Making my own Programmer/Debugger using ARM SWD.

June 27, 2023 · 34 min · magalsh64

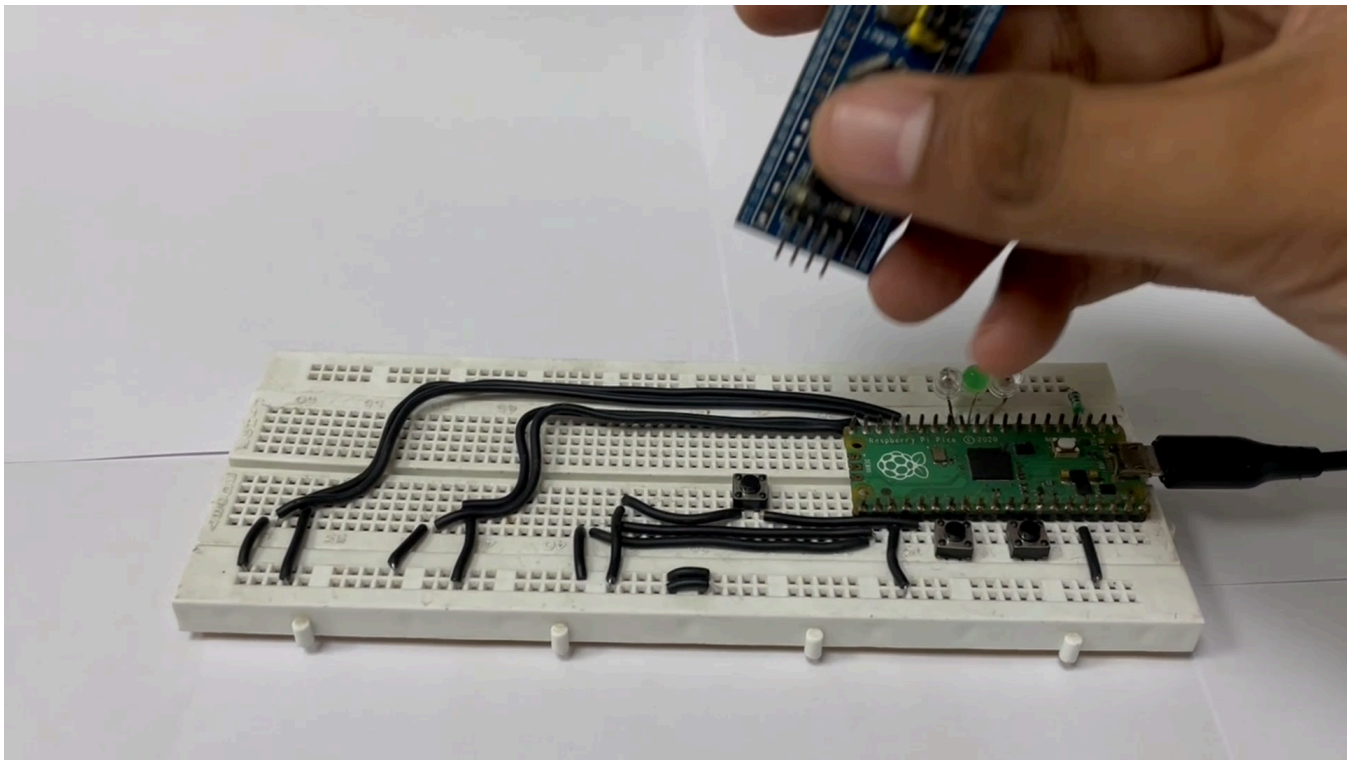
So, recently at my day job our team needed to develop a PCB test JIG for mass manufacturing/testing of the assembled PCBs.

To increase the efficiency of this test process, the idea of testing and programming multiple boards at once came into my mind.

Initially I thought of using multiple Debuggers/Programmers to flash code into multiple boards at once, but as you know, most of these debuggers aren't cheap.

Except for ST-link V2 clones which are available at dirt cheap prices, all other debuggers cost at least 10s of dollars and from then on special mass production versions of J-link can go on for 1000s of dollars not including professional software that you need to purchase separately for flash programming in a manufacturing setting.

So i though well, it's time to dig into ARMs SWD protocol myself and make my own debugger/programmer :)



"A Raspberry Pi Pico being used to program 3 STM32 blue pills at once without the need for any external HOST computer, the firmware to upload is stored in pi pico's internal flash."

What exactly is this JTAG/SWD stuff anyways?

JTAG stands for **Joint Test Action Group**, these are the people who came up with the industry standard for verifying designs and testing of PCBs after they have been manufactured, and that standard is also named JTAG.

Devices (MCUs/SoCs/MPUs) that have JTAG support can give an external tool(debugger/programmer) access to their internal system buses so that they can access all the system resources the same way the internal processor can.

JTAG can also provide other useful features such as boundary scanning, which allows the external tool to directly probe the state of any pins of the device which can be useful to test the system after it has been manufactured.

JTAG, being an industry standard can be used on almost any CPU architecture(or FPGA) depending on whether the manufacturer of the SoC has added support for it

or not.

SWD or Serial Wire Debug is a debugging interface specifically designed by ARM ltd as a part of it's CoreSight Debugging and Trace architecture.

It was developed as a low pin count alternative to JTAG that allowed MCUs/MPUs/SoCs with ARMs Cortex-A/M/R cores to be debugged and provides real-time trace capabilities.

The major problem with JTAG for Microcontrollers was the minimum pin count of 4 (5 if reset was to be included).

Small MCUs with very low pin counts could not afford to give up 4 pins just for the debugging interface.

For basic debugging, SWD has the pin requirement of only 2 pins, one for CLOCK and another for DATA.

If you also want trace capabilities then depending on what method you choose (SWO vs ETM), it can require as low as 1 to as many as 38 extra pins.

The Hardware

For this demo, we will be using a raspberry pi pico as our development platform to make our debugger/programmer, simply because its cheap and readily available.

For the target we will use a STM32 blue pill featuring the infamous STM32F103C8T6 microcontroller.

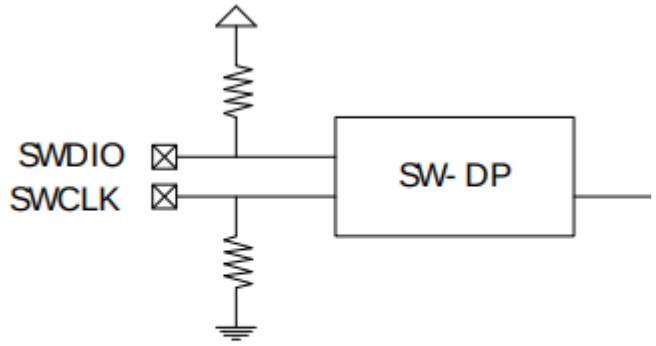
For the DSO to debug the physical layer we will be using the RIGOL 1054Z.

Just to be clear, HOST refers to the debugger and TARGET refers to the device being debugged.

So, lets begin.

The PHY layer

As mentioned above, SWD requires at least 2 pins to function, a CLOCK pin (SWDCLK) and a DATA pin (SWDIO).



The HOST is the only one that drives the CLOCK pin.

The DATA pin can be driven by both the HOST and the TARGET.

The CLOCK pin can have a pull down resistor although it isn't compulsory, it's a good practice, as it attenuates spurious signals when the pin is floating (debugger disconnected).

The DATA pin, if you are using it as an open-drain output must feature a pull-up resistor or else the line capacitance will be enough to corrupt the signal at faster clocks.

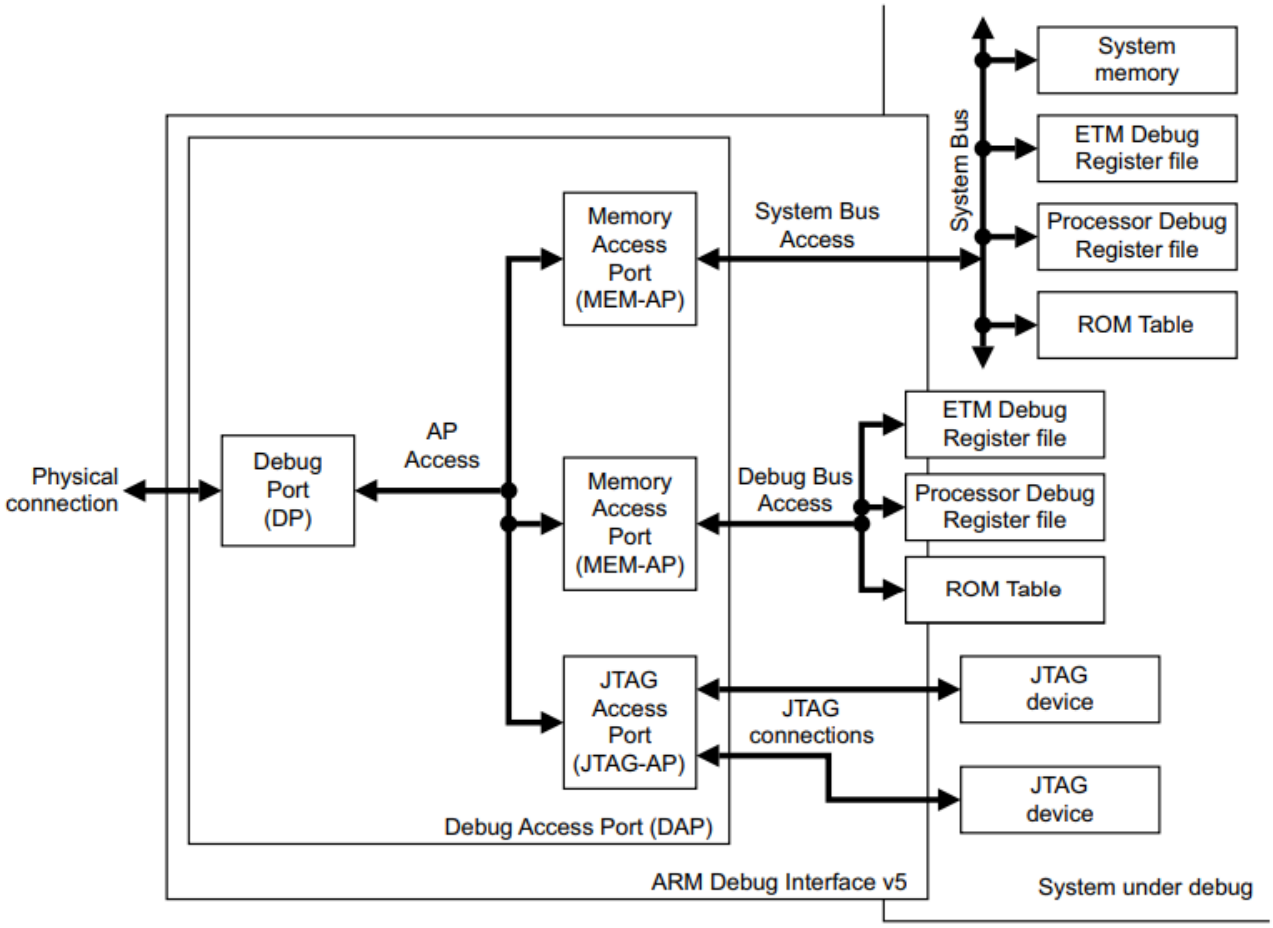
A logic HIGH is interpreted as a '1' and a logic LOW is interpreted as '0' on DATA and CLOCK lines.

The Architecture

SWD just like JTAG enables an external tool (the debugger) to get access to internal components of an MCU/MPU/SoC.

This might include access to the internal buses, special tracing and profiling components such as DWT,ITM and ETM etc.,

So how does this happen?



When you connect you SWD enabled debugger to your MCU/MPU/SoC (from here on referred to as TARGET) via the SWD pins exposed by the said TARGET..

You connect to a component inside the TARGET called **DAP** or **Debug Access Port**.

The DAP is named so because it consists of 2 components, the Debug Port and the Access Port(s).

