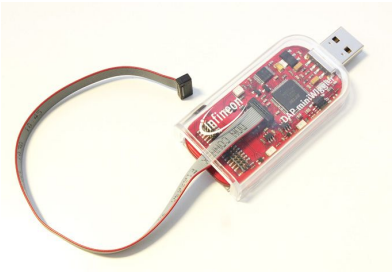
Reverse engineering of the TriCore Aurix debug protocol



Who am I?

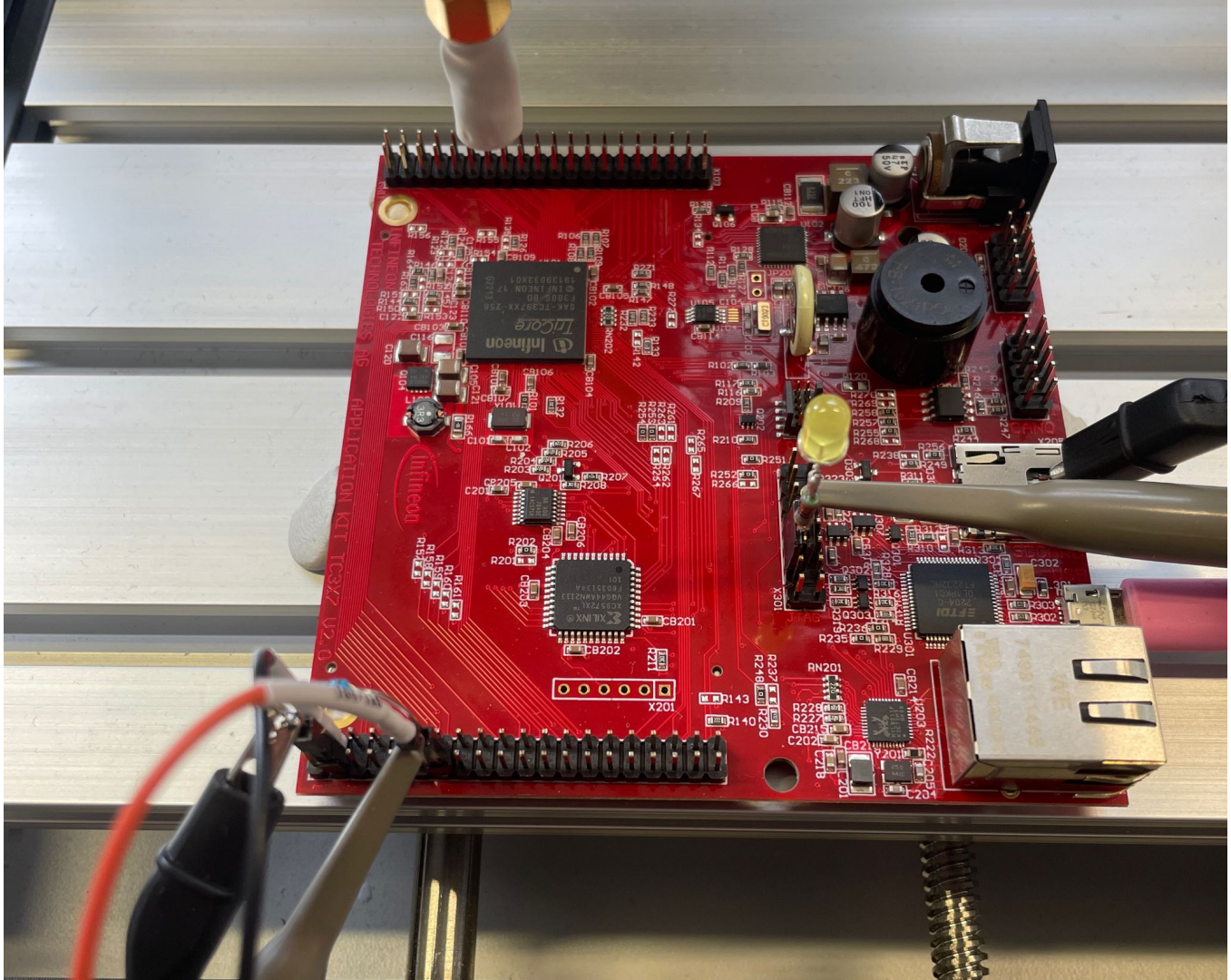
Enrico Pozzobon

- 9 years of experience in automotive IT security
- One of three founders of Dissecto GmbH
- I mostly focus on hardware attacks, such as fault injection

