTAG Access Port (JTAG-AP)**. This AP allows to connect a JTAG chain to the DAP.

For this series of articles, the Memory Access Port is of the greatest interest, so we will focus on it. For information about the JTAG Access Port, please refer to the documentation.

A.2.1 Memory Access Port Registers

The specification for ARM DAP describes the following Memory Access Port registers:

MEM-AP register	Address	Register bank (APBANKSEL)	Offset (A[3:2])
CSW	0x00	0x00	0b00
TAR	0x04	0x00	0b01
DRW	0x0C	0x00	0b11
BD0	0x10	0x01	0b00
BD1	0x14	0x01	0b01
BD2	0x18	0x01	0b10
BD3	0x1C	0x01	0b11
CFG	0xF4	0x0F	0b01
BASE	0xF8	0x0F	0b10
IDR	0xFC	0x0F	0b11

Table 9 - Memory Access Port registers

Control/Status Word Register, CSW)

The `CSW` register is used to configure the access mode to the memory and peripheral devices of the ARM microcontroller.

Using the `CSW` register, you can configure parameters such as the size of the transmitted data (8, 16, or 32 bits), enable the auto-increment mode of the memory address to which access is made, and so on.

The Transfer Address Register (TAR)

This register is used to specify the address to be accessed through the AP. This address can be the address of a register, memory, or any other available address on the device.

When a new value is set in `TAR` in the AP, all subsequent read/write operations will be performed at the specified address until `TAR` is changed to another address and the auto-increment mode is turned off.

In general, before accessing any address through the AP, it is necessary to load this address into the `TAR`.

The Data Read/Write Register (DRW)

The DRW (Data Register for Write) in the Access Port is used for writing data to or reading data from the target resource.

To write data to DRW, the address needs to be set in the TAR (Transfer Address Register), and the necessary access parameters need to be configured in CSW. After that, data can be written to DRW and transferred to the target resource. Similarly, to read data from the target resource, the access parameters in CSW need to be configured and the read address needs to be set in TAR. Then, the data from the target device will be transferred to DRW, ready to be read.

Banked Data Registers 0 to 3 (BD0 to BD3)

The BD0, BD1, BD2, and BD3 registers in the Access Port are used for transferring/receiving data between the target device and the debugging system.

Using these registers can be useful when a large amount of data needs to be transferred or when a higher data transfer speed is required than what is possible with the TAR and DRW registers. Together, these four Banked Data Registers provide access to four words of memory space, starting at the address specified in the TAR. However, transferring data through the BD0, BD1, BD2, and BD3 registers requires more complex logic than transferring through the TAR and DRW registers and may require additional software to manage these registers.

Configuration Register (CFG)

The CFG Register holds information about the configuration of the MEM-AP. In particular, it indicates whether accesses to the connected memory system are big-endian or little-endian.

Debug Base Address Register (BASE)

The BASE Register is a pointer to the connected memory system. It points to one of: - the start of a set of debug registers for the single connected debug component - the start of a ROM Table that describes the connected debug components

Identification Register (IDR)

The IDR identifies the Memory Access Port. The IDR is the only register that must be implemented by any Access Port.

A.2.3 Addressing of AP Registers

One of the four registers within the DP is the AP Select Register (SELECT). This register specifies a particular Access Port, and a bank of four 32-bit words within the register map of that AP.

In any AP access transaction from the debugger, the two address bits A[3:2] are decoded to select one of the four 32-bit words from the register bank indicated by the SELECT register (see Table 9). In other words, they select a specific register within the selected four-register bank.

This access model is shown in Figure 4 shows how the contents of the SELECT register are combined with the A[3:2] bits of the APACC scan-chain to form the address of a register in an AP.