The Debug Port is the main interface that is connected to the external SWD pins and allows access to the internal structure of the ARM Debug Interface.

Access to various components inside the TARGET are provided in form of Access Ports or APs.

The DP acts as the master with multiple slave APs connected to it.

Both DP and AP have internal memory mapped registers which can be accessed by the external debugger.

1. Access to the DP registers is termed as Debug Port Access (DPACC).
2. Access to the AP registers is termed as Access Port Access (APACC).

The Protocol

The SWD protocol is based on transactions.

Every transaction consists of 3 phases:

1. A host to target packet request
2. A target to host packet response.
3. A data transfer phase, if required, either from host to target or vice-versa depending on the request made in phase 1.

The host initialises any transation with the following packet structure:

```
typedef struct _swd_req
{
    uint8_t start : 1;  // This bit is always 1
    uint8_t APnDP : 1;  // This bis it 0 for DPACC and 1 for APACC
    uint8_t RnW : 1;    // This bit is 0 for Write and 1 for Read
    uint8_t A : 2;      // Has different meaning based on if AP is selected or DP.
    uint8_t parity : 1; // This parity check is done on APnDP,RnW,A bits; If no of 1s
    uint8_t stop : 1;   // This bit must be 0 for synchronous SWD, which means always.
    uint8_t park : 1;   // This bit must be 1.
} swd_req;
```

The main things to notice here are the APnDP bit, the RnW bit and the A[2:3] bits.

1. The APnDP bit allows us to select whether we want to access the DP registers or the AP registers.
2. The RnW bit allows us to specify whether this is a read or write request.

3. The A bits are important as these form a part the address of the register to access within the DP or AP.

Address	Read	Write
0x00	IDCODE	ABORT
0x04	CTRL/STAT [1]	CTRL/STAT [1]
0x08	RESEND	SELECT
0x0C	RDBUFF	N/A

[1] WCR register if CTRLSEL bit of SELECT is 1, see [adi5]

For the DP, the address of the register you want to access is formed in the following way:

1. The Address of the DP registers is a 4bit value.
2. The A bits provide the upper 2 bits for this address, the lower 2 bits are always 0.

Thus the only possible values for the register address for the DP/AP are 0x0,0x4,0x8 and 0xC.

Depending on the state of the RnW bit, and the 4 bit address formed using the A bits, you can select the DP registers you want to access.

Some of these DP registers are read-only, some are write-only and only the CTRL/STAT is both read/write capable.

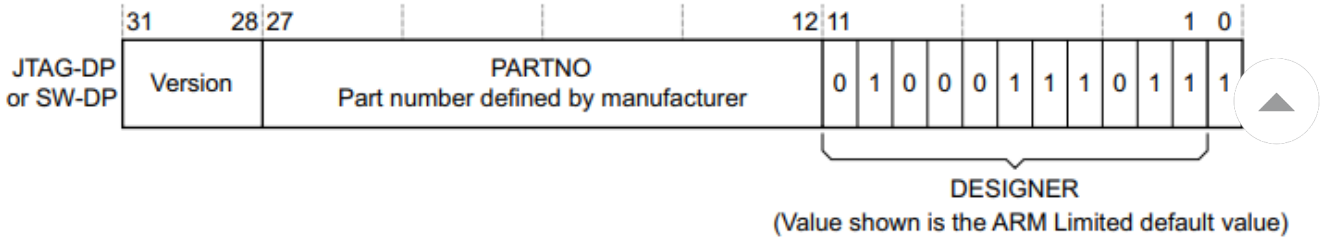
Lets see what some of these registers are:

IDCODE Register:

This register is always present on all DP implementations.

It provides identification information about the Debug Interface.

This is a read-only register and is always accessible, it is the first register the debugger reads when it connects to the TARGET.



This register defines the (DESIGNER), its part number(constant defined by ARM) and the version number.

The DESIGNER field is set by the SoC vendor in the RTL and is usually the assigned JEDEC Manufacturer ID.

The part number is a constant value set by ARM and is either 0xBA00 or 0xBA10.

The version number is also set by the SoC vendor and is used to differentiate between different products by the same vendor.

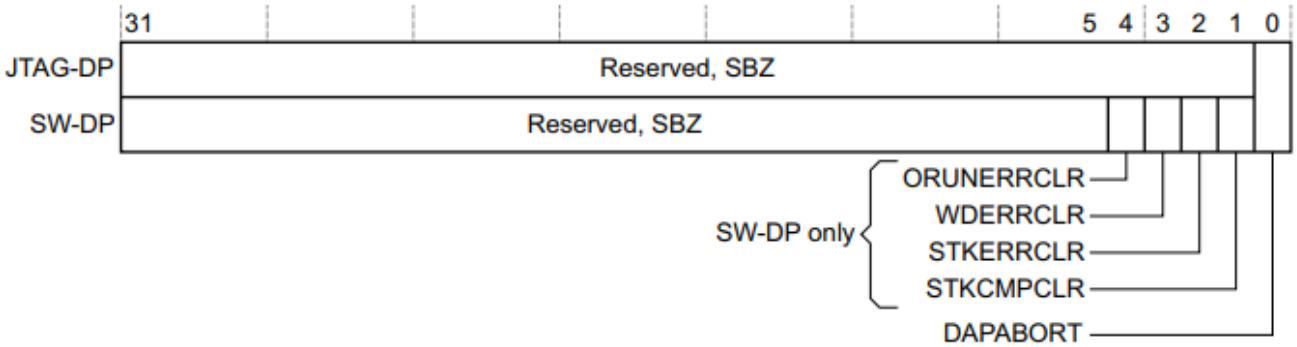
Thus using the IDCODE you can uniquely identify any SoC (If the vendor did his job right).

ABORT Register:

This register is always present on all DP implementations.

It provides the external debugger the ability to force abort any ongoing transaction.

This is a write-only register and is always accessible.



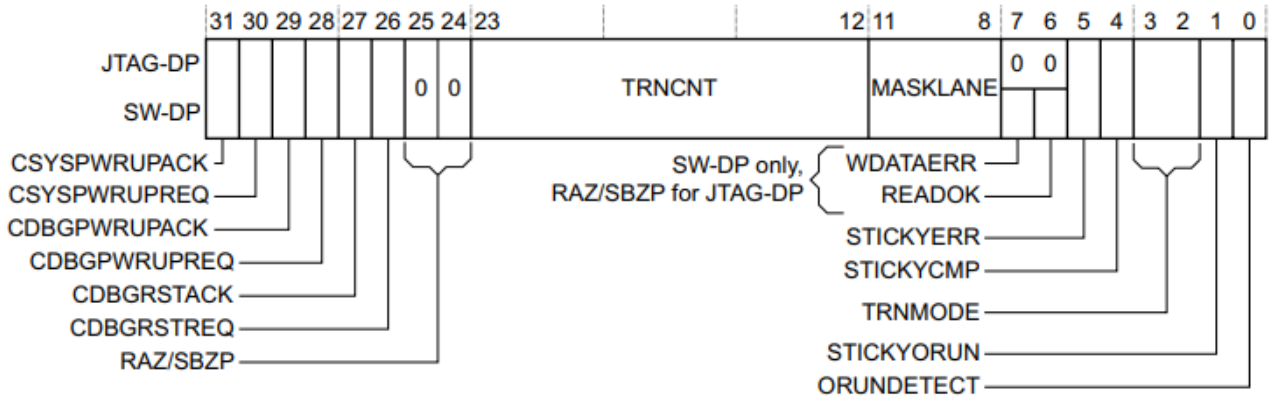
This register also gives the ability to clear any sticky error flags that are set when any fault occurs in the protocol.

CTRL/STAT Register:

This register is always present on all DP implementations.

Its provides control of the DP, and status information about the DP.

It is a read-write register, although some bits are read-only.



The bits of interest of us in this are the CSYSPWRUPREQ bit and the CDBGPWRUPREQ bit.

The DAP power domain model lists 3 major power domains:

1. Always-on power domain.
2. System power domain.
3. Debug power domain.

The DP registers reside in the always-on power domain, thus they are always available even if the rest of the system is in power down/sleep mode.

The rest of the debug system and the core system can go in power down mode in which state we cannot access it via the APs.

Setting the CSYSPWRUPREQ bit and the CDBGPWRUPREQ bit signals these domains to power back up so that the debug system can access them.

CSYSPWRUPACK bit and the CDBGPWRUPACK bit return the response of this request, if successful these are read back as 1.

RDBUFF Register:

This register is always present on all DP implementations.

This is a read-only register.

On a SW-DP, performing a read of the Read Buffer captures data from the AP, presented as the result of a previous read, without initiating a new AP transacti

This means that reading the Read Buffer returns the result of the last AP read access, without generating a new AP access.

RESEND Register:

This register is always present on all DP implementations.

This is a write-only register.

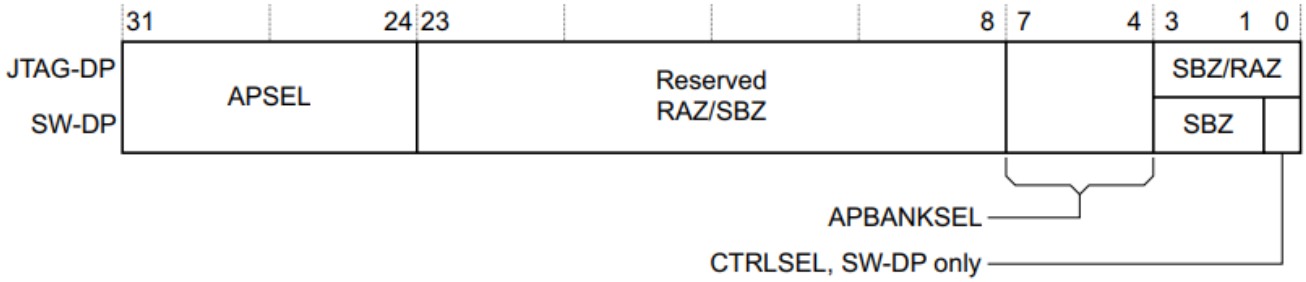
Its purpose is to enable the read data to be recovered from a corrupted debugger transfer, without repeating the original AP transfer.

SELECT Register:

This 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.

This is a write-only register.



As of now, we have only been accessing DP registers, For accessing the AP registers, things are a bit complicated.

As you would remember, the APnDP bit of the SwD request packet selects whether the request to read/write data is for a DP or a AP.

Although there is only one DP, there can be many APs connected as slave to that DP, so how do we select which of these APs we want to read/write to?

Simple! by this SELECT register. The value of APSEL bits selects which AP we want to access.

Furthermore, each AP can have multiple banks, each bank has 4 registers which can be selected by the A[3:2] bits of the swd request packet header.

