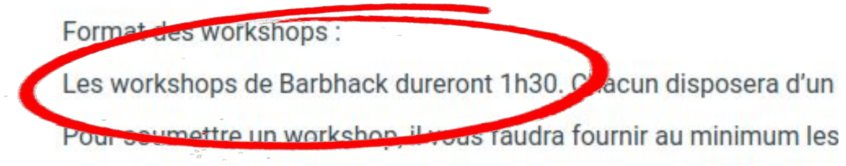
Dump all the (ARM) things !

BARBHACK

2023

Before we start

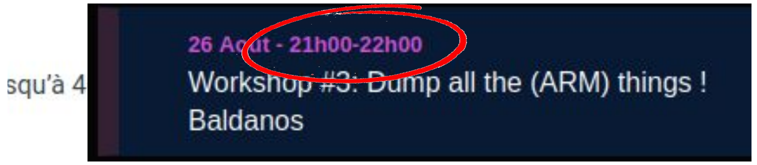
- We might be short on time
- Please download the archive
- Install python dependencies
  - pip install pyHydrabus
- Install Jupyter notebook
- A serial communication utility (screen / putty / minicom / …)



Format des workshops :

Les workshops de Barbhack dureront 1h30. Chacun disposera d'un

Pour soumettre un workshop, il vous faudra fournir au minimum les



squ'à 4

26 Août - 21h00-22h00

Workshop #3: Dump all the (ARM) things !

Baldanos

About us

- Swiss hardware hackers
- Contributors to Hydrabus project
- Conferences / CTF / BBQ enthusiasts

About this workshop

- **Introduction**
  - Hardware hacking 101
  - JTAG

- **Discover the ARM debug interface (SWD)**
  - Architecture, protocols and signals
  - Identification process and low-level interaction
  - Firmware extraction

- **Firmware RE**
  - Ghidra import process and mapping
  - CTF

Please respect workshop materials

This equipment is made available to you thanks to our personal investment

It could be used for future workshops

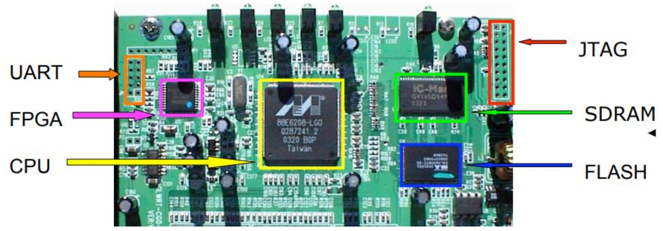
Introduction

Hardware hacking 101 🤯

- Not as difficult as it sounds !

- Similar to traditional penetration testing…but different…

- First step consists to identify the target, more specifically its:
  - Exposed interfaces: Serial, USB, RJ45, BLE, WIFI, …
  - Internal components: Micro controller, memory, …
  - Internal signals: UART, I2C, SPI, JTAG, SWD, …



Debug interface

- Manufacturers tend to leave a debug interface on their devices
  - Useful to reprogram or troubleshoot faulty products

- Some debug interface allows to read-back internal memory
  - Recover the firmware for fun & profit !

- Usual debug interfaces are usually **JTAG** or **SWD** interfaces

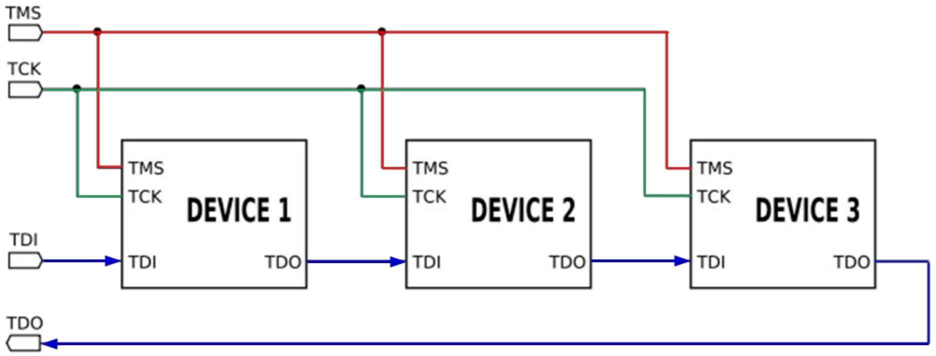
- Debugging can be achieved with Open On-Chip Debugger (OpenOCD)

JTAG - Joint Test Action Group

- Historically used to test circuits
  - Boundary scan
  - Read/Write chip pins

- Nowadays used for debugging
  - Instruction tracing/stepping
  - Memory access
  - …

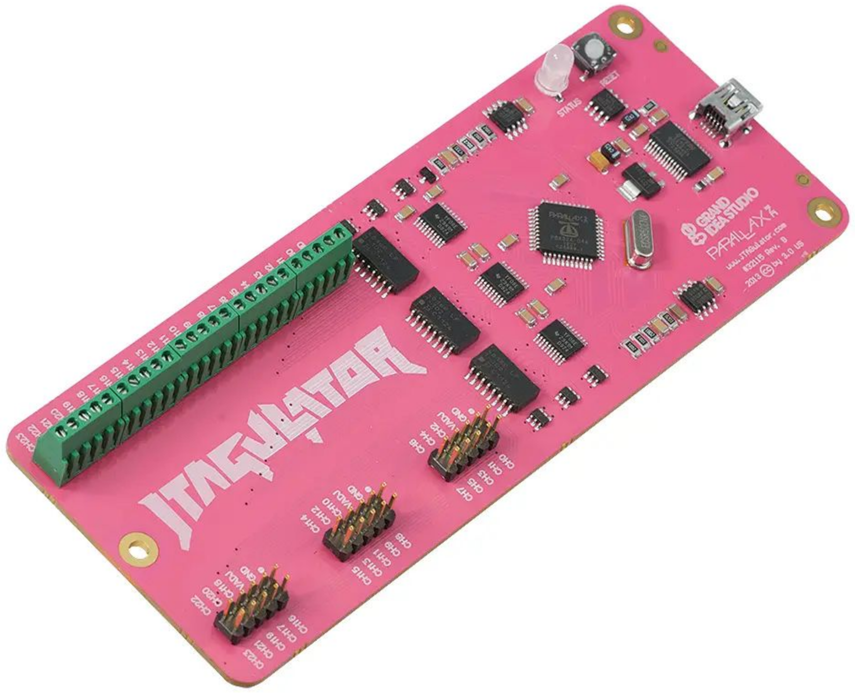
- Main drawback <u>at least</u> 4 signals required



JTAG - Discovery

- Identification issues
  - Labels not always present near each pins
  - Sometimes only test points are available

- Identification technique
  - Wire all pins to the tool
    - JTAGulator
    - Hydrabus
  - Probe every pin one after the other
  - Send the **IDCODE** command (0x00000000)
  - Forces every nodes to return their ID
  - Monitor the output

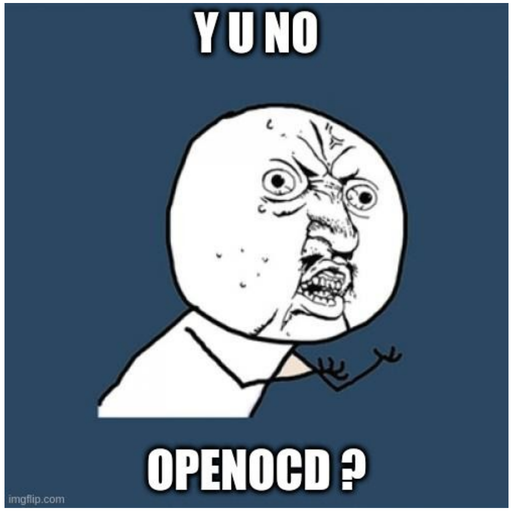


SWD - Serial Wire Debug

- ARM solution to avoid JTAG pin requirement
- Provide similar functionality as normal JTAG
- However, daisy-chaining devices as JTAG is not possible

Why using low-level SWD ?

- OpenOCD does lots of “magic” behind the scenes
- Some bugs can only be triggered when precisely controlling the interface
  - e.g. Race condition on STM32 chips
    https://www.aisec.fraunhofer.de/en/FirmwareProtection.html
- Can analyze less documented custom APs
- Offer a faster access to SWD
  - Useful for fault attacks (e.g. nRF52x)



SWD - Signals

- **SWCLK**
  - The clock signal sent by the host
  - The frequency is defined by the host interface

- **SWDIO**
  - The bidirectional data signal to read from or write to the DP
  - The data is set by the host during the rising edge and sampled by the DP during the falling edge of the SWDCLK signal

- Both lines should be pulled up on the target

SWD - Transactions

- Each transaction has 3 phases:
  - **Request**
    - 8 bits sent from the host
  - **ACK**
    - 3 bits sent from the target
  - **Data**
    - Up to 32 bits sent from/to the host, with an odd parity bit

- On direction change a **Trn** cycle has to be sent

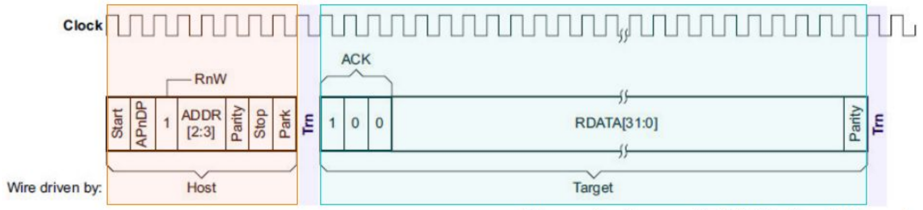


Figure 4 - Successful SWD Read Transfer

SWD - Request

Field	Description
Start	Start bit (Should be 1)
APnDP	Access to DP(0) or AP(1)
RnW	Write(0) or Read(1) request
A[2:3]	AP or DP register address bits[2:3]
Parity	Odd parity over (APnDP, RnW, A[2:3])
Stop	Stop bit (Should be 0)
Park	Park bit sent before changing SWDIO to open-drain (Should be 1)