Context

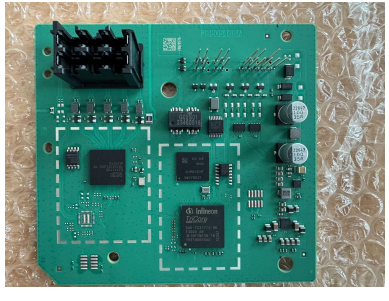
This work started during the hardware penetration test of an ECU (2 years ago)

- Attempting to use side-channel attacks and fault injection to break the security of the target, which is used as a microcontroller in a car.
- Hardware attacks can be used by the customer to bypass security checks and perform unauthorized programming (such as for tuning an engine), or by thiefs to program new keys in the immobilizer.
- Typical targets:
  - Extract secret keys
  - Alter program flow (corrupting the program counter)
  - **Unlock OCD interfaces**


Infineon
XILINX
FTDI
X201
RN201

Aurix

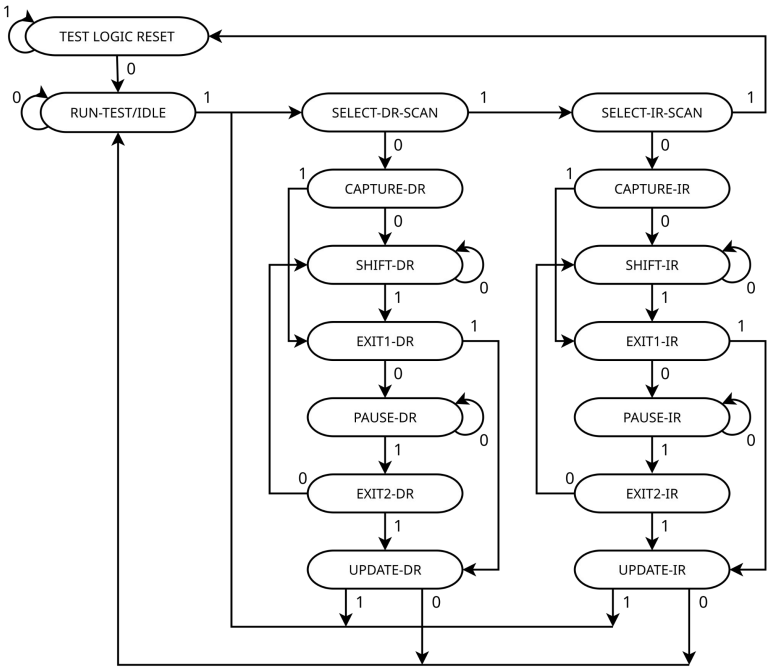
- Latest generation of Infineon Tricore
- Real-time microcontroller, software runs on bare metal, no OS required
- Mostly used in cars
  - Engine control units
  - Gateways
  - Body Control Module
  - Immobilizers



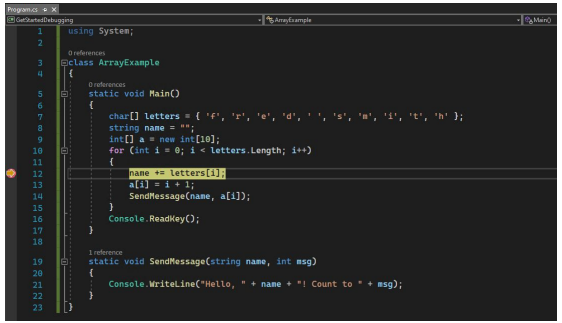
On-Chip Debug (OCD)

- Grants full control of the hardware processor
- Useful for debugging during development, initial provisioning, and failure analysis
- Often uses JTAG as a physical layer
- Different OCD protocols are used by different architectures:
  - SWD (on ARM)
  - Nexus (on Freescale PowerPC)
  - **DAP** (on Tricore)
- Typically “locked” or “censored” with a password when the device is in-field.
  - Unlocking OCD interface is a common target of fault-injection attacks.