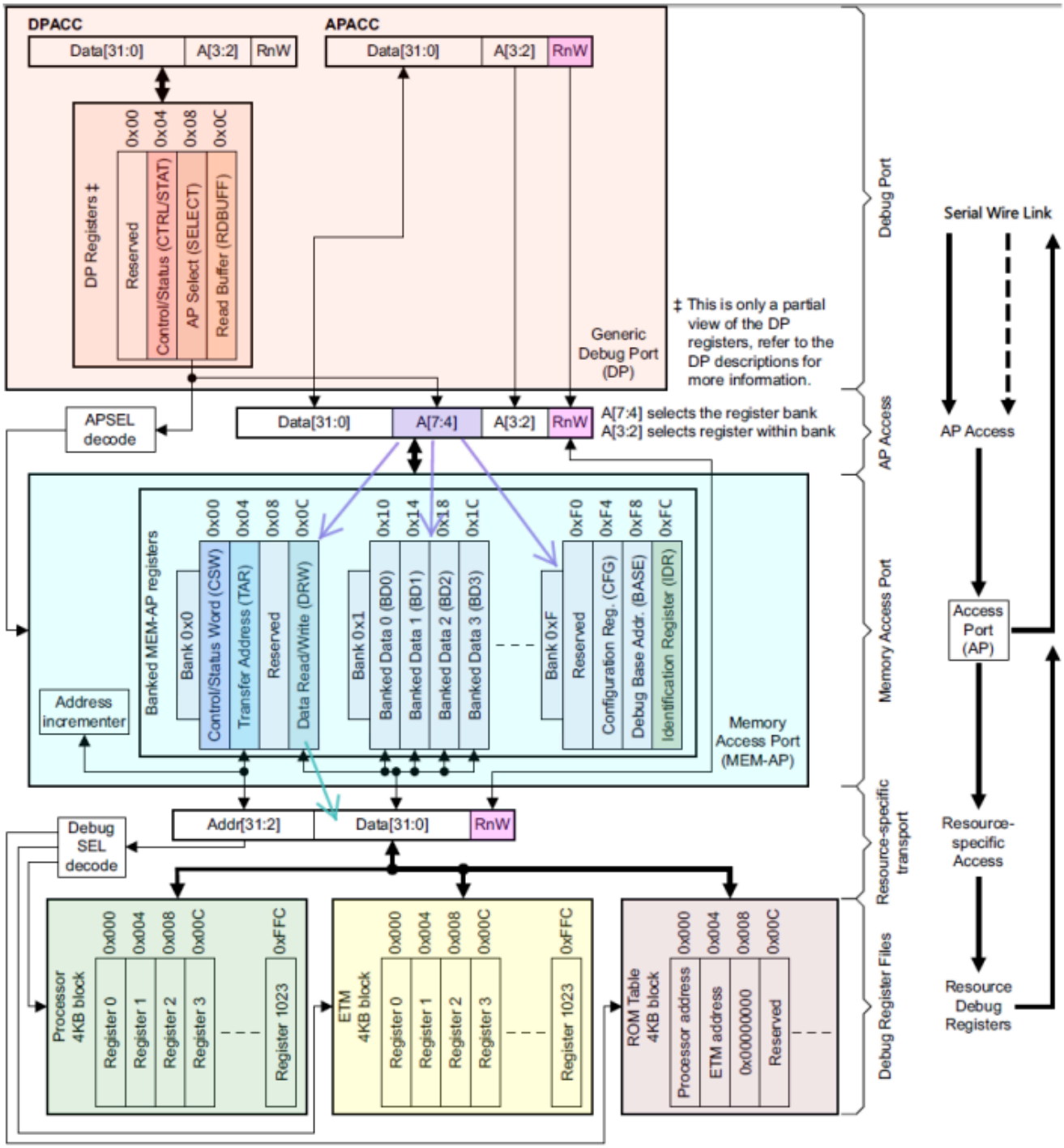
These banks are selected by the APBANKSEL bits of the SELECT register, thus the APBANKSEL bits act as the A[7:4] bits, and the A[1:0] bits are always 0.

Thus each AP can have a total of 16 banks each with 4 registers so a total of 64 registers per AP.

These APs reside in a dedicated memory mapped 4KB address range but in that range ony 64 registers (32bit each) are accessible for a total of 256 Bytes.

This 4KB block of dedicated memory per AP is known as the *Debug Register File*.

Stitching this all together we get something like this:



Coming back to the protocol...

After the initial host-to-target packet request, there is a turnaround cycle after which the control of SWDIO line is released by the HOST and is given over to the TARGET.

The TARGET then returns a 3 bit acknowledgement.

This response can be:

1. 0b100 or OK response, this is given when the DP is ready for the data-phase and no errors have occurred.

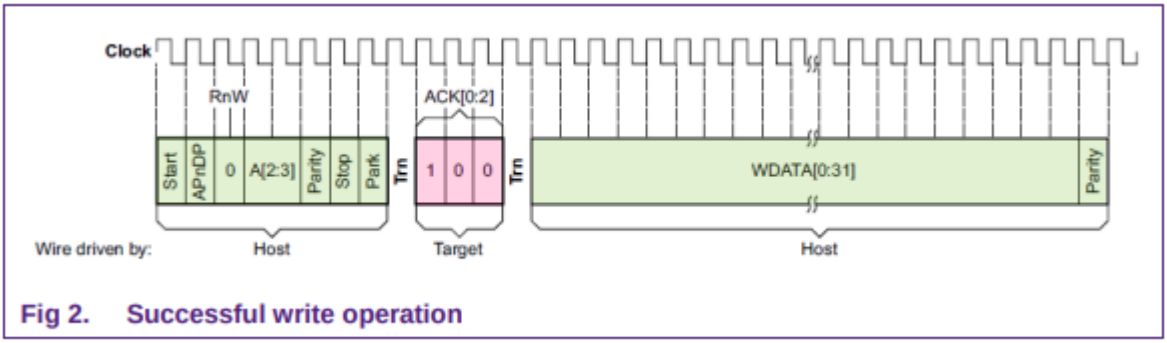


Fig 2. Successful write operation

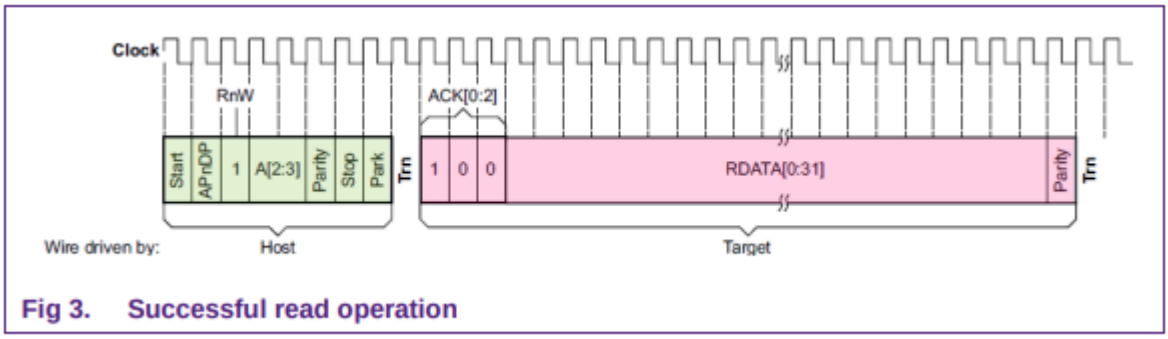


Fig 3. Successful read operation

2. 0b010 or WAIT response, this is given when there is a pending AP transaction going on.

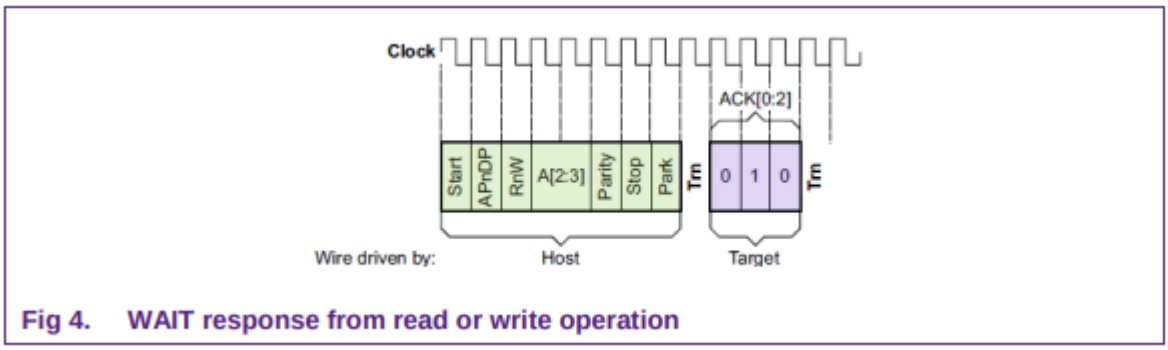


Fig 4. WAIT response from read or write operation

3. 0b001 or FAULT response, this is given when there is a sticky error flag set or when there was a issue in DATA format of previous transaction or a turnaround was missing.

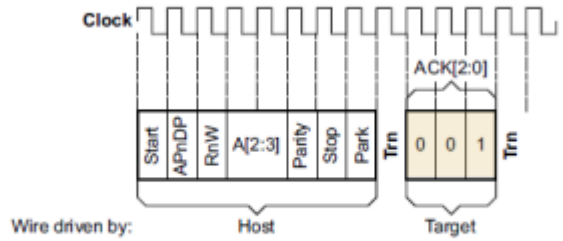


Fig 5. FAULT response from read or write operation

4. 0b111 or PROTOCOL error response, this is given when either there is no TARGET connected (as the line is PULLED UP), or there is an error in the packet headers (incorrect parity).

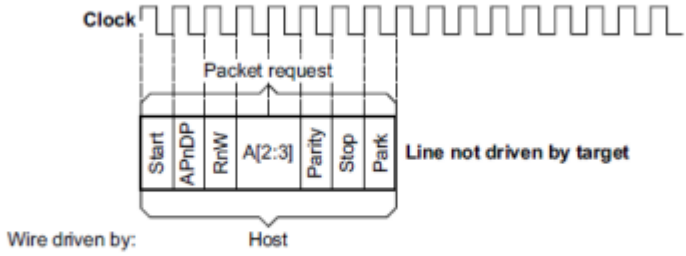


Fig 6. Protocol error sequence

After the ACK by TARGET there is another turnaround period if the request was to write data, the HOST then writes a 32 bit data packet + 1 parity bit for data.

If the initial packet request was for a read then there is no turnaround after ACK and the TARGET returns a 32-bit data packet + 1 parity bit for data which is then followed by a turnaround giving the control of the bus back to the HOST.

The MEM-AP

As ARM is a memory mapped architecture, once someone gets access to the system bus/dcode bus/AHBS, they more or less get access to the whole system including the core registers, code space, peripherals, any internal SRAM/TCM, any external memory and the system space(PPB bus).

This access to TARGETs internal busses for the debugger is provided by a speci called MEM-AP.

Depending on which arm cortex core you are connecting to, the MEM-AP can be essentially a AHB-AP/AHBD-AP/AXI-AP which connects to the systems bus-matrix's salve port as a master giving us access to the rest of the system and the core (even TCM access via AHB-S).

Once we can access the TARGETs internal address space via MEM-AP, we can:

1. Access the code space to view the contents of the internal FLASH.
2. Access the SRAM/TCM.
3. Access processors core registers via System Space.
4. Access any external memory that is memory mapped into the address space.
5. Access Core Debug present in the System Space.
6. Access any peripherals.
7. Do almost anything that is possible via memory mapped registers.

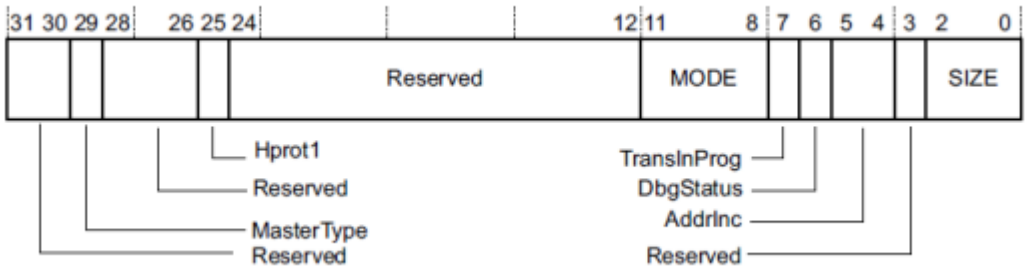
On many simple TARGETs, the AP at address 0x0 is the MEM-AP, although this is highly IMPLEMENTATION DEPENDENT.

Check the TRM of the TARGET to identify how the vendor has implemented the debug subsystem.

In the MEM-AP, the registers at bank 0 are the ones that are the most interesting.