SWD - ACK

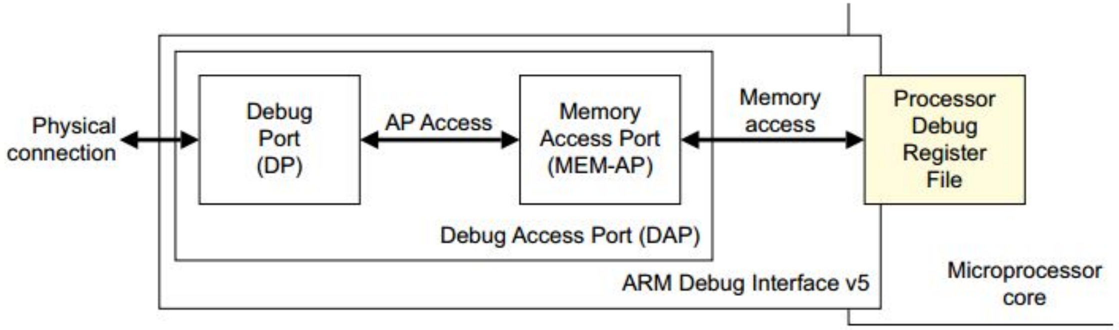
Bit	Description
2	OK response Operation was successful
1	WAIT response Host must retry the request.
0	FAULT response An error has occurred

SWD - Architecture

- Master
  DP – Debug Port

- Internal bus
  DAP - Debug Access Port

- Slaves
  AP – Access Ports



SWD - RTFM

- Everything is explained in the ARM Debug Interface (ADI) architecture specification
  - https://developer.arm.com/documentation/ihi0031

- Official documentation for the debug interface

Debug Port

DP - Debug Port

- Manages the communication with the external host
  - Forwards communication to DAP internal BUS

- 3 main DP types
  - JTAG Debug Port (JTAG-DP)
    - Standard JTAG interface and protocol

  - Serial Wire Debug Port (SW-DP)
    - SWD protocol to access the DAP

  - Serial Wire/JTAG Debug Port (SWJ-DP)
    - Switch between JTAG and SWD via a specific sequence
    - TMS/TCK are reused for SWDIO/SWCLK signals

DP - Registers

- Four registers control the DP
  - **IDCODE/ABORT (@ 0x0)**
    - Identification code register (R)
    - Transaction abortion/error management (W)
  - **CONTROL/STATUS (@0x4)**
    - Manage DP status
  - **SELECT (@ 0x8)**
    - Select AP and AP bank to be contacted
  - **RDBUFF (@ 0xC)**
    - Read buffer

SWD - Interface initialization

1. Send at least 50 SWCLKTCK cycles with SWDIO/TMS HIGH
   a. Ensures that the current interface is in its reset state
   b. The JTAG interface only detects the 16-bit JTAG-to-SWD sequence starting from the Test-Logic-Reset state

2. Send the 16-bit JTAG-to-SWD select sequence on SWDIO/TMS

3. Send at least 50 SWCLKTCK cycles with SWDIO/TMS HIGH
   a. Ensures that if SWJ-DP was already in SWD operation before sending the select sequence, the SWD interface enters line reset state

DP - Initialization

- Once interface is initialized, the DP must be initialized as well
  - Read IDCODE register
  - Power up the debug domain by setting bits in the CTRL/STAT register

- Now ready to talk to Access Ports

LAB 1

SWD Discovery

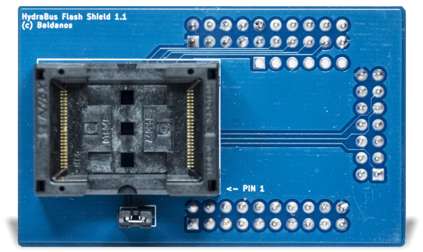
Workshop kit

- Hydrabus
- Wires (~10x)
- Target

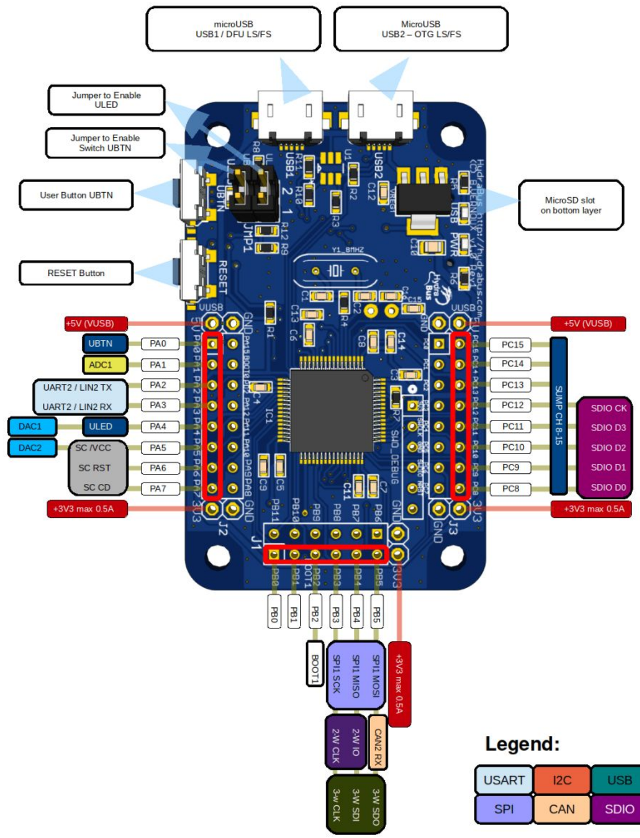
Hydrabus

- Open source multi-tool hardware
    - Created by Benjamin Vernoux
- Supports a lot of protocols
- Python bindings (pyHydrabus)
- Extensions via specific shields
    - HydraFlash
    - HydraLINCAN
    - HydraNFC

Hydra Bus

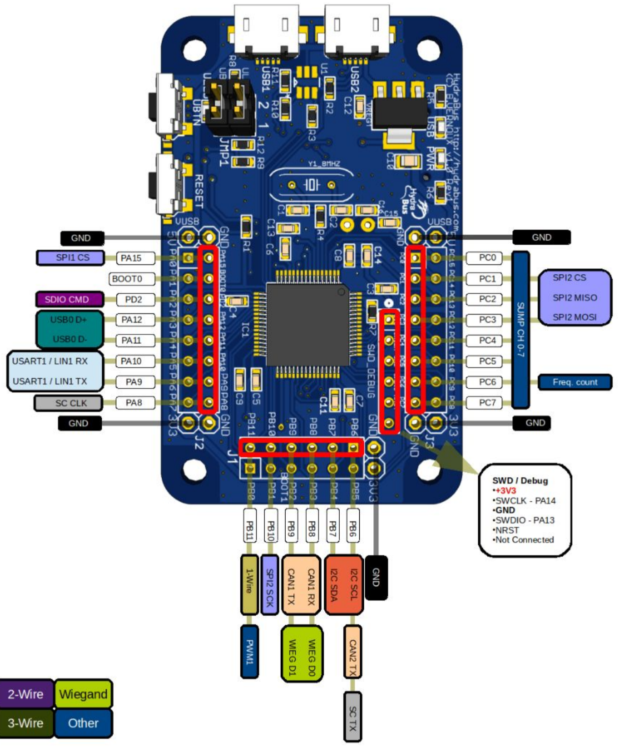


Hydrabus - Main features



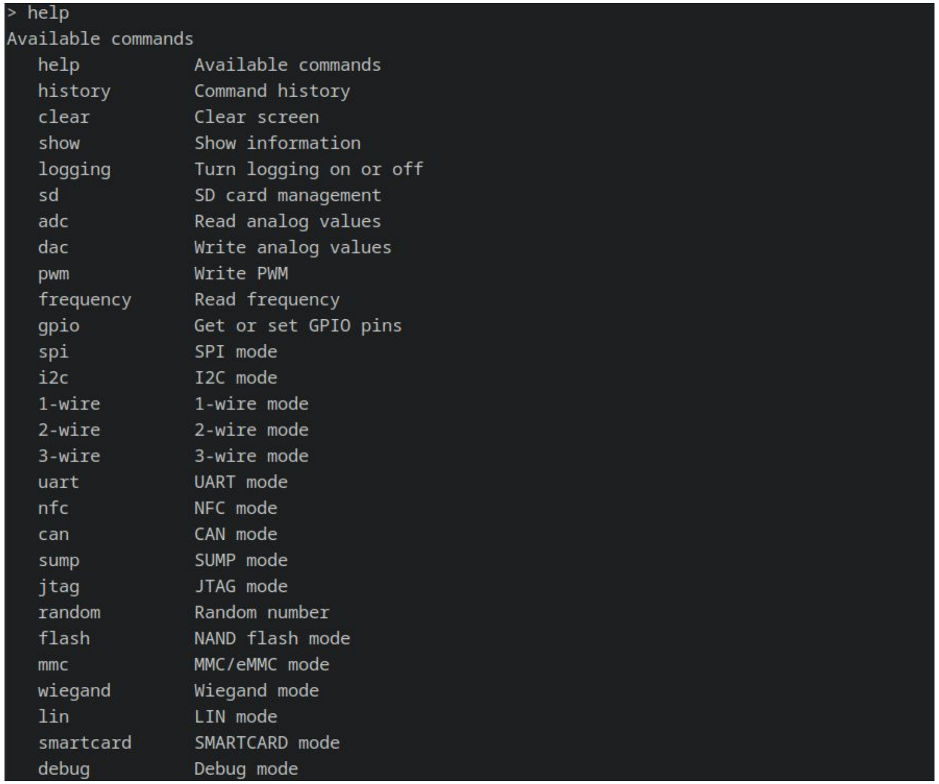
Legend:

USART	I2C	USB	ADC	Smartcard	2-Wire	Wiegand
SPI	CAN	SDIO	DAC	1-Wire	3-Wire	Other



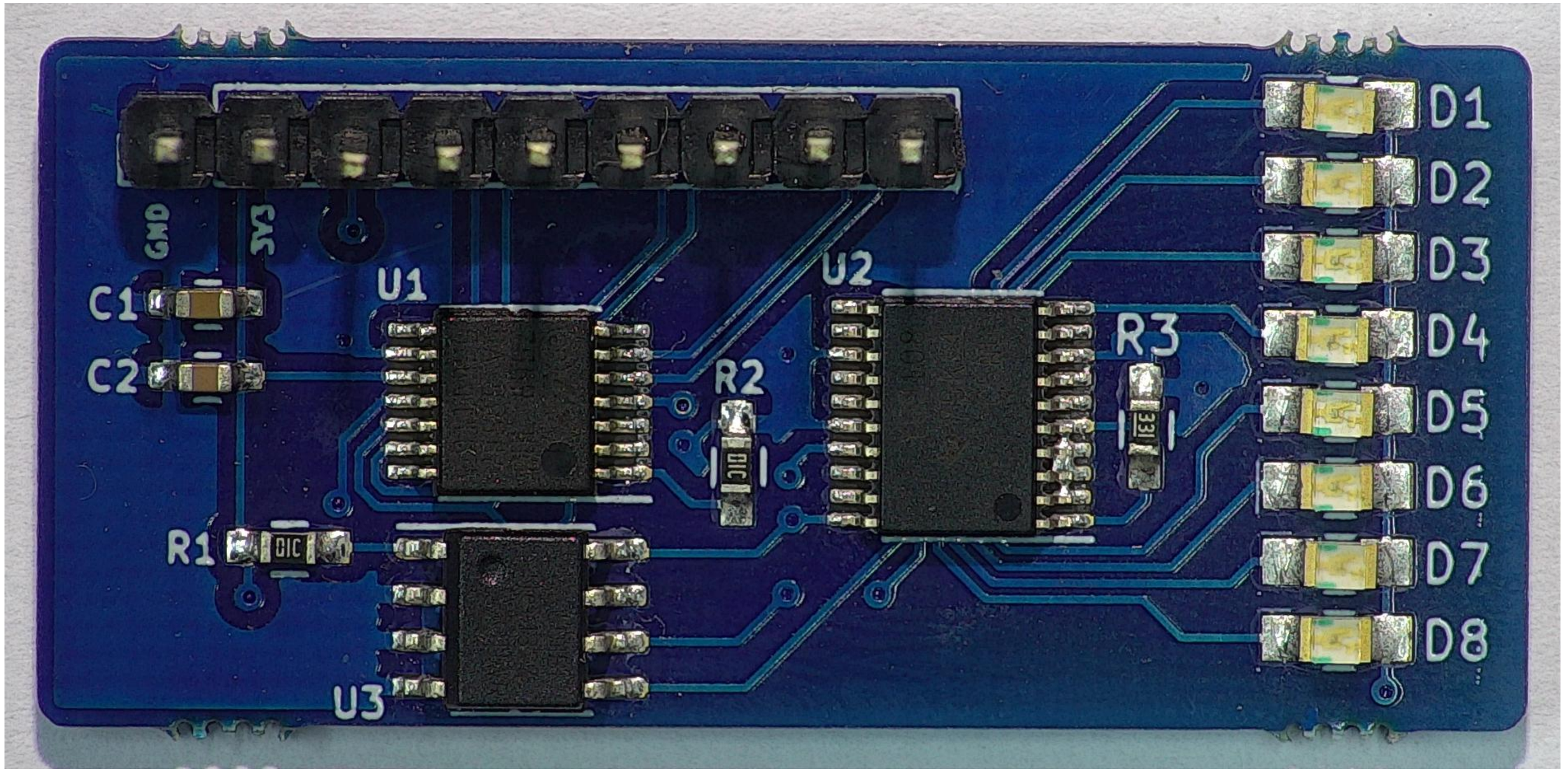
Hydrabus - CLI

- *help* command shows the commands used in each mode
  - The prompt will show you which is the current mode
- The CLI Supports *Tab* completion
- Once in a mode, protocol-specific commands will be shown
- The *show pins* command shows which pins are used for each mode
- More info on the HydraFW wiki
  - https://github.com/hydrabus/hydrafw/wiki



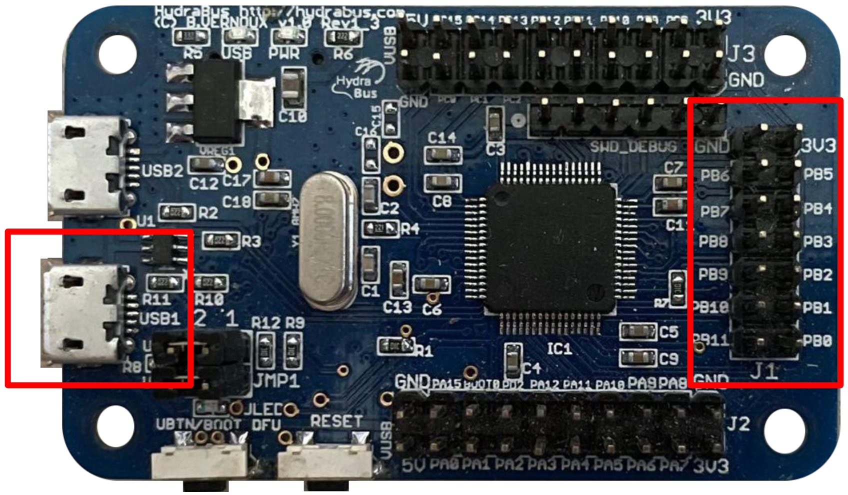
```
> help
Available commands
   help           Available commands
   history        Command history
   clear          Clear screen
   show           Show information
   logging        Turn logging on or off
   sd             SD card management
   adc            Read analog values
   dac            Write analog values
   pwm            Write PWM
   frequency      Read frequency
   gpio           Get or set GPIO pins
   spi            SPI mode
   i2c            I2C mode
   1-wire         1-wire mode
   2-wire         2-wire mode
   3-wire         3-wire mode
   uart           UART mode
   nfc            NFC mode
   can            CAN mode
   sump           SUMP mode
   jtag           JTAG mode
   random         Random number
   flash          NAND flash mode
   mmc            MMC/eMMC mode
   wiegand        Wiegand mode
   lin            LIN mode
   smartcard      SMARTCARD mode
   debug          Debug mode
```

Target



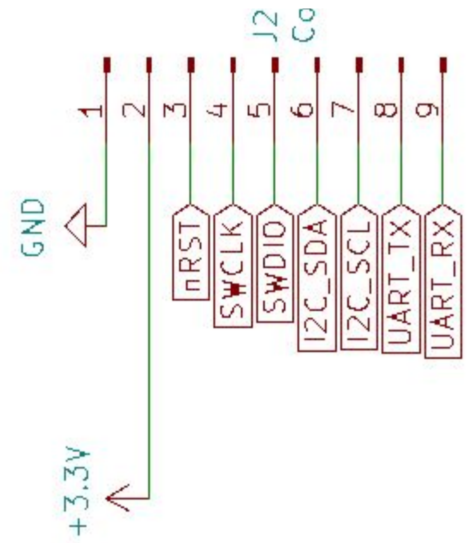
Exercise - Finding SWD !

- Connect the target to Hydrabus
  - GND first!
  - Then all other target pins to PBx
  - Power supply is 3.3V !
- Connect Hydrabus to the PC via USB
  - Access the CLI
    - Putty, telnet (Windows)
    - Screen (Linux)
- Enter 2-wire mode
- Use the integrated bruteforce utility



Solution

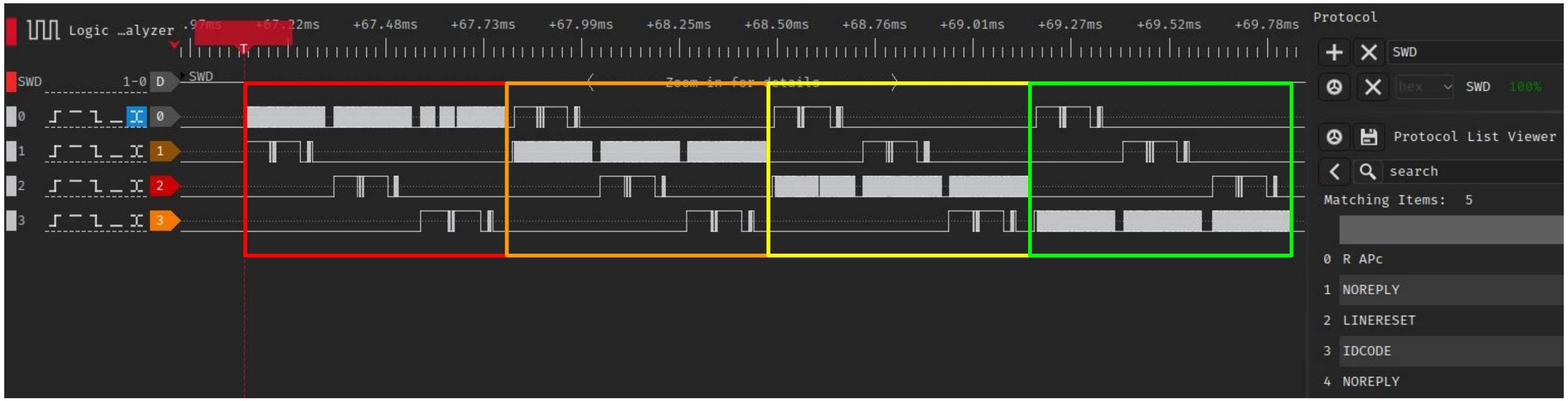
```
> 2-wire
twowire1> brute7
Bruteforce on 7 pins.
Device found. IDCODE : 0x3BC11477
CLK: PBx        IO: PBx
```



How does it work ?

- Hydrabus will treat all pins combination as SWCLK/SWDIO, then for each combination:
  - Send SWD initialization
  - Read DP IDCODE

- If different than 0x0 or 0xffffffff, display to the user



Matching Items: 5

```
0  R APc
1  NOREPLY
2  LINERESET
3  IDCODE
4  NOREPLY
```

Access Port

AP - Access Ports

- Access Ports allow access to the target
- ARM provides specifications for two APs
  - Memory Access Port (MEM-AP)
    - Provides access to the core memory and registers
  - JTAG Access Port (JTAG-AP)
    - Allows connection of a JTAG chain to the DAP
- Multiple APs can be added to the DAP if needed

AP - Access

- Must use the DP to access an AP
- Same as with DP, only 4 registers accessible
  - Extended with banks in the DP SELECT register
- Only one common register: Identification register IDR (@ 0xfc)
  - Bank 0xf, register 0xc

AP - Registers

- Access to a register is made through the DP
- Several steps needed:
    - Set AP address and register bank in DP SELECT register
    - Issue a request to an AP register
    - If necessary, read DP RDBUFF to get return value

MEM-AP

- AP dedicated to the core memory
  - Allows access to all the MCU memory space
- Many advanced features, but we will only present basic memory reads/writes

MEM-AP - Registers

- **CSW** - Control/Status Word (**@ 0x0**)
  - Control MEM-AP features
- **TAR** - Transfer Address Register (**@ 0x4**)
  - Set memory address
- **DRW** - Data Read Write Register (**@ 0x0c**)
  - Data to be read/written

MEM-AP - Memory read

- To read value @ 0x12345678 :
  - Set TAR to 0x12345678
  - Read DRW
- Easy ?

MEM-AP - Complete memory read

- Set DP SELECT register to bank 0, AP 0
- Write 0x12345678 to AP register 0x4 (TAR)
- (Set DP SELECT register to bank 0, AP 0) (Optional)
- Read to AP register 0xC (DRW)
- Read DP register 0xC (RDBUFF)

Core control

- Cortex-M CPUs can be controlled through special memory-mapped registers
- Allow access to CPU registers, control core, …
- DHCSR - Debug Halt and Control Register (@ 0xE000EDF0)
  - Allow to stop core execution, enable debug, ...

