Connected Car Hacking

Introduction to ECU reverse engineering

hardwear.io

Quarkslab

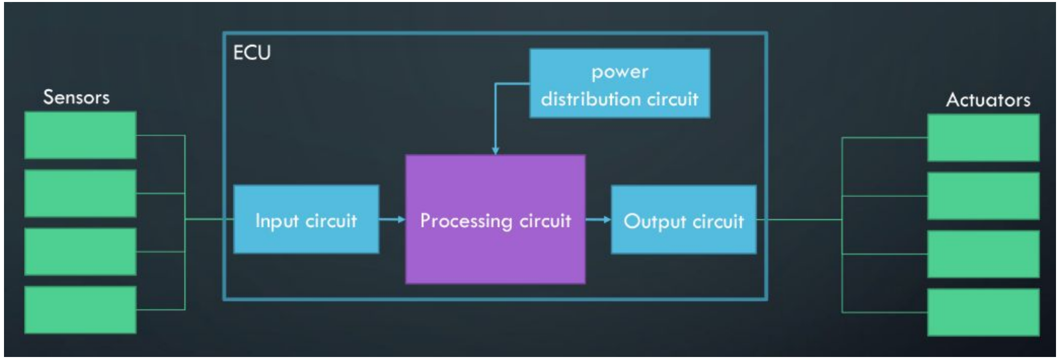
Introduction to ECU reverse engineering

- Analysing **firmware** will help to understand how an **ECU** works, finding **vulnerabilities** , stored **secrets/credentials** or hidden **functions**
- It's a very time-consuming task, as you will face a lot of different **architectures**
- **Reverse engineering** can be **static**, using **SRE** (Software Reverse Engineering) frameworks like IDA/Ghidra/Binary Ninja… or **dynamically** using emulation like Qemu or by using available debug ports
- **Automate** this task as much as you can

ECU inner workings

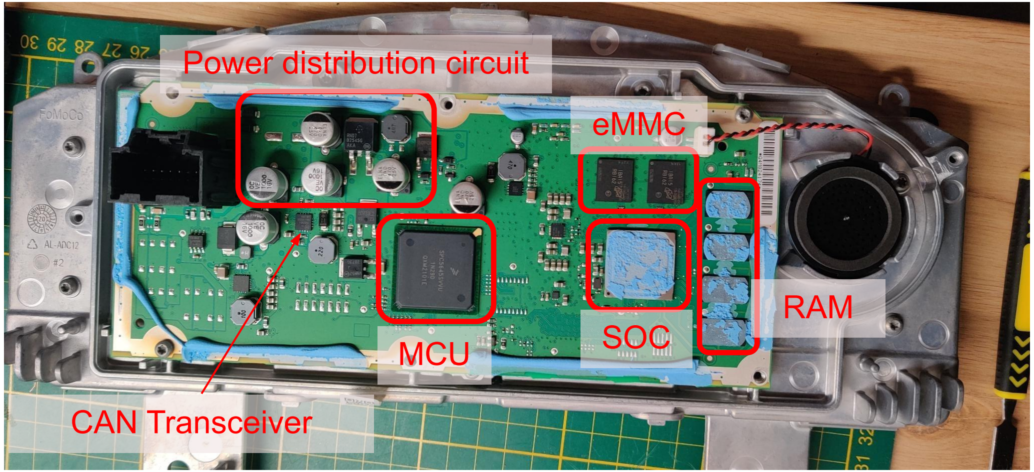
- An **ECU** will have one or multiple **microcontrollers (µC)** or **system-on-chip** (**SOC**), each running its own **firmware/OS**

- **µC/SOC** are powered via a **Power Distribution Circuit** and handle multiples **sensors** and **actuators** through input/output chips, like a CAN transceiver for example



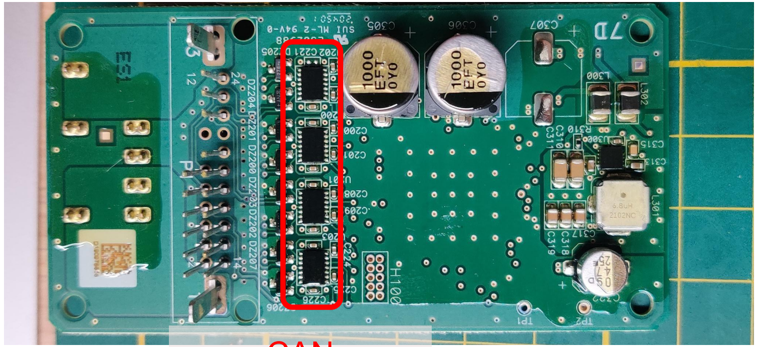
ECU internal example #1 - Modern Instrument Cluster