The Bank-0 of MEM-AP contains the following registers:

1. At address 0x00 we have the Control and Status Word Register (CSW).



Bits	Access & Function	Description
[29]	R/W, MasterType	0 = core, 1 = debug, reset to 1. If cleared, this bit prevents the debugger from being able to halt the core.
[25]	R/W, Hprot1	0 = User, 1 = Privilege control, reset to 1. User mode has restricts on access to some special addresses.
[11:8]	R/W, Mode	Mode of operation: b0000 = normal, others are reserved, reset to b0000.
[7]	RO, TransInProg	Transfer in progress, this bit indicates if a transfer is in progress on the AHB master port..
[6]	RO, DbgStatus	Indicates the status of the DAPEN port. If cleared then no AHB transfers carried out. 1 = AHB transfers permitted. 0 = AHB transfers not permitted.
[5:4]	R/W, AddrInc	Auto address increment on Read or Write data access. Only increments if the current transaction completes with no error. Increments and wraps within a 4-KB address boundary, for example for word incrementing from 0x1000 to 0x1FFC. If the start is at 0x14A0, then the counter increments to 0x1FFC, wraps to 0x1000, then continues incrementing to 0x149C. 0b00 = auto increment off. 0b01 = increment single. 0b10 = increment packed (Not used). 0b11 = reserved. No transfer. Size of address increment is defined by the Size field [2:0].
[2:0]	R/W or RO, Size	Size of access field: b000 = 8 bits b001 = 16bits b010 = 32bits b011-111 are reserved. Resets to b000. If this field is RO, then the AP always perform 32 bit accesses.

The bits of interest to us are 29,25,[5:4] and [2:0].

- Bits 29 and 25 must be set to grant the debugger privileged master access over the AHB-AP.
- AddrInc [5:4] bits must be set to 0b01, this enables auto increment support so that memory address in TAR can auto-increment for serial writes in DRW.
- Size [2:0] bits must be set to 0b010 to perform 32-bit access over AHB-AP.

2. At the address 0x04 we have the Transfer Address Register (TAR).

This register holds the address in memory that we want to read from or write to.

3. At the address 0xC we have the Data Read Write Register (DRW).

This register holds the value that we need to write at the address denoted by the TAR, or it contains the value at the address denoted by the TAR.

To read any memory address, load the address into TAR and then perform an AP read from DRW*.

To write to any memory address, load the address into TAR and perform an AP write to DRW.

**For an AP read operation, you might need to get the value from RDBUFF in DP if the ACK is returned as a WAIT response.*

The Magic Switch

Before we proceed further, we should clear something up.

Except for TARGETs that have very low pin count, almost all provide both JTAG and SWD support for maximum flexibility.

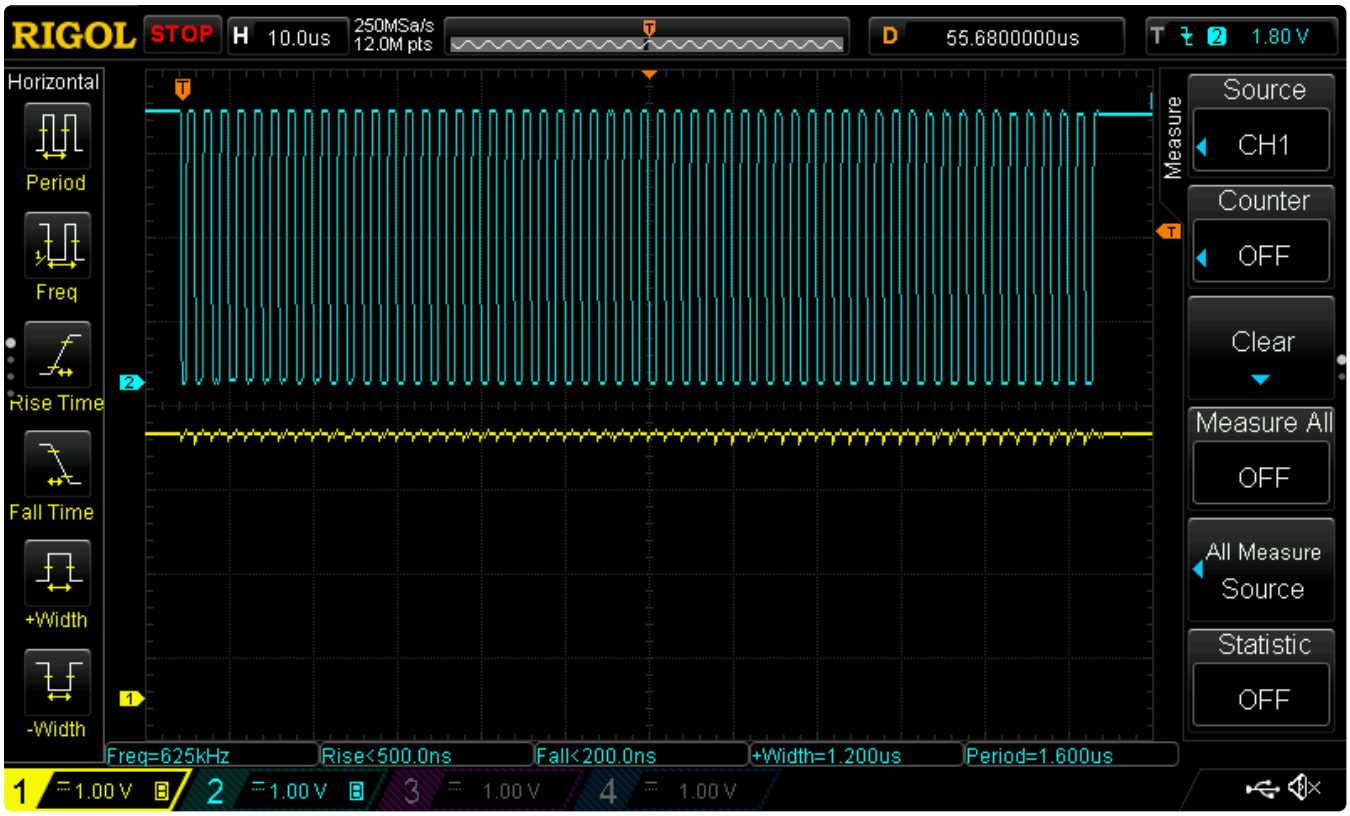
This is possible because pins for SWD and JTAG are muxed togather, the TCK pin of JTAG also acts as the SWDCLK pin, similarly the TMS pin of JTAG also acts as the SWDIO pin.

This combination of JTAG and SWD is called a SWJ-DP (Serial Wire JTAG - Debug Port).

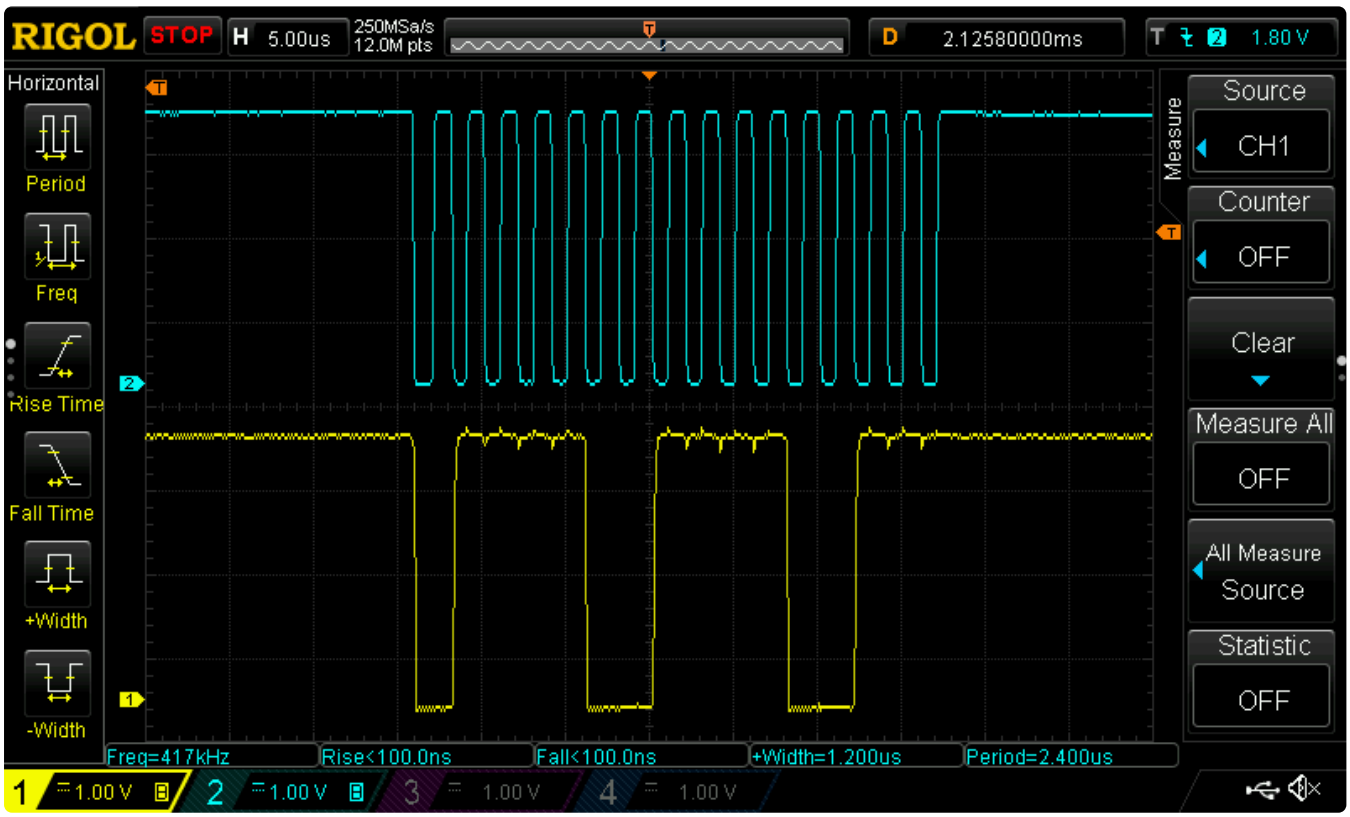
By default in most TARGETs the Debug Port is configured for JTAG, because we are writing a debugger/programmer for SWD, we somehow need to switch the DP to use SWD.

To do this we perform a special switching sequence.

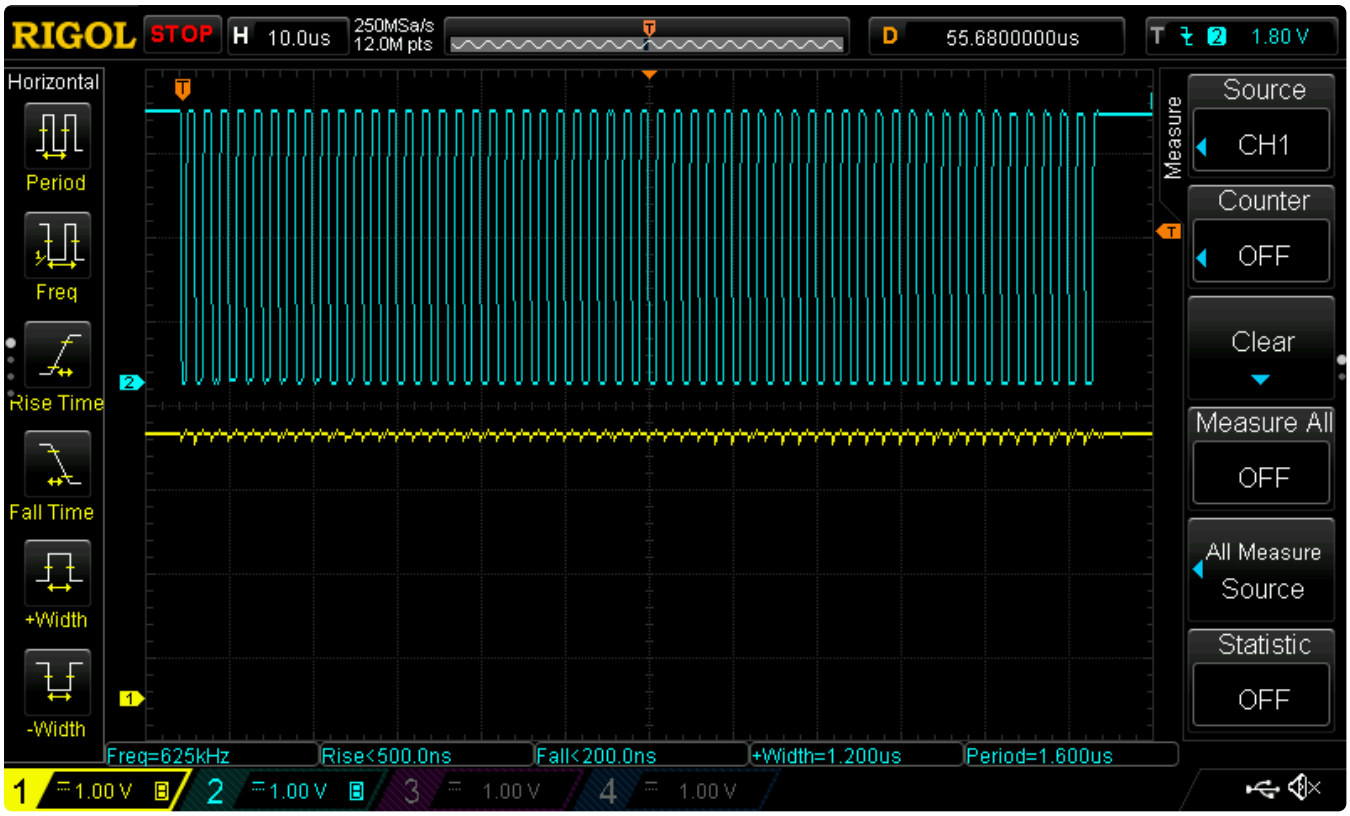
1. First we need to reset the currently selected DP (default JTAG-DP), this is don sending 50 or more clocks while keeping SWDIO HIGH.