Firmware extraction

- Firmware is located at a specific memory location
    - Depends on manufacturer, verify memory map in the datasheet

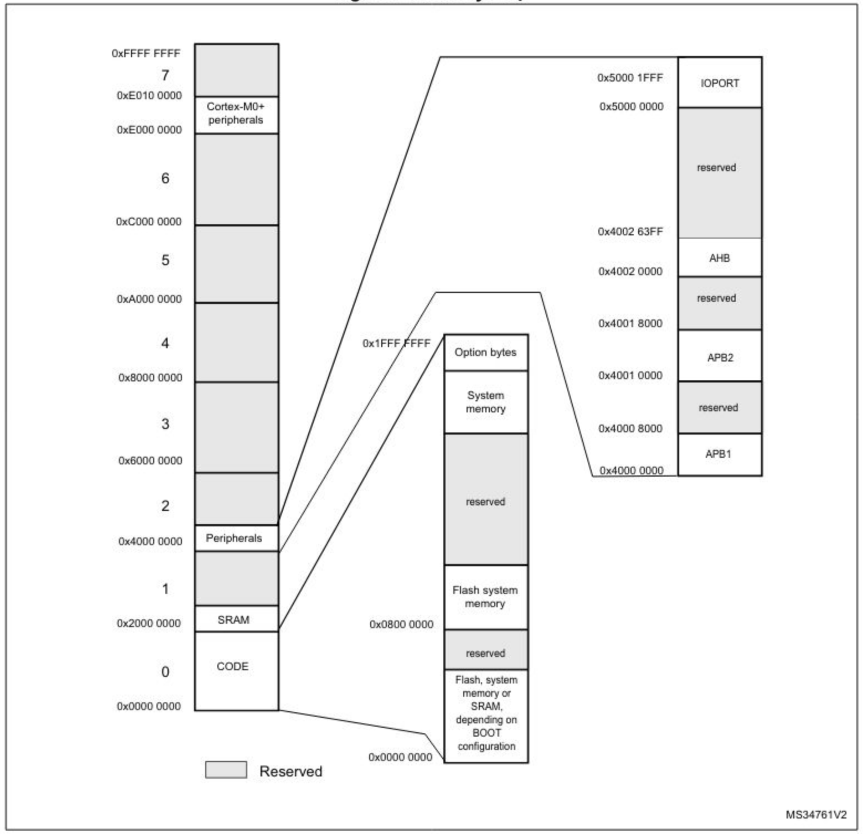
- Access to the firmware must be done with CPU halted

Memory maps

- Extremely useful when reverse engineering a MCU firmware
  - Once the memory map is defined, you can cross-reference any register to find related functions
- Can give a starting point for more complex firmwares
  - Example : Looking for any UART-related functions in a firmware

Example : STM32L011

Figure 2. Memory map



Instrumenting SWD

- For Hydrabus: pyHydrabus has SWD primitives
    - Python bindings (https://pypi.org/project/pyHydrabus/)
        - pip install pyHydrabus

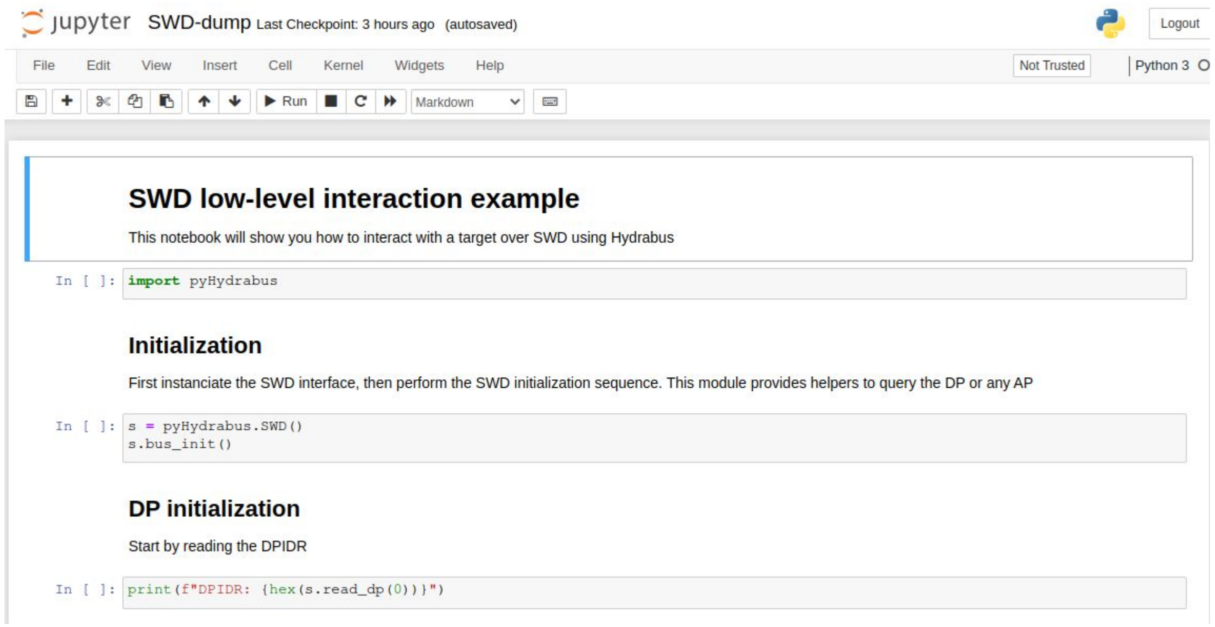
    - AP/DP read/write

LAB 2

Firmware extraction

Exercise - Firmware extraction

- Use the provided Jupyter notebook
- You should end with a file named firmware.bin


jupyter
SWD-dump Last Checkpoint: 3 hours ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Markdown

SWD low-level interaction example

This notebook will show you how to interact with a target over SWD using Hydrabus

```
In [ ]: import pyHydrabus
```

Initialization

First instanciate the SWD interface, then perform the SWD initialization sequence. This module provides helpers to query the DP or any AP

```
In [ ]: s = pyHydrabus.SWD()
        s.bus_init()
```

DP initialization

Start by reading the DPIDR

```
In [ ]: print(f"DPIDR: {hex(s.read_dp(0))}")
```

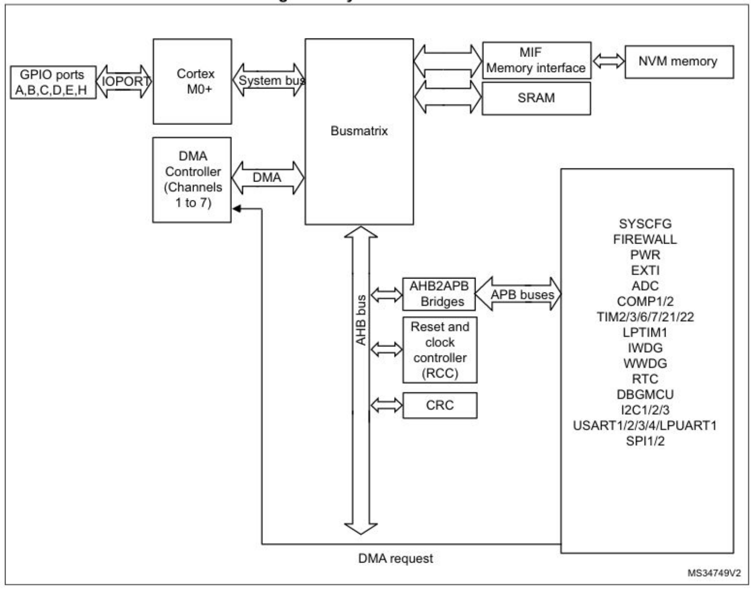
MCU

MCU - Micro Controller Unit

- A single chip embeds a processor, memory and peripherals

- Lots of different options
  - Packaging
  - CPU core
  - Memory capacity
  - Peripherals

Figure 1. System architecture



MCU - Peripherals

- All peripherals and capabilities use memory-mapped registers
    - In the MCU memory space, a region is dedicated to those
- When using peripherals, you basically have to read and write to specified memory locations
- All memory locations are defined in the MCU datasheet
- Called MMIO (Memory-Mapped I/O)

Interrupts

- The MCU can generate interrupts for different events
  - More effective than polling for events in the code
- When an interrupt is enabled
  - MCU stops the current execution
  - Branches to the function pointed at a fixed address
- These fixed addresses are called interrupt vectors
- These interrupts are enabled by the developer

ARM

ARM - CPU architecture

- 16 registers
  - R0 – R15
- R15 is also known as PC (Program Counter)
  - Equivalent to **EIP** in x86
- R14 is also known as LR (Link register)
  - Stores the return address of a function call
- R13 is also known as SP (Stack Pointer)

ARM - Assembly 101

- Different instruction sets
  - ARM (32 bits instructions)
  - Thumb (16 bits instructions)
  - Thumb-2 (16 & 32 bit instructions)
  - NEON / Jazelle / … (not part of this course)

- Most embedded firmwares use Thumb(-2) instructions

- An ARM CPU can switch from ARM to THUMB mode on the fly
  - When calling a function, the LSB represents the mode
    0=ARM, 1=THUMB

ARM - Thumb basic instructions

Instruction	**Meaning**
MOV R0, #5	R0 = 5
ADD R0, R1, R2	R0 = R1 + R2
SUB R0, #10	R0 = R0 - 10
CMP R0, R3	Calculates R0 – R3, sets flags accordingly
BX R8	Jumps (Branches) execution to R8
LDR R4, [PC, #22]	R4 = [PC+22]
STR R3, [R2, R6]	[R2+R6] = R3
PUSH {R0-R3, LR}	Pushes R0, R1, R2, R3, LR on the stack

ARM - Branch instructions

Instruction	**Meaning**
BEQ R0	Branch if Z flag is set (equal to zero)
BNE R0	Branch if Z flag is unset (not equal to zero)
BGE R0	Branch if greater or equal
BLT R0	Branch if lower than
BGT	Branch if bigger than
BLE	Branch if less or equal

Most Thumb instructions can be executed conditionally based on the values of Application Program Status Register (APSR) condition flags (e.g., Zero, Carry, Overflow, Negative).