JTAG state machine



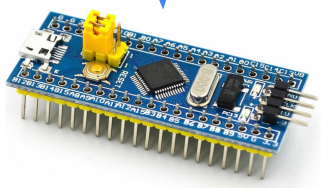
Typical OCD setup



IDE with GDB support
(GDB client)

Debug Interface

JTAG
or SWD



Target Device

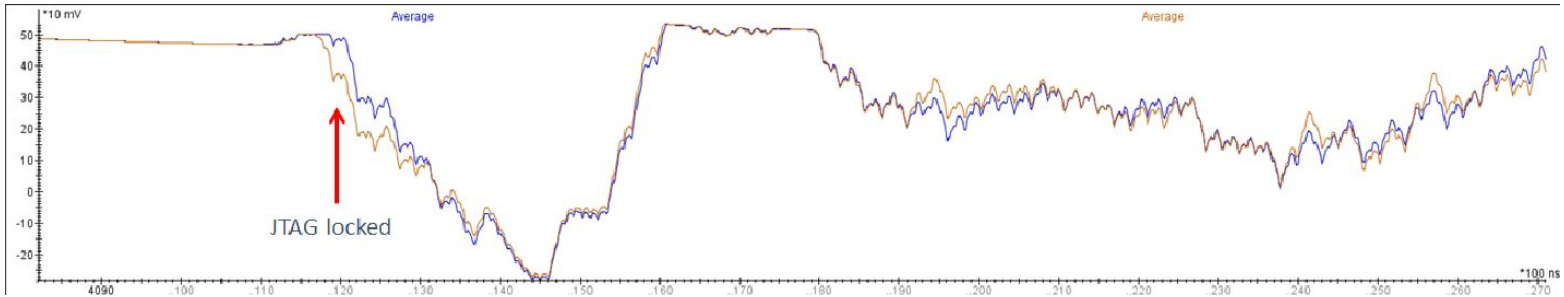
TCP

GDB Server
(e.g. OpenOCD)

USB

Typical OCD fault injection

1. On a devboard, measure power trace before and after locking OCD:



2. Reset the target, and inject a fault at the time where the OCD lock is evaulated.
3. Check if OCD is unlocked. If not, reset and retry.

It might take 10000 attempts to get one successful unlock, but each attempt only takes ~50ms

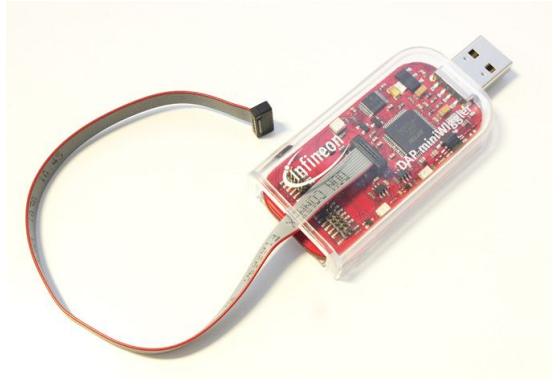
For more information: **Security ≠ Safety On the Resilience of ASIL-D Certified Microcontrollers against Fault Injection Attacks** - Nils Wiersma, Ramiro Pareja - 2017

Debug Access Port (DAP)

DAP is the OCD protocol used on Infineon Tricore microcontrollers:

- No public documentation available
- Only supported by proprietary hardware
  - Infineon Miniwiggler (150€)
  - Lauterbach Trace32 ($$$$$)
- Official Infineon tools only ran on Windows
  - More recently, some Linux tools were released

⇒ Very difficult to automate, terrible for fault injection attacks!



DAP Protocol - What is known?

- Most common variant is the 2-pin version:
  - One clock pin (DAP0)
  - One bidirectional data pin (DAP1)
- “Telegram-based” protocol
- Has CRC error detection, using CRC6, polynomial is x^6+x+1

Nothing is published regarding the framing or meaning of the individual telegrams.

Let’s change that!

Step 1: Where to sniff?

Infineon Memtool

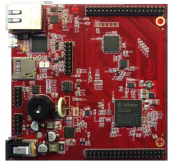
TCP

TAS Server

FTDI MPSSE
(over USB)

FT2232HL
(Miniwiggler)