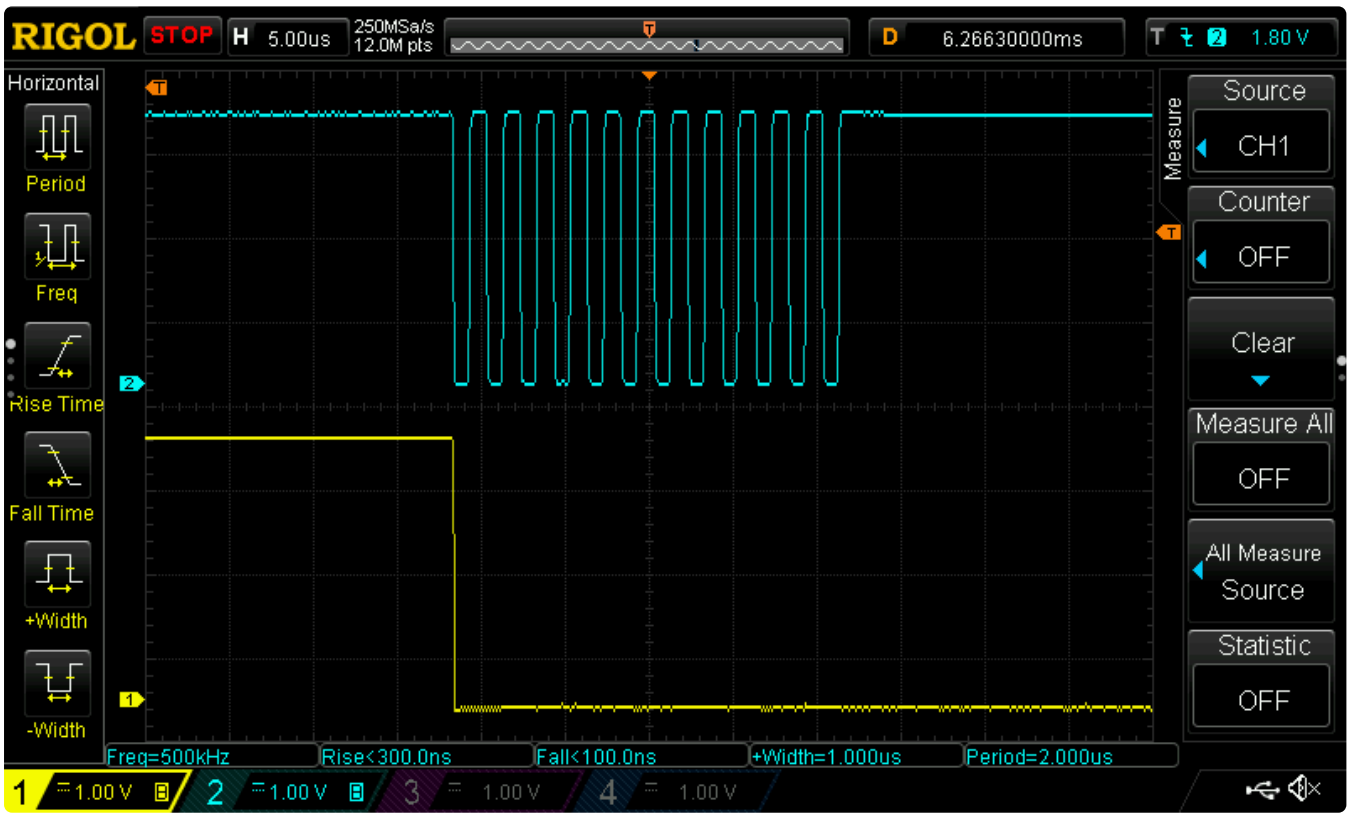
2. Then we send a special 16-bit JTAG-to-SWD select sequence (0xE79E LSB first).



3. Then we again reset the DP (now SW-DP) by sending 50 or more clocks while keeping SWDIO HIGH.



4. Then we send 12 clocks with SWDIO LOW.



Once we have done this, the SW-DP is ready to use.

Note: According to the Coresight Components TRM, this SWJ-DP Switching Sequ is Deprecated and a new method using JTAGNSW pin is used, but as per real-world

experimentation, this sequence still holds true for majority of the MCUs in the market, refer to the TRM for further info.

The part where the FUN starts....

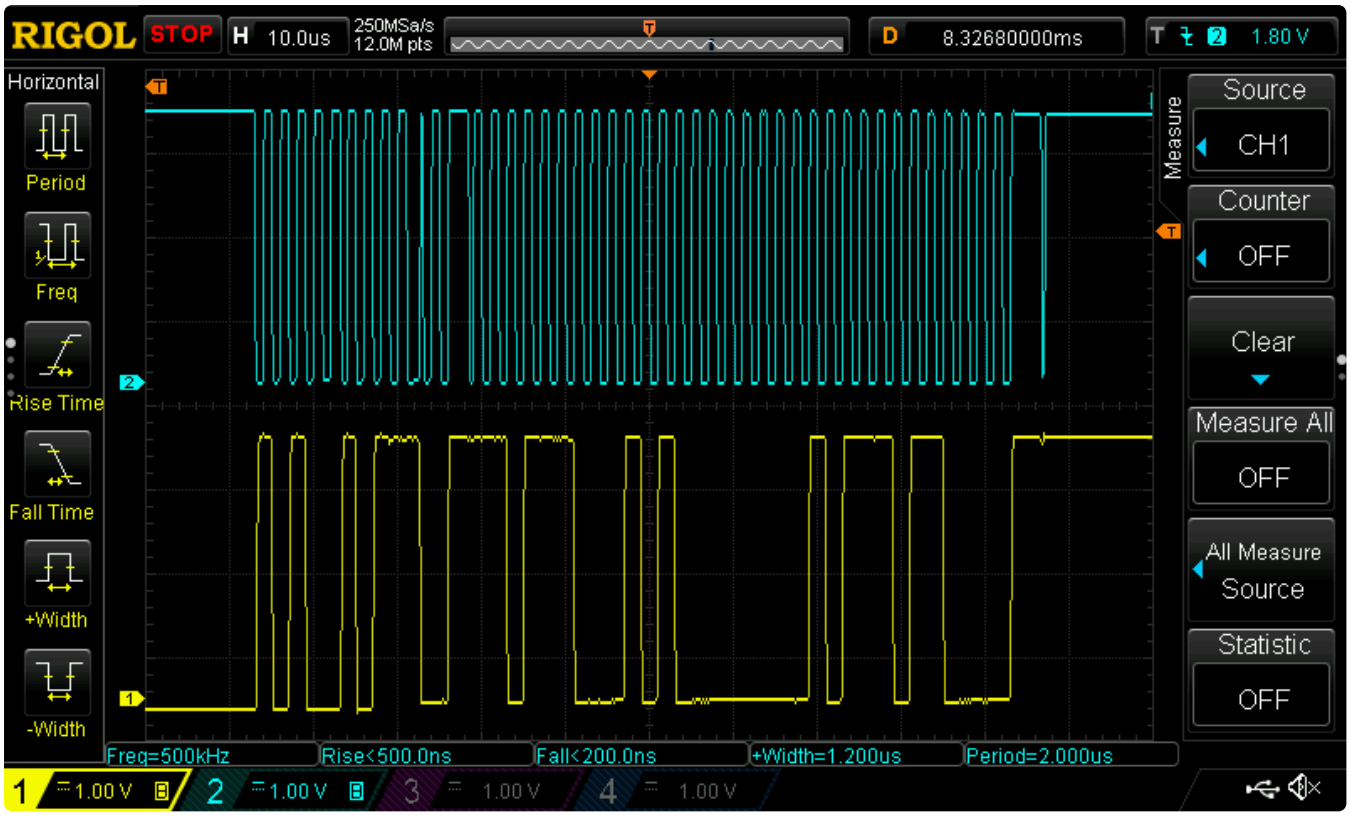
So now the fun stuff starts as after getting the protocol stuff done, we can actually connect a TARGET to our pico and explore the DAP.

-Important thing to note here that the security settings for the TARGET are set to default enabling full unrestricted debug access.-

The first thing to do after doing the switching and reset sequence it to read the IDCODE register.

If done right, this will look something like this:

The blue channel denotes the clock and yellow the data, the initial 4 segments denote the JTAG-to-SWD switching sequence, the last segment is IDCODE read from DP.



Zoomed IDCODE packet request and the subsequent TARGET response.

The STM32F103C8T6 (STM32 blue pill) returns us a IDCODE value of 0x1ba01477.

Then we need to write to the CTRL/STAT register of the DP and enable the CSYSPWRUPREQ and CDBGPWRUPREQ bits to bring the rest of the system online.

Then we need to set the SELECT register such that APSEL is 0x0 and APBANKSEL is also 0x0, so SELECT is 0x00000000 to access the MEM-AP (for STM32F10XX).

Then we need to set CSW register of MEM-AP to the value of 0x22000012, enabling privileged master access over AHB-AP, auto-increment for TAR and 32-bit IO over DRW.

Now we can access the full memory address space exposed by our TARGET by simply selecting the address via TAR and reading/writing via DRW.

Lets start by dumping the FLASH.

The code space in ARMv7-M starts at 0x00000000, the FLASH is also aliased with this address so the reads from this will dump the FLASH.

```
**** SWD PHY Initialised! *****
```

```
DAP: Setup DP and MEM-AP.
IDCODE: 0x1ba01477 ACK: 1
CTRLSTAT_W: 0x50000000 ACK: 1
CTRLSTAT_R: 0xf0000000 ACK: 1
CSW_W: 0x22000012 ACK: 1

20002800
080028b5
08000c5d
08000c63
08000c69
08000c6f
08000c75
00000000
00000000
00000000
00000000
08000c7b
08000c87
00000000
08000c93
.
.
.
.
.
```

Cool! So if there is no security enabled, we can dump the flash from the entire chip via our debugger.

The Core Debug

Dumping flash is fun and all but we are building a debugger and a programmer here, so how does that work?

Lets begin with how debugging works in ARM CoreSight debug architecture.

-This segment is meant for ARMv7-M, although parts of it might be valid for ARMv7-A/R and ARMv6-M.-

But first, what does it actually mean to debug something?

A bug is a flaw in code (or a feature for some), to debug means to remove the flaw.

And to do so we need to understand why the flaw was there in first place.

This process of understanding begins by analyzing how the code works one step at a time.

Step by step we inspect what each instruction that our CPU core executes does, we also watch how the this changes the data that our code deals with.