ARM - Calling convention

- Usually, function parameters are passed in r0 to r3
- Result is stored in r0 at the end of the function

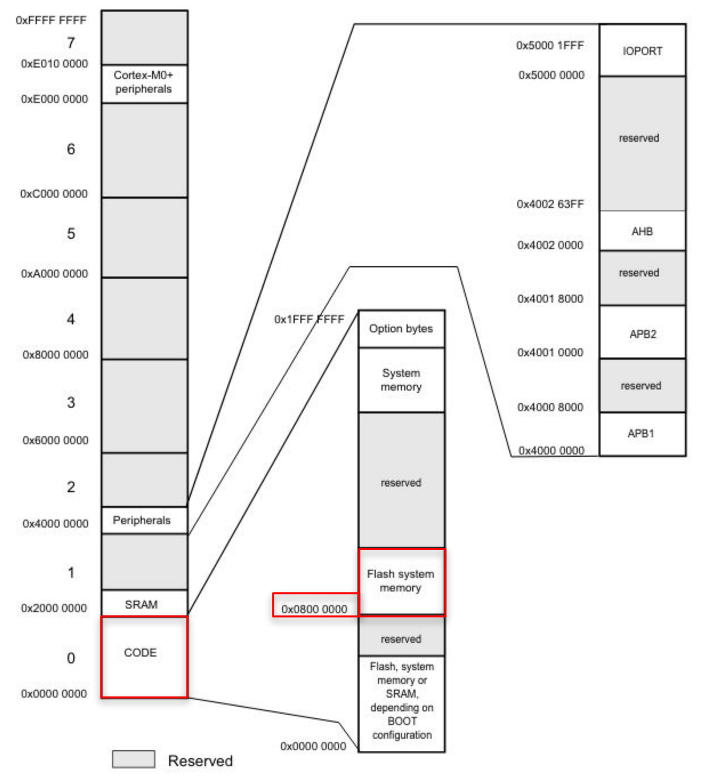
Instructions	Meaning
mov.w r0, #0x3e8 mov.w r1, #0x01 bl *function*	*function*(1000, 1)

LAB 3

Firmware analysis

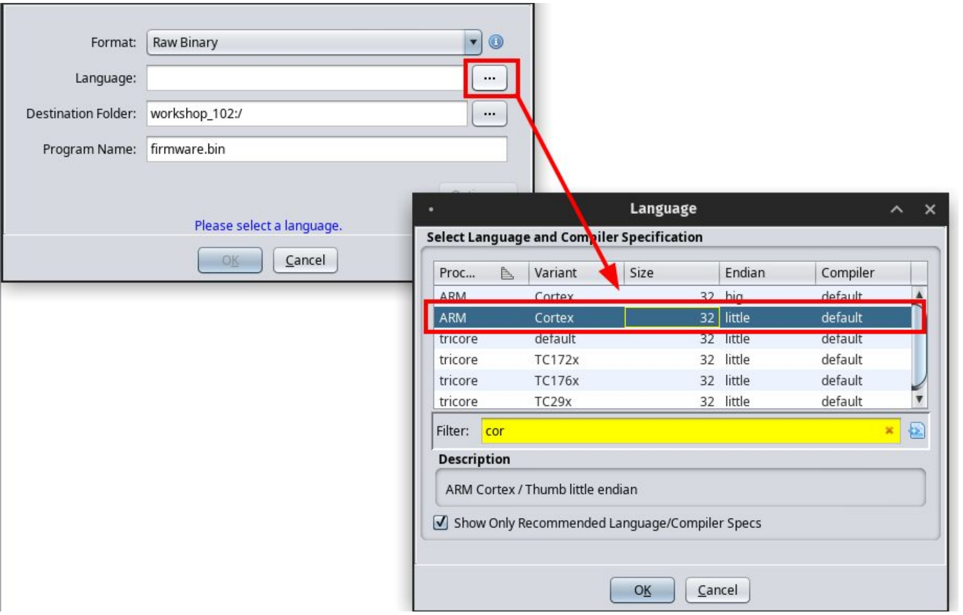
Firmware - Base address

- Update files usually contains only the MCU flash image
- The base address can be different than 0x0
  - On STM32, flash base is **0x0800_0000**
- Loading the firmware to the correct base address is crucial for RE
- If not correctly done, all references will point to the wrong place… ;(



Exercise - Firmware

Load the firmware in Ghidra



Format:	Raw Binary
Language:	
Destination Folder:	workshop_102:/
Program Name:	firmware.bin

Please select a language.

Language

Select Language and Compiler Specification

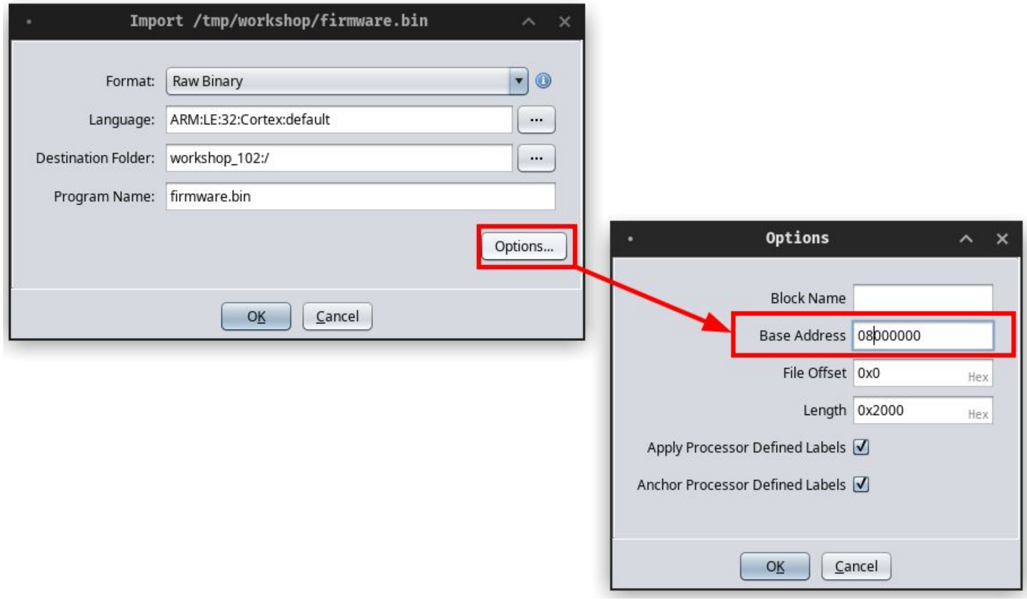
Proc...	Variant	Size	Endian	Compiler
ARM	Cortex	32	big	default
ARM	Cortex	32	little	default
tricore	default	32	little	default
tricore	TC172x	32	little	default
tricore	TC176x	32	little	default
tricore	TC29x	32	little	default

Filter: cor

Description

ARM Cortex / Thumb little endian

☑ Show Only Recommended Language/Compiler Specs



Import /tmp/workshop/firmware.bin

Format:	Raw Binary
Language:	ARM:LE:32:Cortex:default
Destination Folder:	workshop_102:/
Program Name:	firmware.bin

Options

Block Name	
Base Address	08000000
File Offset	0x0 Hex
Length	0x2000 Hex
Apply Processor Defined Labels	☑
Anchor Processor Defined Labels	☑

Firmware - Memory map

- Once code is loaded, we need to reconstruct the memory map

- Knowing registers addresses can help A LOT
  - MMIO
  - Need to get the memory map from the datasheet

- Reconstructing the interrupt vector table also helps
  - Normally in the datasheet
  - At least get the initial PC value

Firmware - Peripherals

CMIS-SVD

Common Microcontroller Software Interface Standard (CMSIS)

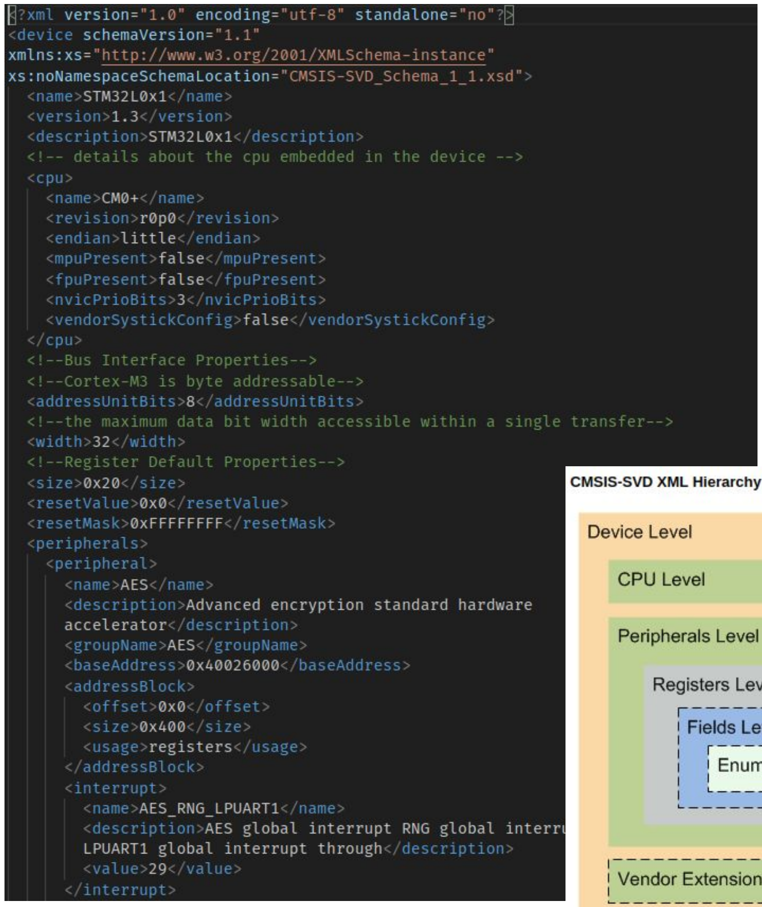
System View Description (SVD)

https://github.com/posborne/cmsis-svd

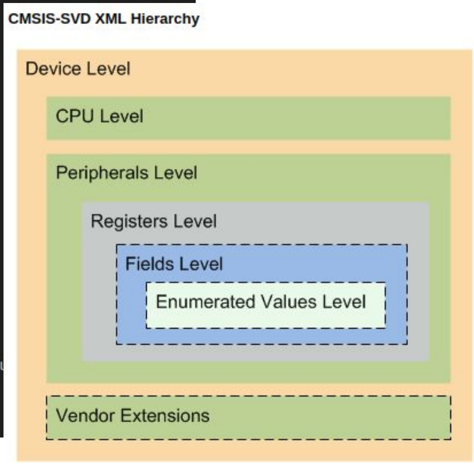
SVD-Loader

Python script for Ghidra which parses SVD files and generates the peripheral structs and memory maps