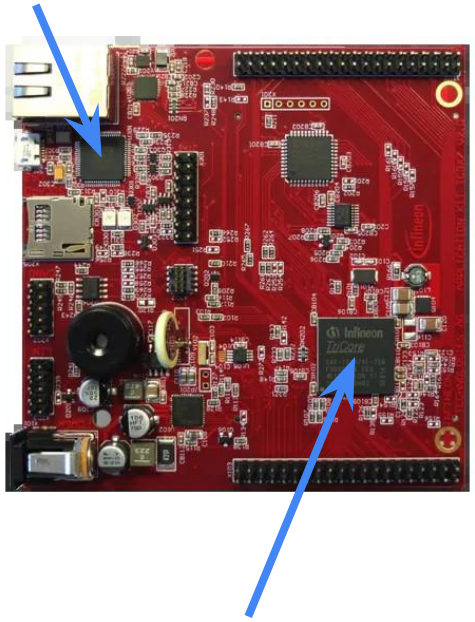
2-pin DAP



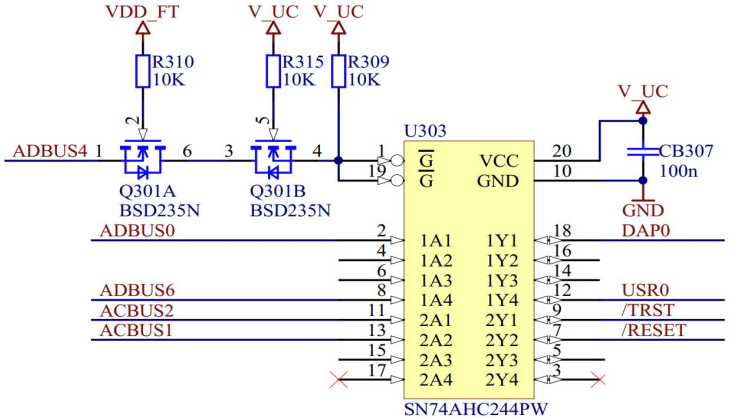
TC397
(Target)

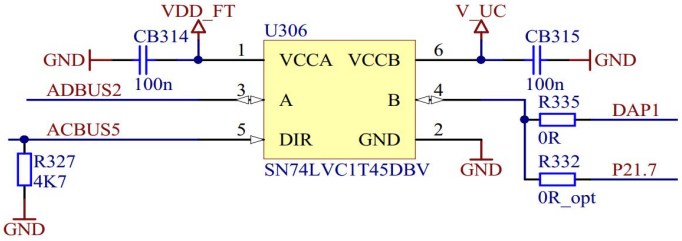
Application Kit TC3X7

Integrated Miniwiggler (FT2232HL)



TC397XX Microcontroller





Sniffing with Logic Analyzer

1Set USB Logic Analyzer Device Arduino ARM FPGA M100 Hot

4.6 ★★★★½ (137)

$9.99

Delivery **Tue, Nov 4**

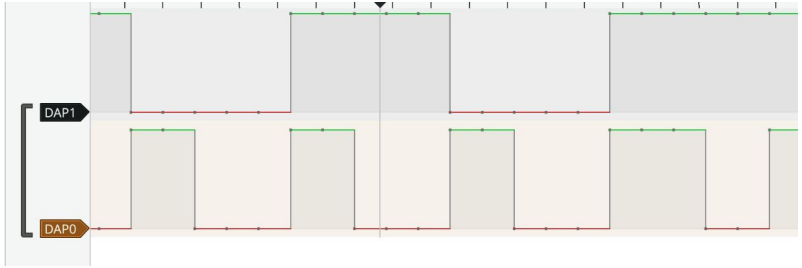
Ships to Germany

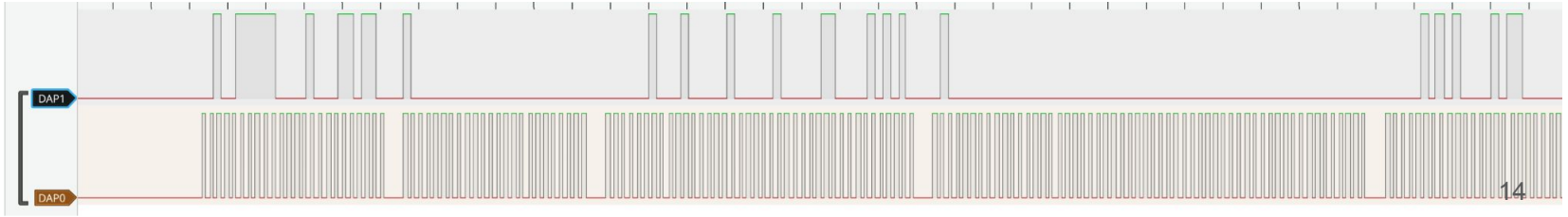
Add to cart

A cheap logic analyzer can be used to sniff the data from the 2 wires.

Evidently, data on DAP1 is sampled on the falling edge of DAP0.