Eventually we will reach the point of failure, we can then rectify the issue by fixing the broken code.

Most of the times, this approach works, but sometimes, we need to have an understanding of the whole system to intuitively guess where the issue might be.

According to our little story here:

1. We need a system that can halt a CPU.
2. The system should be able to execute a single instruction and then halt again, so that we can view the changes that instruction made. (called single stepping)
3. The system should let us see that data the CPU is working on. (core registers and SRAM)
4. The system should let us give it a address so that when the control flow reaches that address, the CPU halts. (called a breakpoint)

For Cortex-M, all of this (except for breakpoints) is done via a set of special registers memory mapped in the System Space in a region referred to as Debug Control Block.

This functionality is called **Core Debug**.

In the grand scale of things, debugging can be divided into the following:

1. **Invasive Debugging**:
   This is the type of debugging which enables you to control and observe the processor and modify its state.
   On ARMv7-M this can be divided into two more catagories:
   - Halting Debug-mode

This is the typical debugging mode in which an external debugger is connected to the system which controls the debug flow, enabling it to halt the core and observe core registers and internal memory.

- Monitor Debug-mode
  This is the type of debugging which enables a core to debug itself without any external debugger, this is possible due to a special DebugMon exception implemented in ARMv7-M.
  In this type of debugging, the core isn't halted on execution of a debug event, Instead a special exception is triggered (DebugMon) which can then be implemented in such a way that it observes the systems state to find the fault, this enables debugging in settings where halting the core can alter the natural system behavior.
  Although some performance impact is still observed due to the control flow switching to DebugMon Exception Handler.

2. **Non-Invasive Debugging**:
   This is the type of debugging which is done only by observing the processor.
   This is most useful when you are debugging and performance profiling a real-time system and halting the core is not an option because it will alter the natural system behavior.
   This is done by tracing and recording each and every instruction that the processor executes and the data it executed on, this data is collected in real-time without affecting the performance of the processor in any way.
   The data is then analyzed later on to find the fault.
   In ARMs CoreSight Debug Architecture, this functionality is provided by the ETM (Embedded Trace Macrocell) and ITM (Instrumentation Trace Macrocell).
   These are not a part of the standard core and must be included separately in the system by your SoC vendor.

In this article we will only be focusing on Halting Debug mode as it is the traditional way most debuggers work.

The **Core Debug** (which provides Invasive-Debug functionality) contains the following 4 registers:

Address	Name	Access	Function
0xE000EDF0	DHCSR	RW	Debug Halting Control and Status
0xE000EDF4	DCRSR	WO	Debug Core Register Selector
0xE000EDF8	DCRDR	RW	Debug Core Register Data
0xE000EDFC	DEMCR	RW	Debug Exception and Monitor Control

Lets have a look at what each of these do.

Debug Halting Control and Status Register

This is the main register that allows us to enable debugging support on a system.

It also contains key functionality to halt the core, single step it, mask special interrupts and report the status of the core.

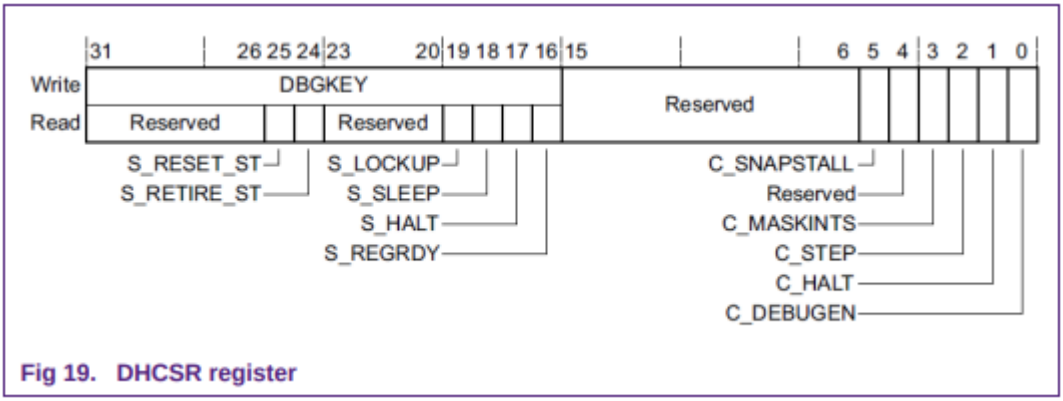


Fig 19. DHCSR register

Bits	Access & Function	Description
[31:16]	WO, DBGKEY	The Debug Key: Software must write 0xA05F to this field to enable write accesses to bits [15:0], otherwise the processor ignores the write access.
[31:26]	-	Reserved
[25]	RO, S_RESET_ST	Indicates whether the processor has been reset since the last read of DHCSR: 0 = No reset since last DHCSR read. 1 = At least one reset since last DHCSR read. This is a sticky bit, that clears to 0 on a read of DHCSR.
[24]	RO, S_RETIRE_ST	Indicates whether the processor has completed the execution of an instruction since the last read of DHCSR: 0 = No instruction retired since last DHCSR read. 1 = At least one instruction retired since last DHCSR read. This is a sticky bit that clears to 0 on a read of DHCSR. A debugger can check this bit to determine if the processor is stalled on a load, store or fetch access. This bit is UNKNOWN after a Power-on or Local reset, but then is set to 1 as soon as the processor executes and retires an instruction.
[23:20]	-	Reserved
[19]	RO, S_LOCKUP	Indicates whether the processor is locked up because of an unrecoverable exception: 0 = Not locked up 1 = Locked up

		This bit can only be read as 1 by a remote debugger, using the DAP. The value of 1 indicates that the processor is running but locked up. The bit clears to 0 when the processor enters Debug state.
[18]	RO, S_SLEEP	ndicates whether the processor is sleeping: 0 = Not sleeping. 1 = Sleeping. The debugger must set the C_HALT bit to 1 to gain control, or wait for an interrupt or other event to wake up the system.
17	RO, S_HALT	Indicates whether the processor is in Debug state 0 = Not in debug state 1 = In Debug state
16	RO, S_REGRDY	A handshake flag for transfers through the DCRDR: • Writing to DCRSR clears the bit to 0. • Completion of the DCRDR transfer then sets the bit to 1. 0 = There has been a write to the DCRDR, but the transfer is not complete 1 = The transfer to or from the DCRDR is complete. This bit is valid only when the processor is in Debug state, otherwise the bit is UNKNOWN.
[15:6]	-	Reserved
[5]	RW, C_SNAPSTALL	If the processor is stalled on a load or store operation, a debugger can set this bit to 1 to attempt to break the stall. This will result in the memory status to being undefined, and thus should only be used in error recovery scenarios. Omitted, set to 0
[4]	-	Reserved
[3]	RW, C_MASKINTS	When debug is enabled, the debugger can write to this bit to mask PendSV, SysTick and external configurable interrupts: 0 = Do not mask. 1 = Mask PendSV, SysTick and external configurable interrupts. The bit does not affect NMI.
[2]	RW, C_STEP	Omitted, set to 0.
[1]	RW, C_HALT	Processor halt bit, write this bit to 0 = No effect 1 = Halt the processor. If the processor is already halted, writing this bit to 1 has no side effect, the processor remains halted (in debug state).
[0]	RW, C_DEBUGEN	Halting debug enable bit: 0 = Disabled. 1 = Enabled. This bit can only be set to 1 by the debugger (via DAP), it cannot be set to 1 under software control. If a write to DHCSR changes this bit from 0 to 1, it must also write 0 to C_MASKINTS bit.

The key things to note from the above figures are:

- Any Write to this register must be made with the upper half of the 32-bit word containing the DBGKEY 0xA05F.
  This prevents accidental writes to this registers memory address from breaking the system.

- The Halting-Debug mode can only be enabled by an external debugger via DAP (MEM-AP write).
  This prevents any software running on the core to lock the system in a halted state.

Debug Core Register Data Register

This register is used to hold the value that needs to be written to an internal core register.

It also contains the value of an internal core register when a read request is made.

It can also be used as a temporary data cache to pass information between debugger and the firmware running on the core itself.

Debug Core Register Selector Register

This registers job is to select an internal core register to which we need to perform a read/write operation on.

It's also a write-only register.

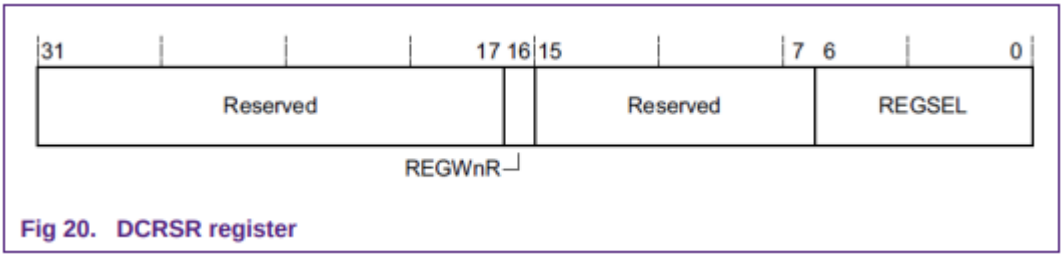


Fig 20. DCRSR register

Bits	Access & Function	Description
[16]	RW, REGWnR	Specifies the access direction of the transfer 0 = read 1 = write
[6:0]	RW, REGSEL	Specifies the ARM core register, special-purpose register, or Floating-point extension register, to transfer 0x00, 0x01, 0x02,… 0x0C = R0, R1, R2,…R12 0x0D = The current SP (MSP or PSP) 0x0E = LR 0x0F = DebugReturnAddress, or "PC": the address of the first instruction to be executed on exit from Debug state. 0x10 = xPSR 0x11 = MSP 0x12 = PSP 0x14 = 4in1 SFR: [31:24] = CONTROL [23:16] = FAULTMASK [15:08] = BASEPRI [07:00] = PRIMASK Values are packed with leading zeros. 0x21 = Floating-point Status and Control Register, FPCSR 0x40, 0x41, 0x42,…0x5F = S0, S1, S2,…S31 (32 FPU registers)

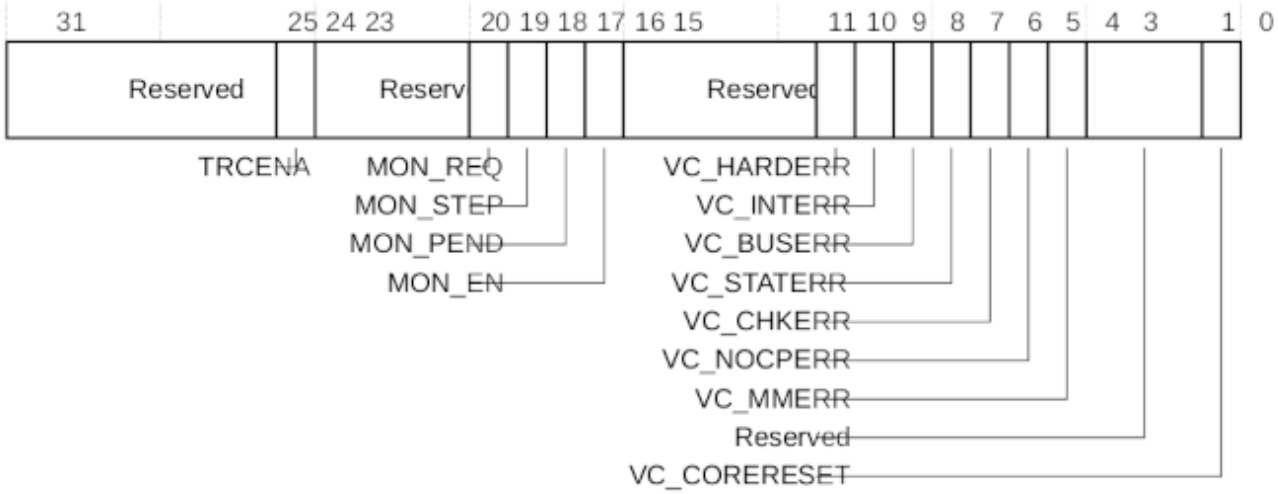
To write to an internal core register, we first put the value that we need to write into DCRDR, and then write to this register with the REGWnR bit set and the correct core register selected.