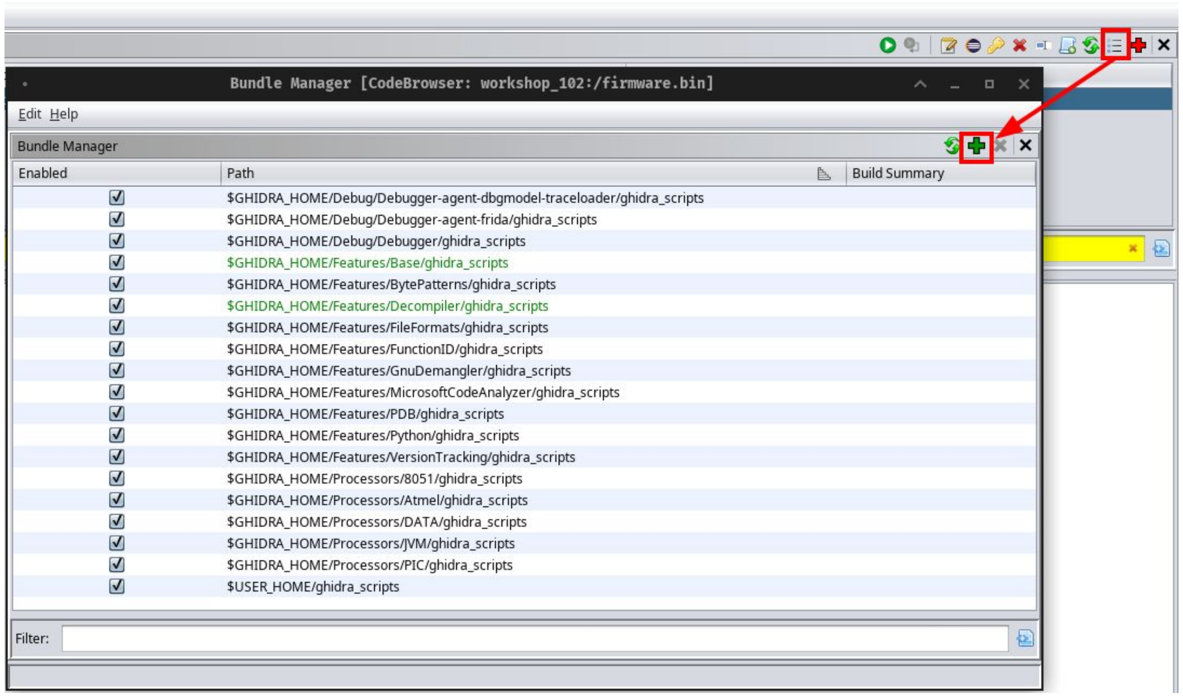
https://github.com/leveldown-security/SVD-Loader-Ghidra



```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<device schemaVersion="1.1"
xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
xs:noNamespaceSchemaLocation="CMSIS-SVD_Schema_1_1.xsd">
  <name>STM32L0x1</name>
  <version>1.3</version>
  <description>STM32L0x1</description>
  <!-- details about the cpu embedded in the device -->
  <cpu>
    <name>CM0+</name>
    <revision>r0p0</revision>
    <endian>little</endian>
    <mpuPresent>false</mpuPresent>
    <fpuPresent>false</fpuPresent>
    <nvicPrioBits>3</nvicPrioBits>
    <vendorSystickConfig>false</vendorSystickConfig>
  </cpu>
  <!--Bus Interface Properties-->
  <!--Cortex-M3 is byte addressable-->
  <addressUnitBits>8</addressUnitBits>
  <!--the maximum data bit width accessible within a single transfer-->
  <width>32</width>
  <!--Register Default Properties-->
  <size>0x20</size>
  <resetValue>0x0</resetValue>
  <resetMask>0xFFFFFFFF</resetMask>
  <peripherals>
    <peripheral>
      <name>AES</name>
      <description>Advanced encryption standard hardware
      accelerator</description>
      <groupName>AES</groupName>
      <baseAddress>0x40026000</baseAddress>
      <addressBlock>
        <offset>0x0</offset>
        <size>0x400</size>
        <usage>registers</usage>
      </addressBlock>
      <interrupt>
        <name>AES_RNG_LPUART1</name>
        <description>AES global interrupt RNG global interru
        LPUART1 global interrupt through</description>
        <value>29</value>
      </interrupt>
```



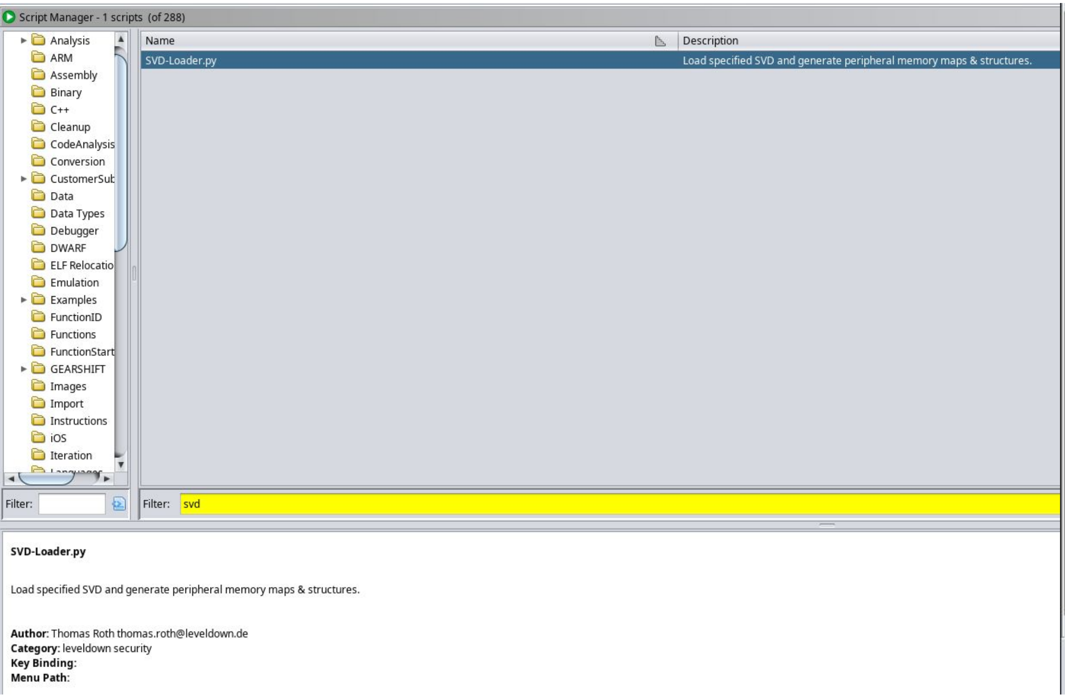
Exercise - Firmware



Bundle Manager [CodeBrowser: workshop_102:/firmware.bin]

Enabled	Path	Build Summary
✓	$GHIDRA_HOME/Debug/Debugger-agent-dbgmodel-traceloader/ghidra_scripts	
✓	$GHIDRA_HOME/Debug/Debugger-agent-frida/ghidra_scripts	
✓	$GHIDRA_HOME/Debug/Debugger/ghidra_scripts	
✓	$GHIDRA_HOME/Features/Base/ghidra_scripts	
✓	$GHIDRA_HOME/Features/BytePatterns/ghidra_scripts	
✓	$GHIDRA_HOME/Features/Decompiler/ghidra_scripts	
✓	$GHIDRA_HOME/Features/FileFormats/ghidra_scripts	
✓	$GHIDRA_HOME/Features/FunctionID/ghidra_scripts	
✓	$GHIDRA_HOME/Features/GnuDemangler/ghidra_scripts	
✓	$GHIDRA_HOME/Features/MicrosoftCodeAnalyzer/ghidra_scripts	
✓	$GHIDRA_HOME/Features/PDB/ghidra_scripts	
✓	$GHIDRA_HOME/Features/Python/ghidra_scripts	
✓	$GHIDRA_HOME/Features/VersionTracking/ghidra_scripts	
✓	$GHIDRA_HOME/Processors/8051/ghidra_scripts	
✓	$GHIDRA_HOME/Processors/Atmel/ghidra_scripts	
✓	$GHIDRA_HOME/Processors/DATA/ghidra_scripts	
✓	$GHIDRA_HOME/Processors/JVM/ghidra_scripts	
✓	$GHIDRA_HOME/Processors/PIC/ghidra_scripts	
✓	$USER_HOME/ghidra_scripts	

SVD files



Script Manager - 1 scripts (of 288)

Analysis
ARM
Assembly
Binary
C++
Cleanup
CodeAnalysis
Conversion
CustomerSub
Data
Data Types
Debugger
DWARF
ELF Relocatio
Emulation
Examples
FunctionID
Functions
FunctionStart
GEARSHIFT
Images
Import
Instructions
iOS
Iteration

Name	Description
SVD-Loader.py	Load specified SVD and generate peripheral memory maps & structures.

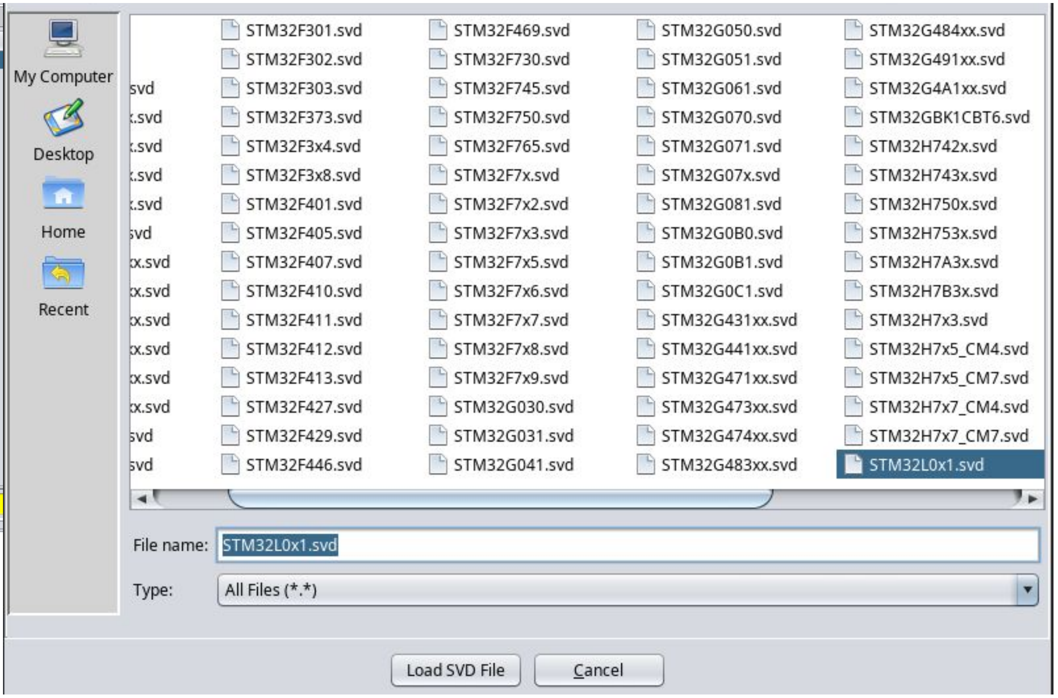
Filter:

Filter: svd

SVD-Loader.py

Load specified SVD and generate peripheral memory maps & structures.