We can already extract a sequence of bits out of this trace, but no direction.



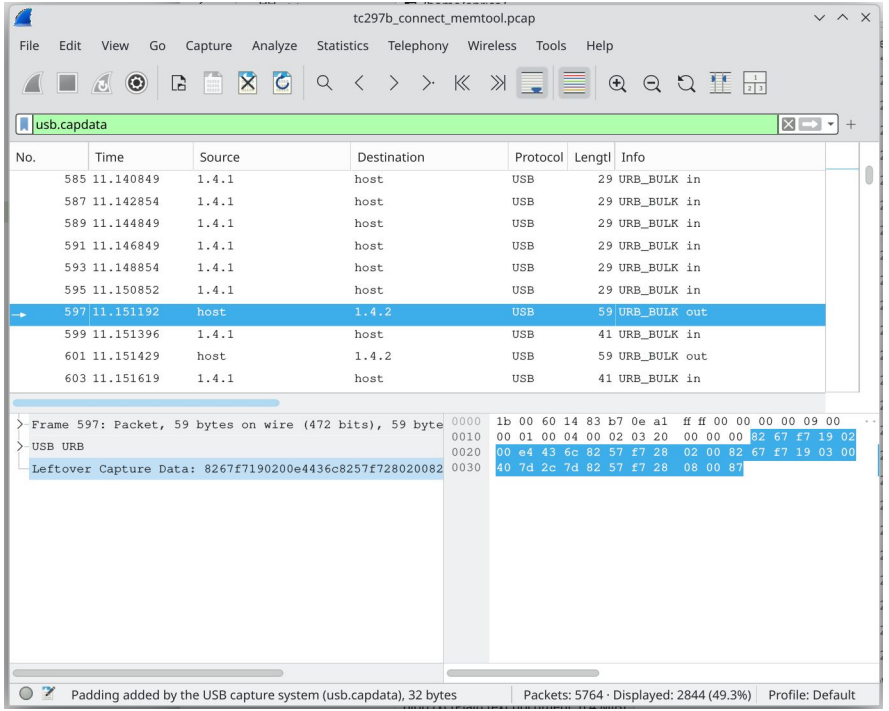


Sniffing with USBPcap

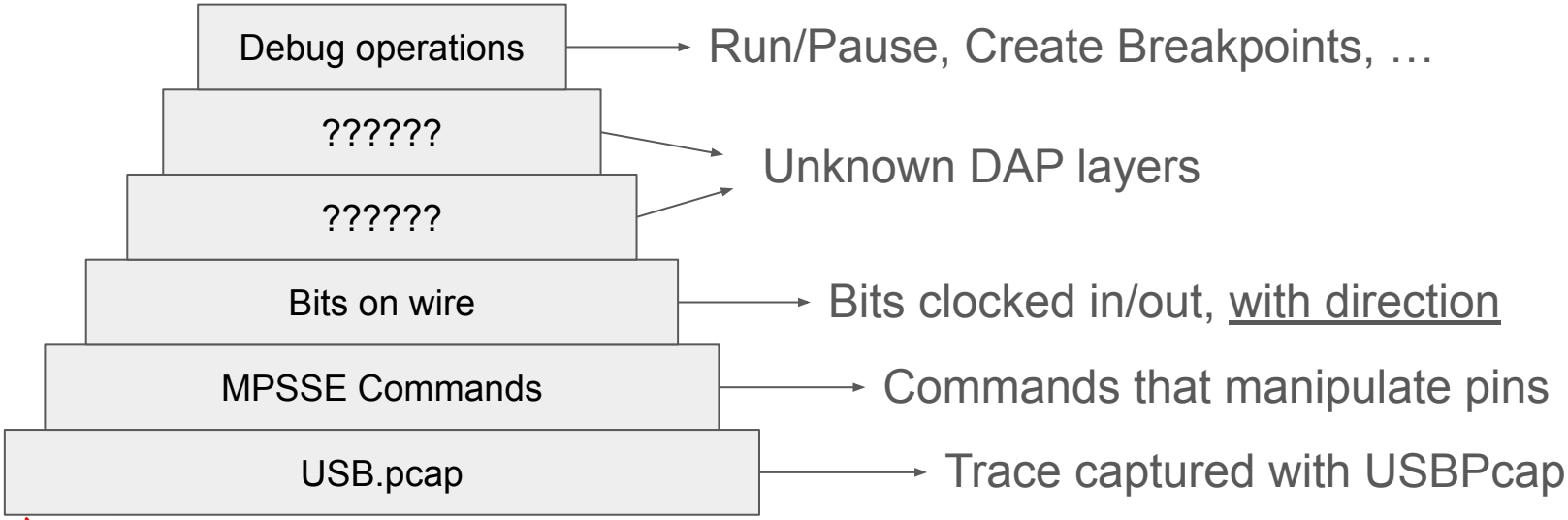
USBPcap can be used on windows to capture USB data to a pcap file.