- **MCU:** PowerPC SPC58 handles the CAN communication and most of the GPIO
- **µC/SOC**: IMX.6 SOC running QNX manage the display. The OS is stored on one of the two eMMC

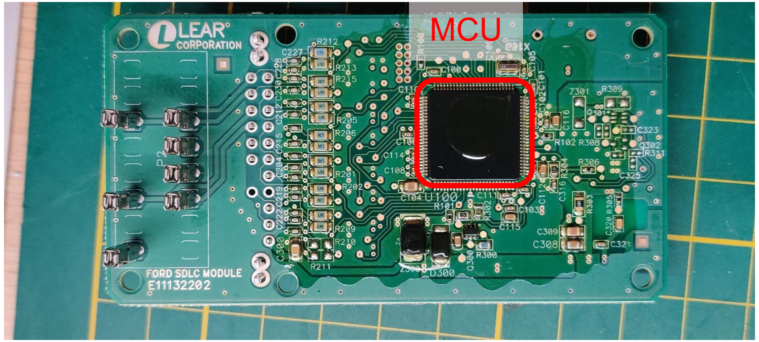


ECU internal example #1 - Gateway

- This ECU only have one MCU (PowerPC SPC58)
- **µC/SOC**: As it is connected to all CAN buses, there are several **CAN transceivers**

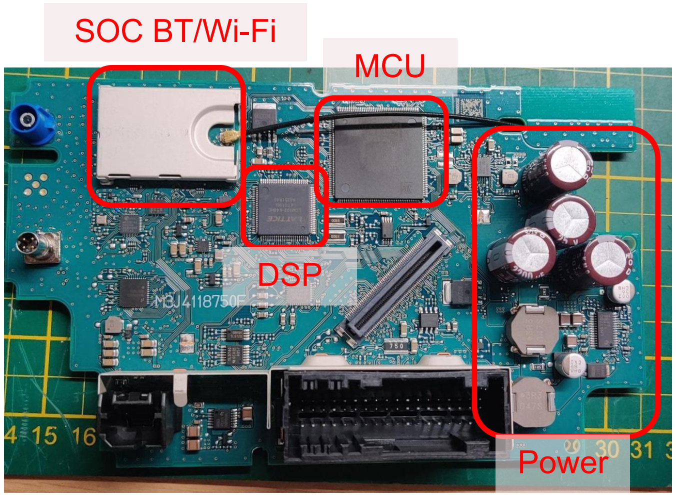


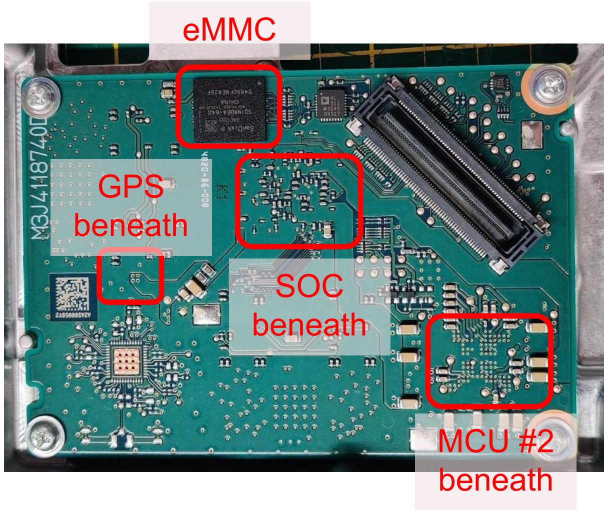
CAN
Transceivers



ECU internal example #3 - IVI

- Infotainment unit is one of the most complex **ECUs**, having several **SOCs** for the OS and the various radio protocol (Bluetooth, Wi-Fi…)