Author: Thomas Roth thomas.roth@leveldown.de
Category: leveldown security
Key Binding:
Menu Path:



My Computer
Desktop
Home
Recent

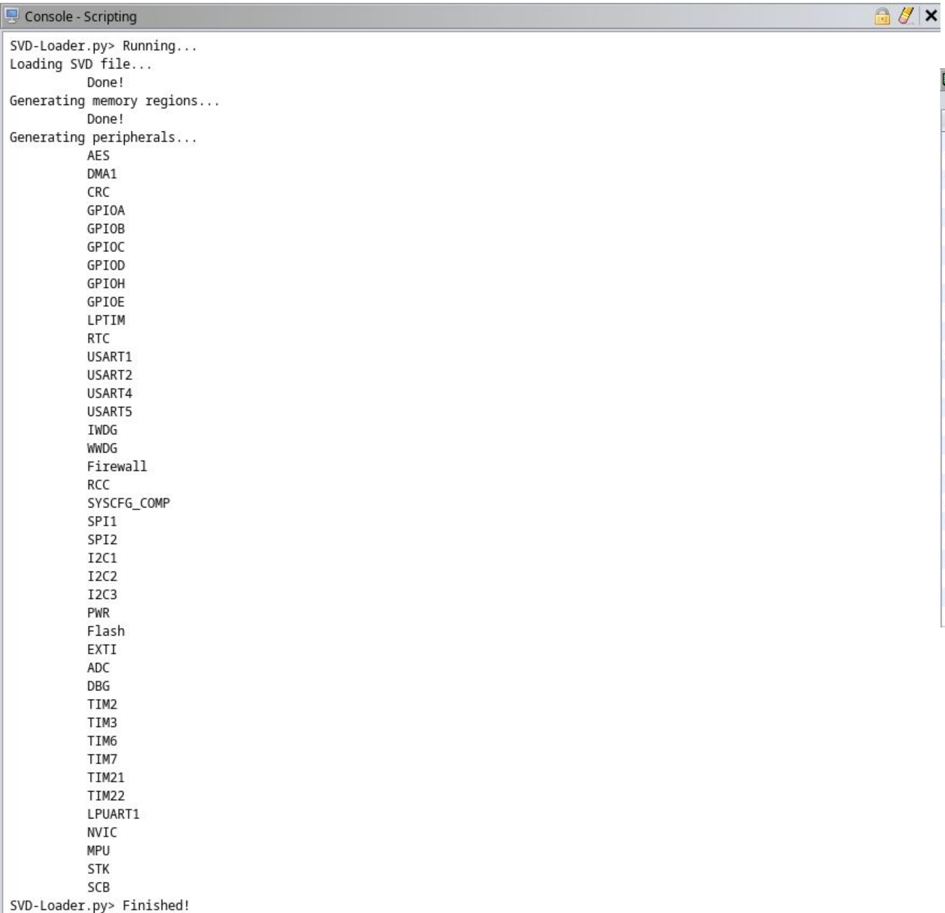
```
               STM32F301.svd   STM32F469.svd   STM32G050.svd     STM32G484xx.svd
               STM32F302.svd   STM32F730.svd   STM32G051.svd     STM32G491xx.svd
svd            STM32F303.svd   STM32F745.svd   STM32G061.svd     STM32G4A1xx.svd
.svd           STM32F373.svd   STM32F750.svd   STM32G070.svd     STM32GBK1CBT6.svd
.svd           STM32F3x4.svd   STM32F765.svd   STM32G071.svd     STM32H742x.svd
.svd           STM32F3x8.svd   STM32F7x.svd    STM32G07x.svd     STM32H743x.svd
.svd           STM32F401.svd   STM32F7x2.svd   STM32G081.svd     STM32H750x.svd
svd            STM32F405.svd   STM32F7x3.svd   STM32G0B0.svd     STM32H753x.svd
x.svd          STM32F407.svd   STM32F7x5.svd   STM32G0B1.svd     STM32H7A3x.svd
x.svd          STM32F410.svd   STM32F7x6.svd   STM32G0C1.svd     STM32H7B3x.svd
x.svd          STM32F411.svd   STM32F7x7.svd   STM32G431xx.svd   STM32H7x3.svd
x.svd          STM32F412.svd   STM32F7x8.svd   STM32G441xx.svd   STM32H7x5_CM4.svd
x.svd          STM32F413.svd   STM32F7x9.svd   STM32G471xx.svd   STM32H7x5_CM7.svd
x.svd          STM32F427.svd   STM32G030.svd   STM32G473xx.svd   STM32H7x7_CM4.svd
svd            STM32F429.svd   STM32G031.svd   STM32G474xx.svd   STM32H7x7_CM7.svd
svd            STM32F446.svd   STM32G041.svd   STM32G483xx.svd   STM32L0x1.svd
```

File name: STM32L0x1.svd

Type: All Files (*.*)

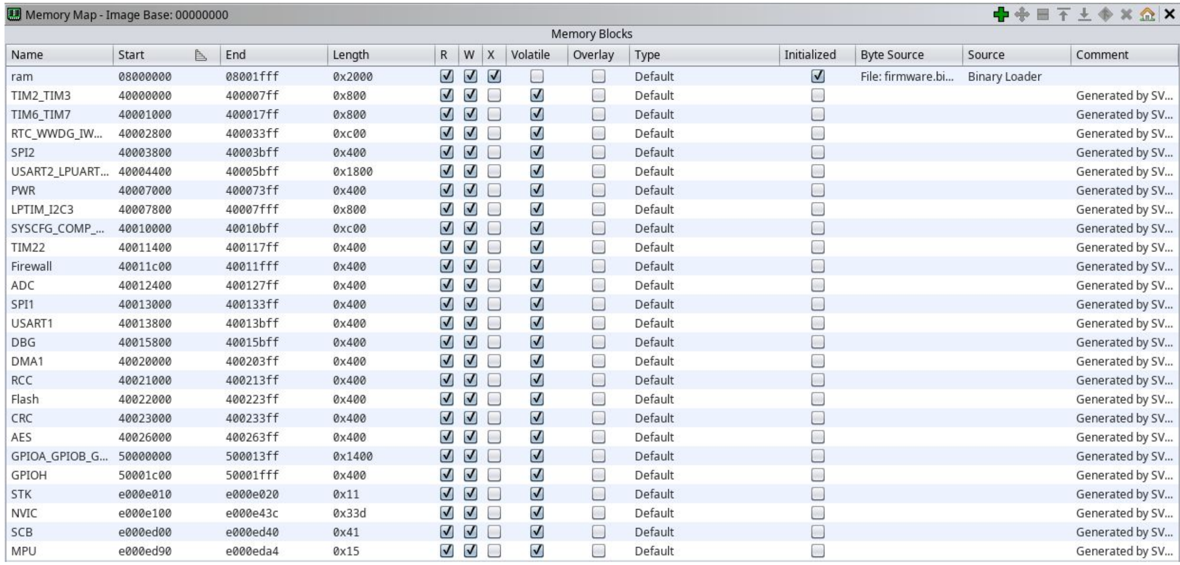
Load SVD File Cancel

SVD Loader output



Console - Scripting

```
SVD-Loader.py> Running...
Loading SVD file...
          Done!
Generating memory regions...
          Done!
Generating peripherals...
          AES
          DMA1
          CRC
          GPIOA
          GPIOB
          GPIOC
          GPIOD
          GPIOH
          GPIOE
          LPTIM
          RTC
          USART1
          USART2
          USART4
          USART5
          IWDG
          WWDG
          Firewall
          RCC
          SYSCFG_COMP
          SPI1
          SPI2
          I2C1
          I2C2
          I2C3
          PWR
          Flash
          EXTI
          ADC
          DBG
          TIM2
          TIM3
          TIM6
          TIM7
          TIM21
          TIM22
          LPUART1
          NVIC
          MPU
          STK
          SCB
SVD-Loader.py> Finished!
```



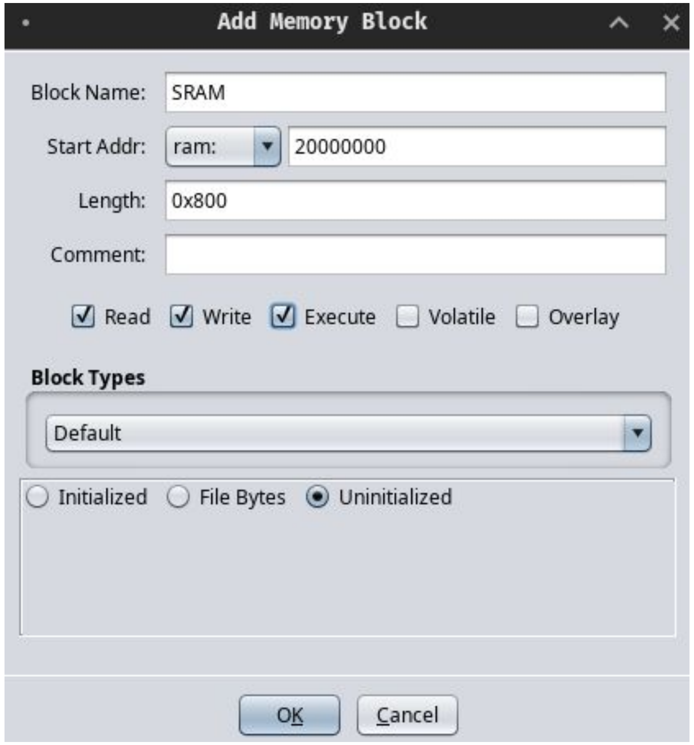
Memory Map - Image Base: 00000000

Memory Blocks

Name	Start	End	Length	R	W	X	Volatile	Overlay	Type	Initialized	Byte Source	Source	Comment
ram	08000000	08001fff	0x2000	✓	✓	✓	☐	☐	Default	✓	File: firmware.bi...	Binary Loader	
TIM2_TIM3	40000000	400007ff	0x800	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
TIM6_TIM7	40001000	400017ff	0x800	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
RTC_WWDG_IW...	40002800	400033ff	0xc00	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
SPI2	40003800	40003bff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
USART2_LPUART...	40004400	40005bff	0x1800	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
PWR	40007000	400073ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
LPTIM_I2C3	40007800	40007fff	0x800	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
SYSCFG_COMP_...	40010000	40010bff	0xc00	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
TIM22	40011400	400117ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
Firewall	40011c00	40011fff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
ADC	40012400	400127ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
SPI1	40013000	400133ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
USART1	40013800	40013bff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
DBG	40015800	40015bff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
DMA1	40020000	400203ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
RCC	40021000	400213ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
Flash	40022000	400223ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
CRC	40023000	400233ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
AES	40026000	400263ff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
GPIOA_GPIOB_G...	50000000	500013ff	0x1400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
GPIOH	50001c00	50001fff	0x400	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
STK	e000e010	e000e020	0x11	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
NVIC	e000e100	e000e43c	0x33d	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
SCB	e000ed00	e000ed40	0x41	✓	✓	☐	✓	☐	Default	☐			Generated by SV...
MPU	e000ed90	e000eda4	0x15	✓	✓	☐	✓	☐	Default	☐			Generated by SV...