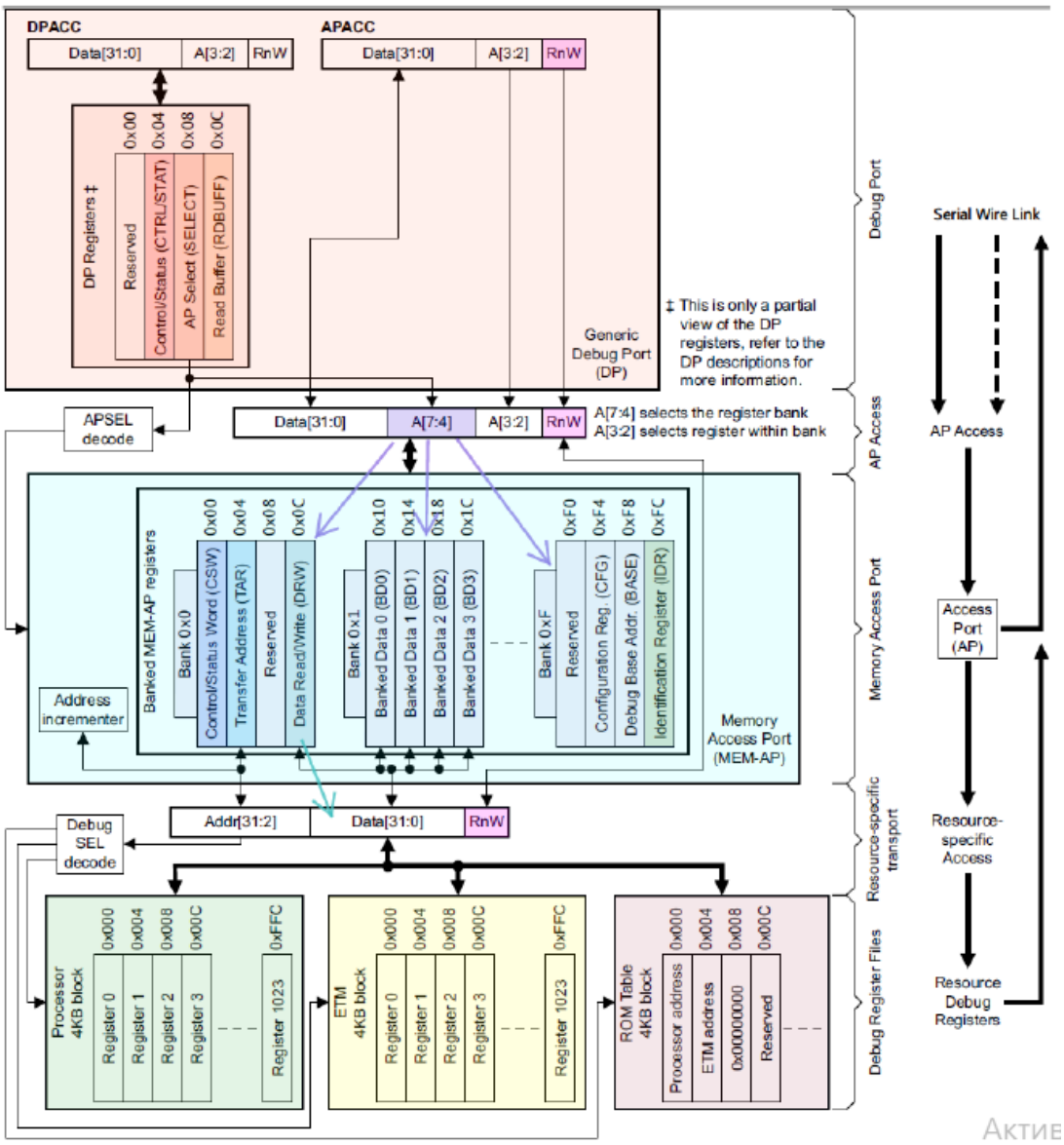


Figure A.4 - MEM-AP access model

As can be seen in this figure, Memory Access Port registers are all 32 bit long (4 bytes), two fields in the DP `SELECT` register select the AP (`APSEL` field) and the register bank in the AP (`APBANKSEL` field), and finally the `A[3:2]` field of `APACC` specify the exact AP register in the bank.