PCAP file can be examined with wireshark and processed with Python (e.g. with Scapy).

We need to decode the MPSSE layer, but we get to see the direction of the data!



Parsing the traces:



Multi-Protocol Synchronous Serial Engine (MPSSE)

MPSSE is a hardware block which allows to efficiently encapsulate multiple protocols over USB, as well as to control individual pins synchronously (bit-banging).

The protocol is well documented and there are libraries for many languages and operating systems.

3.4 LSB FIRST

The following commands are used when data is transferred with the Least Significant Bit (LSB) first.

OPCODE	Data IN	Data OUT	BITS / BYTES	IN CLK EDGE	OUT CLK EDGE
0x18	-	YES	BYTES	-	+VE
0x19	-	YES	BYTES	-	-VE
0x1A	-	YES	BITS	-	+VE
0x1B	-	YES	BITS	-	-VE
0x28	YES	-	BYTES	+VE	-
0x2C	YES	-	BYTES	-VE	-
0x2A	YES	-	BITS	+VE	-
0x2E	YES	-	BITS	-VE	-
0x39	YES	YES	BYTES	+VE	-VE
0x3C	YES	YES	BYTES	-VE	+VE
0x3B	YES	YES	BITS	+VE	-VE
0x3E	YES	YES	BITS	-VE	+VE

See: Application Note AN_108: **Command Processor for MPSSE and MCU Host Bus Emulation Modes**

MPSSE Parsing

```
MPSSE(0x82, x'67f7')                        // Set DAP1 as output
MPSSE(0x19, x'0200e4436c')                  // Clock bytes OUT
MPSSE(0x82, x'57f7')                        // Set DAP1 as input
MPSSE(0x28, x'0200') -> x'020000'           // Clock bytes IN
MPSSE(0x82, x'67f7')                        // Set DAP1 as output
MPSSE(0x19, x'040004004413054')             // Clock bytes OUT
MPSSE(0x82, x'57f7')                        // Set DAP1 as input
MPSSE(0x28, x'0600') -> x'00000000001000'   // Clock bytes IN
MPSSE(0x82, x'67f7')                        // Set DAP1 as output
MPSSE(0x19, x'0700604209280900e063')        // Clock bytes OUT
MPSSE(0x82, x'57f7')                        // Set DAP1 as input
MPSSE(0x28, x'0600') -> x'0000c0bd6e8263'   // Clock bytes IN
MPSSE(0x82, x'67f7')                        // Set DAP1 as output
MPSSE(0x19, x'04001a00000000')              // Clock bytes OUT
MPSSE(0x82, x'57f7')                        // Set DAP1 as input
MPSSE(0x28, x'0100') -> x'0b00'             // Clock bytes IN
MPSSE(0x82, x'67f7')                        // Set DAP1 as output
MPSSE(0x19, x'06004464501200c06f')          // Clock bytes OUT
MPSSE(0x82, x'57f7')                        // Set DAP1 as input
MPSSE(0x28, x'0600') -> x'00000002000004'   // Clock bytes IN
MPSSE(0x87)                                 // Send commands immediately
```

Parsed IN/OUT bits out of MPSSE

```
Send(001001111100001000110110)
-> 000000000000000000000000

Send(000000100010000010000010000011000010101O)
-> 000000000000000000000000000000000000000100000000000

Send(000001100100001010010000001010010010000000000000000011111000110)
-> 000000000000000000000111011110101110110010000011100O110

Send(010110000000000000000000000000000000000)
-> 1101000000000000

Send(001000100010011000001010010010000000000000000001111110110)
-> 000000000000000000000000100000000000000000000000100000
```

Statistical Analysis

- Group messages by size and header
- Compute entropy for every bit position

- CRC6 bits should have highest entropy

- Figure out which bits the CRC6 is computed over

Understanding the Message Format

```
00100111110000100110110

00000010001000001000001000001100000101010

00000101010000101110100000001010010010000000000000000001111001011 0
```

- All sent messages start with a ‘1’ (start bit).
- All sent messages end with a ‘0’. If they don’t they are padded with a ‘0’.
- There can be several ‘0’ bits of padding in the beginning.
- The last 6 bits are the CRC6.