SRAM ?

- If SRAM is missing, just add it manually



Block Name: SRAM

Start Addr: ram: 20000000

Length: 0x800

Comment:

[✓ Read] [✓ Write] [✓ Execute] Volatile Overlay

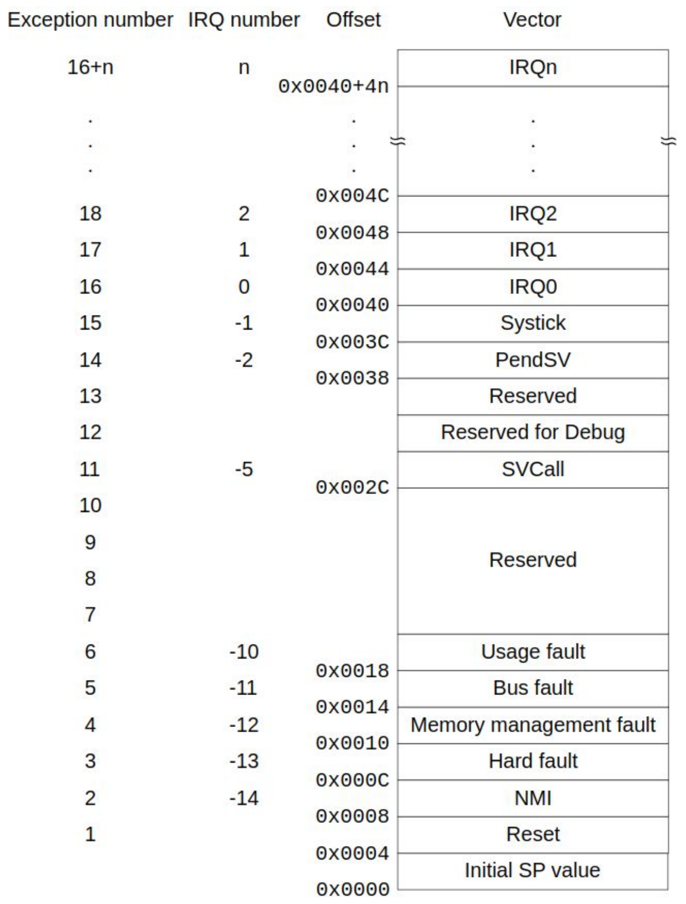
Block Types

Default

Initialized File Bytes [✓ Uninitialized]

Interrupt Vector Table (IVT)

- Contains the reset value of the stack pointer and the start address
- Exception vectors, for all exception handlers
- The least-significant bit of each vector is 1 (exception is in Thumb code)



Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
. . .		. . .	. . .
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

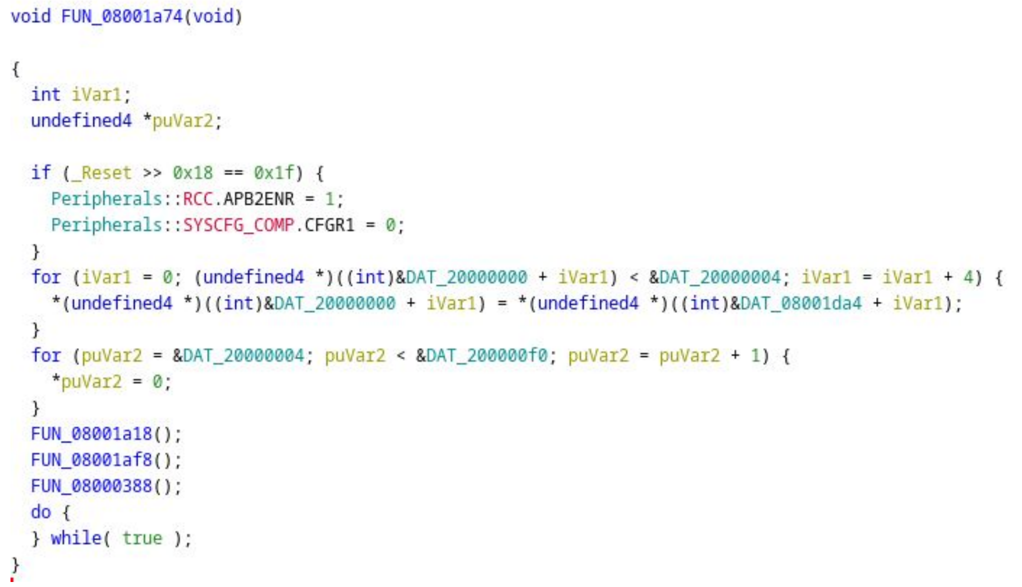
Reset Function

- Initialize some registers

- 0x08001a18 == SystemInit()
  - Functions for system and clock setup available in system_stm32l0xx.c

- 0x08001af8 == __libc_init_array()
  - GCC will put every constructor into an array in their own section of flash
  - Newlib will iterate through the array to call static constructors

- 0x08000388 == main()



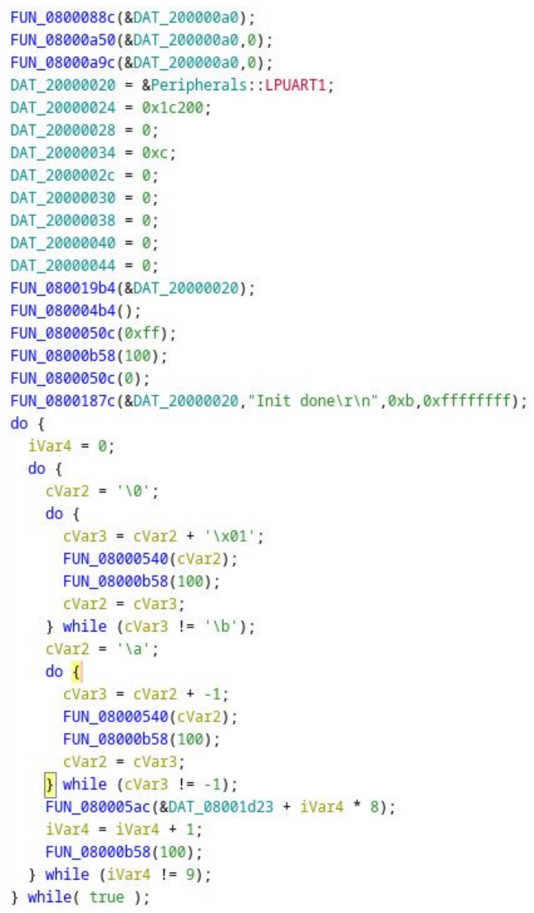
```c
void FUN_08001a74(void)

{
  int iVar1;
  undefined4 *puVar2;

  if (_Reset >> 0x18 == 0x1f) {
    Peripherals::RCC.APB2ENR = 1;
    Peripherals::SYSCFG_COMP.CFGR1 = 0;
  }
  for (iVar1 = 0; (undefined4 *)((int)&DAT_20000000 + iVar1) < &DAT_20000004; iVar1 = iVar1 + 4) {
    *(undefined4 *)((int)&DAT_20000000 + iVar1) = *(undefined4 *)((int)&DAT_08001da4 + iVar1);
  }
  for (puVar2 = &DAT_20000004; puVar2 < &DAT_200000f0; puVar2 = puVar2 + 1) {
    *puVar2 = 0;
  }
  FUN_08001a18();
  FUN_08001af8();
  FUN_08000388();
  do {
  } while( true );
}
```

RE tips

- Start from something known
- UART printings “Init done”
- LEDs blinking
- Enough tips…get the flag !



```
FUN_0800088c(&DAT_200000a0);
FUN_08000a50(&DAT_200000a0,0);
FUN_08000a9c(&DAT_200000a0,0);
DAT_20000020 = &Peripherals::LPUART1;
DAT_20000024 = 0x1c200;
DAT_20000028 = 0;
DAT_20000034 = 0xc;
DAT_2000002c = 0;
DAT_20000030 = 0;
DAT_20000038 = 0;
DAT_20000040 = 0;
DAT_20000044 = 0;
FUN_080019b4(&DAT_20000020);
FUN_080004b4();
FUN_0800050c(0xff);
FUN_08000b58(100);
FUN_0800050c(0);
FUN_0800187c(&DAT_20000020,"Init done\r\n",0xb,0xffffffff);
do {
  iVar4 = 0;
  do {
    cVar2 = '\0';
    do {
      cVar3 = cVar2 + '\x01';
      FUN_08000540(cVar2);
      FUN_08000b58(100);
      cVar2 = cVar3;
    } while (cVar3 != '\b');
    cVar2 = '\a';
    do {
      cVar3 = cVar2 + -1;
      FUN_08000540(cVar2);
      FUN_08000b58(100);
      cVar2 = cVar3;
    } while (cVar3 != -1);
    FUN_080005ac(&DAT_08001d23 + iVar4 * 8);
    iVar4 = iVar4 + 1;
    FUN_08000b58(100);
  } while (iVar4 != 9);
} while( true );
```

Congratulations !!!

- You survived the workshop !
- Hopefully learn something new about electronics
    - Low-level SWD interactions with an ARM device
    - Firmware extraction via the debug interface
    - Firmware loading & analysis with Ghidra

THANK YOU !