To read from an internal core register, we write to DCRSR with the REGWnR bit set to 0 and the correct core register selected, then the DCRDR can be read to get the value of the internal core register.

Debug Exception and Monitor Control Register

This register is primarily used for Monitor-Debug functionality and Vector Catch functionality that controls behavior of debug system on an exception generated due to faults.

It also has TRACENA bit which is used to enable ITM and DWT features if present.



The most important bit for us however, is the VC_CORERESET bit.

This bit allows us to catch the Reset Vector, i.e it will halt the Cortex-M core and enter the system into debug state when the system is reset by a Local Reset with the PC[R15] pointing to the Reset_Handler.

We need this because we ideally want the system to be in a known state before we start to debug it.

Setting this bit and then performing a Local Reset(via NVIC.AIRCR) will reset the core and hopefully rest of the system. (How much of the system is reset is IMPLEMENTATION DEFINED.)

Thus now we know how to:

1. Halt the core. (Set C_HALT + C_DEBUGEN in DHCSR)
2. Single Step the halted core. (Set C_STEP + C_DEBUGEN in DHCSR)
3. Read and Write to core registers. (Use DCRDC and DCRSR)
4. Halt the core after Reset. (VC_CORERESET in DEMCR)

To resume the core, just write 0 to both C_HALT and C_DEBUGEN.

Let's test this, to do that, lets make a simple program that we can debug.

Why not blink an LED? Its the Hello World equivalent in the embedded doamin

We will use assembly to write this as it is simple enough.

Here we go, The reader must be familiar with Thumb-2 ISA to understand this, although I have kept it as simple as possible with added comments.

```
.syntax unified

.global blink_main

.thumb

.align 2

.equ FLASH_ACR,0x40022000
.equ FLASH_PRFTBE,0x10

.equ SYSTICK_CTRL,0xE000E010
.equ SYSTICK_LOAD,0xE000E014
.equ SYSTICK_VAL,0xE000E018
.equ SYSTICK_DELAY,0x160000

.equ RCC_AB2ENR,0x40021018
.equ RCC_AB2ENR_IOPCEN,0x10

.equ GPIOC_CRH,0x40011004
.equ GPIOC_BSSR,0x40011010
.equ GPIOC_PIN13_SET,0x00002000
.equ GPIOC_PIN13_RESET,0x20000000


.section .text
.word 0x20005000
.word blink_main + 1
.word NMI_Handler
.word HardFault_Handler
.fill 42,1,0

NMI_Handler:
        b NMI_Handler

HardFault_Handler:
        b HardFault_Handler

.align 2
blink_main:
        cpsid i

        #enable flash prefetch buffer
        ldr r0,=FLASH_ACR
        ldr r1,=FLASH_PRFTBE
```

```
        str r1,[r0]

        #enable GPIOC clock
        ldr r0,=RCC_AB2ENR
        ldr r1,=RCC_AB2ENR_IOPCEN
        ldr r2,[r0]
        orr r2,r1
        str r2,[r0]

        #setup PC13 as push-pull output
        ldr r0,=GPIOC_CRH
        ldr r1,=0x200000
        str r1,[r0]

        #set value for PIN13
        ldr r0,=GPIOC_BSSR
        ldr r1,=GPIOC_PIN13_SET
        str r1,[r0]


        #setup SysTick for delay timing
        #set reload value of systick
        ldr r0,=SYSTICK_LOAD
        ldr r1,=SYSTICK_DELAY
        str r1,[r0]
        #use cpu clock as systick clock source,do not generate systick interrupts and
        ldr r0,=SYSTICK_CTRL
        mov r1,0x5
        str r1,[r0]


        #the main blink loop
        ldr r0,=GPIOC_BSSR
        ldr r1,=GPIOC_PIN13_RESET
        ldr r2,=GPIOC_PIN13_SET
blink_loop:
        str r1,[r0]
        bl delay
        str r2,[r0]
        bl delay
        b blink_loop

delay:
        push {r0-r2}
        ldr r0,=SYSTICK_VAL
        mov r1,0x1
        str r1,[r0]
        mov r2,5
delay_loop:
```

```
        ldr r1,[r0]
        cmp r1,r2
        bge delay_loop
        pop {r0-r2}
```

Lets assemble it with the GNU ARM embedded toolchain.

```
arm-none-eabi-as blink.s -mcpu=cortex-m3 -c -o blink.o
```

We will use a simple linker script to set the base address for our .text section.

```
ENTRY(blink_main)

SECTIONS
{
    . = 0x20000000;
    .text : ALIGN(4)
    {
        *(.text);
    }
}
```

And finally.. to get our bin.

```
arm-none-eabi-ld -o blink.elf blink.o -T link.ld && arm-none-eabi-objcopy -O binary bl
```

We will use the xxd tool to generate a hex dump as a C array of our bin which we than then include on our debugger code running on Pi Pico.

```
xxd -g4 -i blink.bin > bin.h
```

We can then use the MEM-AP to write this binary directly at the SRAM base.

Here she goes....

```
***** SWD PHY Initialised! *****

DAP: Setup DP and MEM-AP.
IDCODE: 0x1ba01477 ACK: 1
CTRLSTAT_W: 0x50000000 ACK: 1
CTRLSTAT_R: 0xf0000000 ACK: 1
CSW_W: 0x22000012 ACK: 1
```

```
CPUID_R: 0x411fc231 ACK: 1

Core Debug: Read DHCSR
DHCSR ADDR: 0xe000edf0 ACK: 1
DHCSR VALUE: 0x03090000 ACK: 1

Core Debug: Enable Debug Mode.
DHCSR ADDR: 0xe000edf0 ACK: 1
DHCSR VALUE: 0xa05f0003 ACK: 1

Core Debug: Read DHCSR
DHCSR ADDR: 0xe000edf0 ACK: 1
DHCSR VALUE: 0x00030003 ACK: 1

Core Debug: Enable Halt on Reset
DEMCR ADDR: 0xe000edfc ACK: 1
DEMCR VALUE: 0x00000001 ACK: 1

NVIC.AIRCR: Issue SYSRESET.
AIRCR ADDR: 0xe000ed0c ACK: 1
AIRCR VALUE: 0xfa050004 ACK: 1

Core Debug: Read DHCSR
DHCSR ADDR: 0xe000edf0 ACK: 1
DHCSR VALUE: 0x00030003 ACK: 1

Core Debug: Setup PC
DCRDR ADDR: 0xe000edf8 ACK: 1
DCRDR VALUE: 0x20000119 ACK: 1
DCRSR ADDR: 0xe000edf4 ACK: 1
DCRSR VALUE: 0x0001000f ACK: 1

Core Debug: Setup MSP
DCRDR ADDR: 0xe000edf8 ACK: 1
DCRDR VALUE: 0x20004000 ACK: 1
DCRSR ADDR: 0xe000edf4 ACK: 1
DCRSR VALUE: 0x0001000d ACK: 1

VTOR: Relocate VTOR to SRAM
VTOR ADDR: 0xe000ed08 ACK: 1
VTOR VALUE: 0x20000000 ACK: 1

Writing bin to RAM
CODE : 0x0 0x20005000
CODE : 0x4 0x20000118
CODE : 0x8 0x20000110
CODE : 0xc 0x20000112
CODE : 0x10 0x20000116
CODE : 0x14 0x20000116
```

```
CODE : 0x18 0x20000116
CODE : 0x1c 0x00000000
CODE : 0x20 0x00000000
CODE : 0x24 0x00000000
CODE : 0x28 0x00000000
CODE : 0x2c 0x20000116
CODE : 0x30 0x20000116
CODE : 0x34 0x00000000
CODE : 0x38 0x20000116
CODE : 0x3c 0x20000114
CODE : 0x40 0x00000000
CODE : 0x44 0x00000000
CODE : 0x48 0x00000000
CODE : 0x4c 0x00000000
CODE : 0x50 0x00000000
CODE : 0x54 0x00000000
CODE : 0x58 0x00000000
CODE : 0x5c 0x00000000
CODE : 0x60 0x00000000
CODE : 0x64 0x00000000
CODE : 0x68 0x00000000
CODE : 0x6c 0x00000000
CODE : 0x70 0x00000000
CODE : 0x74 0x00000000
CODE : 0x78 0x00000000
CODE : 0x7c 0x00000000
CODE : 0x80 0x00000000
CODE : 0x84 0x00000000
CODE : 0x88 0x00000000
CODE : 0x8c 0x00000000
CODE : 0x90 0x00000000
CODE : 0x94 0x00000000
CODE : 0x98 0x00000000
CODE : 0x9c 0x00000000
CODE : 0xa0 0x00000000
CODE : 0xa4 0x00000000
CODE : 0xa8 0x00000000
CODE : 0xac 0x00000000
CODE : 0xb0 0x00000000
CODE : 0xb4 0x00000000
CODE : 0xb8 0x00000000
CODE : 0xbc 0x00000000
CODE : 0xc0 0x00000000
CODE : 0xc4 0x00000000
CODE : 0xc8 0x00000000
CODE : 0xcc 0x00000000
CODE : 0xd0 0x00000000
CODE : 0xd4 0x00000000
CODE : 0xd8 0x00000000
```

```
CODE : 0xdc 0x00000000
CODE : 0xe0 0x00000000
CODE : 0xe4 0x00000000
CODE : 0xe8 0x00000000
CODE : 0xec 0x00000000
CODE : 0xf0 0x00000000
CODE : 0xf4 0x00000000
CODE : 0xf8 0x00000000
CODE : 0xfc 0x00000000
CODE : 0x100 0x00000000
CODE : 0x104 0x00000000
CODE : 0x108 0x00000000
CODE : 0x10c 0x00000000
CODE : 0x110 0xe7fee7fe
CODE : 0x114 0x4770e7fe
CODE : 0x118 0x4817b672
CODE : 0x11c 0x0110f04f
CODE : 0x120 0xea426802
CODE : 0x124 0x60020201
CODE : 0x128 0xf44f4814
CODE : 0x12c 0x60011100
CODE : 0x130 0xf44f4813
CODE : 0x134 0x60015100
CODE : 0x138 0xf06f4812
CODE : 0x13c 0x6001417f
CODE : 0x140 0xf04f4811
CODE : 0x144 0x60010105
CODE : 0x148 0xf04f480d
CODE : 0x14c 0xf44f5100
CODE : 0x150 0x60015200
CODE : 0x154 0xf804f000
CODE : 0x158 0xf0006002
CODE : 0x15c 0xe7f8f801
CODE : 0x160 0x480ab407
CODE : 0x164 0x31b0f44f
CODE : 0x168 0xf04f6001
CODE : 0x16c 0x6801020a
CODE : 0x170 0xdafc4291
CODE : 0x174 0x4770bc07
CODE : 0x178 0x40021018
CODE : 0x17c 0x40011004
CODE : 0x180 0x40011010
CODE : 0x184 0xe000e014
CODE : 0x188 0xe000e010
CODE : 0x18c 0xe000e018

Reading bin from RAM

Verification Successful!
```

```
Core Debug: Read DHCSR
DHCSR VALUE: 0x00030003 ACK: 1

Core Debug: Disable Debug Mode and run core.
DHCSR ADDR: 0xe000edf0 ACK: 1
DHCSR VALUE: 0xa05f0000 ACK: 1

Core Debug: Read DHCSR
DHCSR VALUE: 0x01010000 ACK: 1
```

If you have done everything right then you will be greeted with a blinking onboard-LED on the STM32 blue pill. :)

We can now also test our debugger to single step the code and read the value of PC.