Understanding the Message Format

```
00100111110000100011011 0

000000100010000010000010000011000010101 0

00000101010000101101000000010100100100000000000000001111001011 0
```

- Bits [6:12) are the **length** of the “argument” (in bits).
- Bits [1:6) are the **command type**.
- Bits [12:12+length) are the **argument**.

Most Common Message Sequences

CMD 28:
1 00111 110000 100 011011

CMD 8:
1 00010 000010 0000100000110000 101010

CMD 10:
1 01010 000101 1010000…111 101110

<Response>

CMD 28:
1 00111 110000 100 011011

CMD 8:
1 00010 000010 0000100000110000 101010

CMD 9:
1 10010 000101 00…111 000110

<Raw Payload>
1 10101000…00000000000000

CMD 8:
1 00010 001001 100000…111 010011

Command 10: Read

Command:

1 01010 000101 10100000 00000010100001101100000000001111 001101 0

```
10 36 5,0xf0036140
```

Response:

0000000000000000000001 0100101000001001111000101101010010 01 00011100100110100001111011100000 11000000000000000000000000000000 00000000000

```
5290474b 38597807 300000000000
```

- Responses also start with a ‘1’ start bit.
- `0xf0036140` is the address of CHIPID.
- `0x4b479052` was the CHIPID of this microcontroller (TC297 in this case).
- `0x07785938` is the CRC32 (when reading 0 the CRC32 was 0xc704dd7b)

Command 9: Write

Command:

1 10010 000101 00100000 00000000101111111000000100011111 000110

9 36 4, 0xf881fd00 // 4-byte write at address 0xf881fd00 (Debug status register)

Payload:

01 11100000000000000000000000000000 00

0x00000007 // Payload of the write: {7, 0, 0, 0} (enable debug and halt)

- Writes are made out of 2 “telegrams”: **command** (contains size and address) and **payload**.
- Payload has no CRC, but still has a start bit.

Write ‘7’ to Debug Status Register: Halt execution

DBGSR — Address: $FD00_H$

Debug Status Register — Reset value: $0000\ 0000_H$

31–19	18–16
0	EVTVMN
r	rh

15–14	13–12	11–8	7	6–5	4	3	2–1	0
0	EVTSRC		PEVT	0	SUS OUT	SUSI N	HALT	DE
r	rh		rwh	r	rwh	rh	rwh	rh

Field	Bits	Type	Description
DE	0	rh	**Debug enable** Determines whether the CDC is enabled or not. 0_B The CDC is disabled. 1_B The CDC is enabled.
HALT	2:1	rwh	**CPU Halt request, status field** HALT can be set or cleared by software.HALT[0] is the actual Halt bit. HALT[1] is a mask bit to specify whether or not HALT[0] is to be updated on a software write. HALT[1] is always read as 0. HALT[1] must be set to 1 in order to update HALT[0] by software (R: read; W: write). 00_B R: CPU running. W: HALT[0] unchanged. 01_B R: CPU halted. W: HALT[0] unchanged. 10_B R: Not Applicable. W: reset HALT[0]. 11_B R: Not Applicable. W: If DBGSR.DE == 1 (The CDC is enabled), set HALT[0]. If DBGSR.DE == 0 (The CDC is not enabled), HALT[0] is left unchanged.

Unlocking a Password-Protected DAP

- Lock a new TC397 with a known password
- Unlock it, record the USB transfers
- See what additional DAP telegrams are used to unlock the interface

CMD 8 with 36-bit argument `0x76d6e24a4`

256-bit password is then sent 32-bit at a time:

CMD 8 with 36-bit argument `0x`**`deadbeef`**`4` to send `0xdeadbeef`

3.1.1.7.7 Debug System handling

The SSW internal flag Unlock Debug Interface is set to control debug access to the device, according to the following evaluation sequence:

1. The SSW checks whether an external tool has requested debug access by writing a defined content - 32-bit value CMD_KEY_EXCHANGE, defined as 0x76D6E24A for AURIX™ TC3xx - into COMDATA register
   a) if yes - continue with the next step
   b) if not - go to step 4.
2. While still in Cerberus Communication mode, the SSW confirms request reception and receives 8 further words from the COMDATA register
3. the data received (256 bits) is sent by SSW to DMU to be checked as debug interface password, and the result is evaluated by SSW:
   a) if OK - debug password is correct, set SSW internal flag, debug interface will be unlocked, exit the sequence
   b) otherwise - continue with the next step
4. instal into COMDATA a 32-bit value serving for an external tool to identify the device connected - UNIQUE_CHIP_ID_32BIT

   Note: The name here used (UNIQUE_CHIP_ID_32BIT) should be not misleading - the value considered and written into COMDATA register is NOT identifying uniquely any single device, but the product variant only. In case chip-unique identification is needed, the user Software (or the tool) should read the Unique Chip Identifier from UCB_USER - refer to the "User Configuration Block (UCB)" Section of the "Non Volatile Memory (NVM) Subsystem" Chapter.

5. check if Flash read protection is activated:
   a) if yes - debug interface will be left locked, no debug access to device, exit the sequence
   b) if not - set SSW internal flag, debug interface will be unlocked, exit the sequence

What can we do now?

By reading and writing 32-bit words to arbitrary addresses, we can:

- Dump the flash/ram memory
- Write directly to RAM
- Halt and resume execution
- Set the program counter (**`0xf88*fe08`** where * is the CPU number)
  (3.2.1 of TC1.8 architecture manual)
- Set breakpoints (**0xf88*f000**)
  (13.8 of TC1.8 architecture manual)
- Write flash
  - It is possible to write the flash directly using “Command Sequences”. (6.5.2.2.2 of TC3xx User’s Manual)
  - It is **much** faster to download a “shell” that copies data from RAM to Flash
- Unlock the DAP interface if we have the correct password

Reimplementing it with open source tools

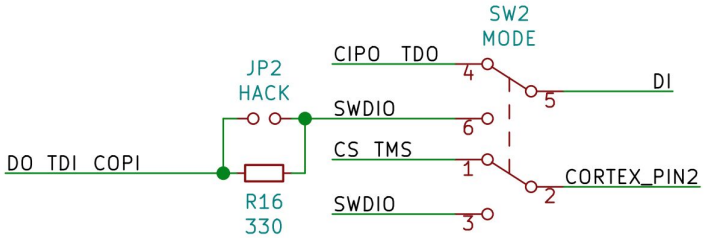
The shown DAP functions were reimplemented to work with a purely open source stack:

Hardware: Tigard

- Open Hardware
- Based on FT2232H, identical to Miniwiggler
- No “direction” pin,
  - but we can use the SWD switch to connect DI and DO with a resistor

Software: Python

- pyftdi module
- batch-based implementation of DAP for **deterministic timings**

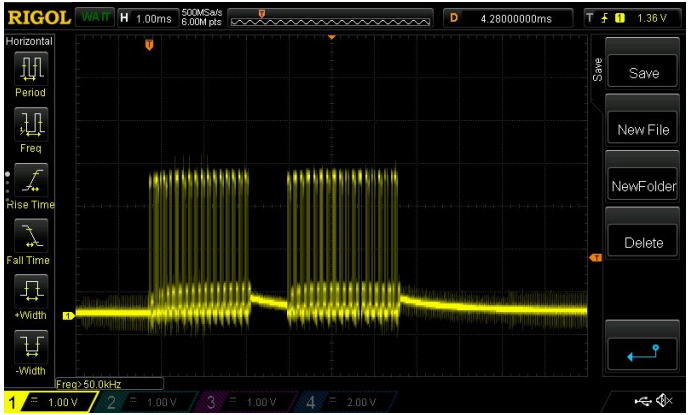


Live demonstration

(if there is time)

Why is this useful?

- Fast and deterministic DAP operations
  - One attempt at unlocking DAP now takes less than 10ms.
  - Great for hardware attacks!



- Removes the need for Windows or expensive tools.

Future Work

- Figure out the other DAP telegrams meaning
- Contribute to OpenOCD
- Test hardware attacks on the DAP interface
  - Fault Injection on password entry
  - Fault Injection on reset when the DAP lock is evaluated
  - Side Channel Analysis of password entry: is the password compared directly?

Thank you for your attention!

Get the code: https://github.com/epozzobon/tricore-things

Contact me: enrico@epozzobon.it

Questions?

HACK IN BO® Winter 2025 Edition 25ª EDIZIONE