A.2.2 Accessing the AP registers

The situation with accessing the AP registers is the same as accessing the DP registers, except that instead of the `DPACC` register, the `APACC` register is used. When you write `APACC` command in the `IR` register, the `DR` register is mapped to the `APACC` register, and the `APACC` register is mapped to one of the common AP registers defined in Table 9. This creates a chain of registers: `DR --> APACC --> AP Reg`.

APACC register format

These registers also have a size of 35 bits and the value of the bits depends on whether we write something to these registers or read from them.

Format for writing:

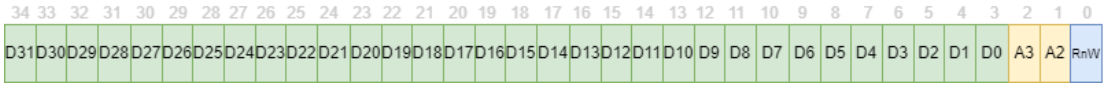


Figure A.5 - APACC register read format

- D - This is the data bits. In these 32 bits, you store the value that you want to write to the selected register.
- A - These are the sub-address bits of the AP register. These two bits, together with four more bits from the DP register `SELECT`, determine the specific AP register you want to work with:

Address	A[3:2]	Register
0x00	[00]	CSW
0x04	[01]	TAR
0x0C	[11]	DRW
0x10	[00]	BD0
0x14	[01]	BD1
0x18	[10]	BD2
0x1C	[11]	BD3
0xF4	[01]	CFG
0xF8	[10]	BASE
0xFC	[11]	IDR

Table 10 - A[3:2] for MEM-AP registers

- RnW - This bit has a value of 0 for a write operation.

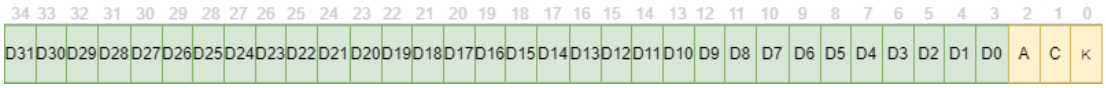


Figure A.6 - APACC register read format

- DATA[31:0]) - These are data bits. In other words, these 32 bits contain the value that you read from the selected register.
- Bits 2:0 (ACK[2:0]) = 3-bit Acknowledge of read operation:
  - 0b010 = OK/FAULT
  - 0b001 = WAIT

Operation algorithm

Now let's step-by-step go through the algorithms for writing/reading a new value to/from an AP register:

Algorithm for writing

- Write the `DPACC` command (`0b1010`) to the `IR` register

- Write 35 bits to the DR register in which we indicate that we are working with the SELECT register and write the corresponding values of the APSEL[31:24] and APBANKSEL[7:4] bits to this register:
  - DATA [31:0]:
    - APSEL[31:24] - select the desired AP
    - APBANKSEL[7:4] - select the desired bank of registers
  - A[3:2] - 0b10 - address of SELECT register (see Table 7)
  - RnW = 0b0
- Write the APACC command (0b1011) to the IR register
- Write the data to the DR register according to the described bit values (see Figure 5):
  - DATA[31:0] = data which need to be written
  - A[3:2] = Address of Memory Access Port register (see Table 9).
  - RnW = 0b0

Algorithm for reading

- Write the DPACC command (0b1010) to the IR register
- Write 35 bits to the DR register in which we indicate that we are working with the SELECT register and write the corresponding values of the APSEL[31:24] and APBANKSEL[7:4] bits to this register:
  - DATA[31:0]:
    - APSEL[31:24] - select the desired AP
    - APBANKSEL[7:4] - select the desired bank of registers
  - A[3:2] - 0b10 - address of SELECT register (see Table 7).
  - RnW = 0b1
- Write the APACC command (0b1011) to the IR register
- Create the data read request. Write in DR next data:
  - DATA[31:0] = 0x00000000
  - A[3:2] = Address of Memory Access Port register (see Table 9).
  - RnW = 0b1
- Read 35 bits from the DR register
- Check the values of the ACK[2:0] bits and if they are not equal to OK/FAULT (0b001), then the data was read, if it is equal to WAIT (0b001), we need to repeat the reading because the data is not ready.