Debugger Single Step Output.	**Disassembly of the SRAM code.**

```
PC : 0x20000118     20000118 <main>:
PC : 0x2000011a     20000118:       b672            cpsid   i
PC : 0x2000011c     2000011a:       4817            ldr     r0, [pc, #92]   ; (200001
PC : 0x20000120     2000011c:       f04f 0110       mov.w   r1, #16
PC : 0x20000122     20000120:       6802            ldr     r2, [r0, #0]
PC : 0x20000126     20000122:       ea42 0201       orr.w   r2, r2, r1
PC : 0x20000128     20000126:       6002            str     r2, [r0, #0]
PC : 0x2000012a     20000128:       4814            ldr     r0, [pc, #80]   ; (200001
PC : 0x2000012e     2000012a:       f44f 1100       mov.w   r1, #2097152    ; 0x20000
PC : 0x20000130     2000012e:       6001            str     r1, [r0, #0]
PC : 0x20000132     20000130:       4813            ldr     r0, [pc, #76]   ; (200001
PC : 0x20000136     20000132:       f44f 5100       mov.w   r1, #8192       ; 0x2000
PC : 0x20000138     20000136:       6001            str     r1, [r0, #0]
PC : 0x2000013a     20000138:       4812            ldr     r0, [pc, #72]   ; (200001
PC : 0x2000013e     2000013a:       f06f 417f       mvn.w   r1, #4278190080 ; 0xff000
PC : 0x20000140     2000013e:       6001            str     r1, [r0, #0]
PC : 0x20000142     20000140:       4811            ldr     r0, [pc, #68]   ; (200001
PC : 0x20000146     20000142:       f04f 0105       mov.w   r1, #5
PC : 0x20000148     20000146:       6001            str     r1, [r0, #0]
PC : 0x2000014a     20000148:       480d            ldr     r0, [pc, #52]   ; (200001
PC : 0x2000014e     2000014a:       f04f 5100       mov.w   r1, #536870912  ; 0x20000
PC : 0x20000152     2000014e:       f44f 5200       mov.w   r2, #8192       ; 0x2000
PC : 0x20000154
PC : 0x20000160     20000152 <blink_loop>:
PC : 0x20000162     20000152:       6001            str     r1, [r0, #0]
PC : 0x20000164     20000154:       f000 f804       bl      20000160 <delay>
```

```
PC : 0x20000168     20000158:       6002            str     r2, [r0, #0]
PC : 0x2000016a     2000015a:       f000 f801       bl      20000160 <delay>
PC : 0x2000016e     2000015e:       e7f8            b.n     20000152 <blink_loop>
PC : 0x20000170
PC : 0x20000172     20000160 <delay>:
PC : 0x2000016e     20000160:       b407            push    {r0, r1, r2}
PC : 0x20000170     20000162:       480a            ldr     r0, [pc, #40]   ; (200001
PC : 0x20000172     20000164:       f44f 31b0       mov.w   r1, #90112      ; 0x16000
PC : 0x2000016e     20000168:       6001            str     r1, [r0, #0]
PC : 0x20000170     2000016a:       f04f 020a       mov.w   r2, #10
PC : 0x20000172
PC : 0x2000016e     2000016e <delay_loop>:
PC : 0x20000170     2000016e:       6801            ldr     r1, [r0, #0]
PC : 0x20000172     20000170:       4291            cmp     r1, r2
PC : 0x2000016e     20000172:       dafc            bge.n   2000016e <delay_loop>
PC : 0x20000170     20000174:       bc07            pop     {r0, r1, r2}
PC : 0x20000172     20000176:       4770            bx      lr
.                   20000178:       40021018        andmi   r1, r2, r8, lsl r0
.                   2000017c:       40011004        andmi   r1, r1, r4
.                   20000180:       40011010        andmi   r1, r1, r0, lsl r0
.                   20000184:       e000e014        and     lr, r0, r4, lsl r0
.                   20000188:       e000e010        and     lr, r0, r0, lsl r0
                    2000018c:       e000e018        and     lr, r0, r8, lsl r0
```

As you can see the control flow via single stepping matches the disassembly output.

Key things to take care of when executing from SRAM are:

- Reset_Handler address must have the 1st bit set to enable thumb-mode.
- The VTOR must be relocated to SRAM before resuming the core.
- The MSP must also be set along with the PC.

BreakPoints!

Now, lets wrap the debug section up by briefly talking about breakpoint.

ARMv7-M supports 2 types of breakpoints

- Hard Breakpoints
- Soft Breakpoints.

-Depending on the functionality of FPB unit implemented by the SoC vendor, either all or parts of the following are true.-

Hard Breakpoints are made possible by the FPB (Flash Patching and BreakPoint Unit).

It consists of registers which which act as comparators, You set them up with the value of PC where you want the breakpoint to be and as soon as the control reaches to the PC value, the PC is remapped to the address present in FP_REMAP register of FPB unit **or** to a *bkpt* instruction, which will eventually trigger a DebugMon exception.

There are about 2-8 comparators available in FPB, the actual value is IMPLEMENTATION DEFINED.

Soft Breakpoints are make possible by the *bkpt* instuction of ARMv7-M.

Depeding on how things are set up, as soon as control reaches this instruction, the CPU enters debug state by entering the DebugMon exeption.

Why don’t you just FLASH them?

This is the final part of this article and the only remaining quest of our journey.

We have learned about SWD, How debugging works, how to make, run and debug our own firmware via our own debugger.

The only thing that remains is the question of how does FLASH programming works?

This is where things get messy.

Unlike the other stuff that we have mentioned above, FLASH programming in ARM ecosystem is completely non-standardised.

It is upto the job of the SoC vendor to define the specifics of how their system enables this process to happen.

This is why we see every SoC vendor provide their own tools and debuggers/programmers.

ST has ST-link, NXP has LPC-link/MCU-link, TI, SiLabs, Atmel everyone has their own way.

This is good as it provides flexibility for SoC vendors, Its also bad for us because making a universal programmer is now a very difficult job.

There are companies out there that actualy spend time and money over this and have made a hugely succesful business out of it. (SEGGER)

But still some of the ways this process happens are listed below:

1. Manually Program the FLASH using MEM-AP to access whatever FLASH programming peripheral is implemented in the SoC.
2. Upload a FLASHLOADER and binary to SRAM and use it to access the FLASH programming peripheral and FLASH the binary.
3. Use any helper functions implemented inside the ROM to program the flash.

Out of these, 1st method is the slowest as it is dependent on the speed at which you can transact over SWD.

2nd method is the most used followed by 3rd method in some cases.

We will use the 2nd method.

To do this, We will program a special binary into the SRAM called the FLASHLOADER.

The job of the FLASHLOADER is to run on the target core and program its FLASH by accessing the internal memory mapped peripheral responsible to do so.

This FLASHLOADER can run at the speed that core can run natively thus is way faster that doing it manually via individual SWD MEM-AP transactions.

The program to upload is also put inside the SRAM, broken into smaller blocks depending on the size of the binary to FLASH.

The FLASHLOADER can then flash the part in SRAM, and then signal the debugger using any number of mechanisms.

The debugger can then upload the next block to program and then re-runs the FLASHLOADER, this happens until the whole binary is FLASHED.

On our TARGET (STM32F103) the FLASH programming is taken care of by the FPEC. (Flash Programming and Erase Controller)

The FLASHLOADER that I have used is shown below.

It uses R0,R1,R2 and R3 to pass the start address of flash,sram, size of block, to mass erase or not.

```
.syntax unified

.global flashloader

.thumb

.align 2

#Keys for unlocking access to FPEC(Flash Program and Erase Controller) on STM32F10XXX
.equ FLASH_KEY1,        0x45670123
.equ FLASH_KEY2,        0xCDEF89AB

#FPEC registers
.equ FLASH_ACR,         0x40022000
.equ FLASH_KEYR,        0x40022004
.equ FLASH_OPTKEYR,     0x40022008
.equ FLASH_SR,          0x4002200C
.equ FLASH_CR,          0x40022010
.equ FLASH_AR,          0x40022014
.equ FLASH_OBR,         0x4002201C
.equ FLASH_WRPR,        0x40022020

.equ DEBUG_DCRDR,       0xE000EDF8
.equ MAGIC_WORD,        0xD33DB33F

.section .text

.word 0x20004000
.word flashloader
.word NMI_Handler
.word HardFault_Handler

NMI_Handler:
        b NMI_Handler
```

```
HardFault_Handler:
        b HardFault_Handler

flashloader:
        #disable interrupts
        cpsid i

        push {r0-r4}

        #unlock FPEC registers by writing the unlock sequence
        ldr r0,=FLASH_KEYR
        ldr r1,=FLASH_KEY1
        ldr r2,=FLASH_KEY2
        str r1,[r0]
        str r2,[r0]

        #wait for flash to be free from pending ops before proceeding, useful on sudde
        bl busy_check

        pop {r0-r4}

        #r0 contains the start address of flash write operation
        #r1 contains the start address of data in SRAM
        #r2 contains the number of bytes to write, i.e the size of data
        #r3 contains whether the request is for only mass erase or for program.

        #check if mass erase only set.
        mov r8,0x1
        cmp r8,r3
        beq mass_erase

        #prepare r8 to host SRAM base + SRAM data
        mov r8,r1
        add r8,r2

.align 2
program_loop:
        #set PG bit in flash control register.
        ldr r5,=FLASH_CR
        mov r6,0x1
        strh r6,[r5]

        #load the data in SRAM at r1 into r5
        ldr r5,[r1]

        #store lower half-word at the FLASH address given by r0
        strh r5,[r0]

        bl busy_check
```

```
        #add 2 to the FLASH address.
        add r0,0x2

        #set PG bit in flash control register.
        ldr r5,=FLASH_CR
        mov r6,0x1
        strh r6,[r5]

        #shift upper half to lower
        ldr r5,[r1]
        lsr r5,r5,16

        #store the upper half into the updated FLASH address
        strh r5,[r0]

        bl busy_check

        #add 2 to the FLASH address.
        add r0,0x2
        #mov to the next word in SRAM
        add r1,0x4

        cmp r8,r1

        bne program_loop

        #signal to the debugger that we are done.
        ldr r0,=DEBUG_DCRDR
        ldr r1,=MAGIC_WORD
        str r1,[r0]

        b .

.align 2
mass_erase:
        #start the erase operation.
        ldr r0,=FLASH_CR
        mov r1,0x4
        str r1,[r0]
        ldr r1,[r0]
        orr r1,0x40
        str r1,[r0]

        #wait for mass erase to complete
        bl busy_check

        #signal to the debugger that we are done.
        ldr r0,=DEBUG_DCRDR
```

```
        ldr r1,=MAGIC_WORD
        str r1,[r0]

        b .

.align 2
busy_check:
        push {r0-r4}

.align 2
bc_loop:
        ldr r0,=FLASH_SR
        ldr r1,[r0]
        mov r2,0x1
        and r1,r2
        beq bc_loop

        pop {r0-r4}
```

The Gist of it all.

If you have stayed this far then congratulations!

You now are familiar with:

- ARM SWD PHY layer and protocol.
- How debugging works on ARMv7-M.
- Different methods used to Program FLASH.

The last thing to explain is how this works in general tools and debuggers for example ST-link, MCU-link, J-link etc..

All of these debuggers mentioned above do the same things explained in this artitle.

The are just given the commands by your IDE running a GDB session.

All these manufacturers have their own client application running on a HOST computer.

This application communicates over USB/Ethernet to the debugger.

This application also sets up a GDB server running the GDB Remote Serial Protocol.

Our IDEs GDB session then connects to this server as a remote target to actually sent commands for debugging which are then forwareded over USB/Ethernet to the actual physical debugger.

Thats it for today folks!

I hope you learned someting out of this. :)

References

ARM Debug Interface v5 Architecture Specification

CoreSign Components TRM

ARMv7-M Architecture Reference Manual

ARM Cortex-M3 TRM / ARM Cortex-M4 TRM / ARM Cortex-M7 TRM

NXP Application Note AN11553

SiLabs Application Note AN0062

STM32F10XX Reference Manual

STM32 PM0076 Programming Manual

« PREV PAGE

The Core Wars, ARM Cortex M0+ vs M3 vs M4 vs M7

NEXT PAGE »

Machine Learning on Microcontrollers.