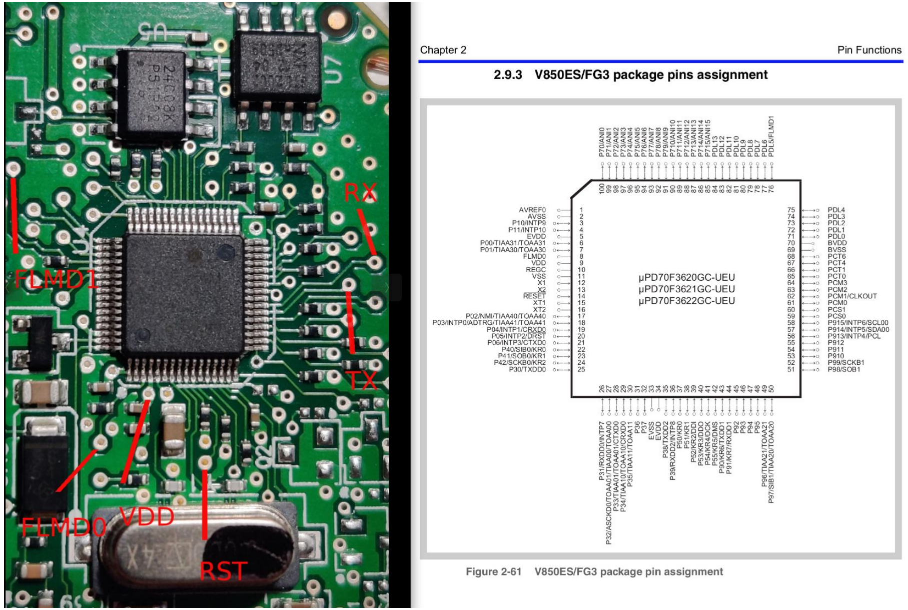
Step #1: getting the firmware

- An **ECU** will have one or multiple **firmwares**, depending on how many **µC/SOC** are inside
- For **SOC**, the **firmware/OS image** could be stored inside an **external Flash memory**
- Each **µC/SOC** will have a **debug port** on which you may read its firmware, however such ports are often **disabled/secured**
- For **PCM/ABS/BCM ECUs,** you could find firmware images online on **chip-tuning/reprogramming forums**. However, dumps are sometime **incomplete as** only a part of the memory was read
- Having access to a **diagnostic tool** or looking at **manufacturer website** could provide firmware through **updates**, but they could be **encrypted**

Getting the firmware: debug access port

- Depending on the **µC**, you may access to the debug port using **JTAG**, **SWD** or **proprietary tools**

- Most of the time, **µC** memory is **read-protected**, to bypass it you'll have to try different attacks like **Fault Injection** or **Cold Boot Attack**



TinyECU GPSM V850 µC debug ports

Getting the firmware: Flash memory

- For **SOC**, used in **IVIs** or **TCUs**, it's common to find the **firmware/OS image** on an external **Flash memory**
- **EMMC** memory could sometime be dumped without removing the chip, using an SD-Card reader connected to pins **CLK, CMD, DAT[0-3]** (example blogpost)

Left: eMMC Flash (BGA) - Right: NAND Flash (TSOP 48)

Getting the firmware: UDS way

- It always worth a shot to look for **UDS** services such as **Read Memory By Address** or even **Request Upload** (less likely), trying every available **Diagnostic Session** on an **ECU**

- Compare the **size** of the **extracted data** to the **chip datasheet**, you'll find which kind of memory you had access to (RAM, Flash memory …)

- If you have access to a **Diagnostic Tool**, you can sniff a **firmware upgrade.** However, data could be **incomplete** (ex: calibration update only)

ISO 22901: ODX/PDX

- **Firmware updates** are often provided as one or several **ODX** (**O**pen **D**iagnostic Data E**x**change) files
- These are **XML** document describing **supported services**, data to update on specific **memory location**…
- **ODX** files and binary related data could be provided as a **PDX** (Packaged ODX), which is a zip file

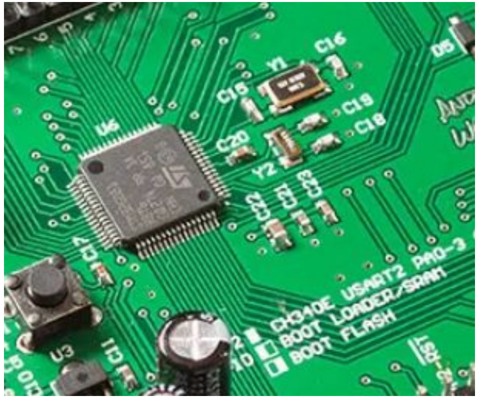
Lab 1 - Firmware extraction

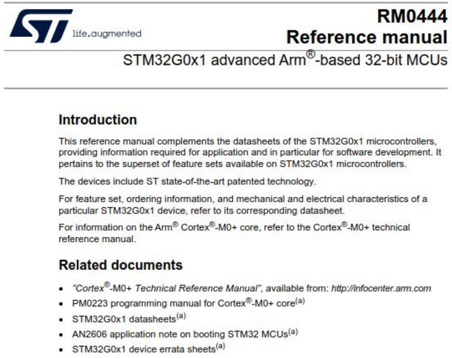
Goals

- We got an odx file from a tuning forum 95B909144K_1902_BP.odx-f, have a look at it
- Look for specific mnemonics, like '**DATABLOCK**', '**FLASHDATA**' or '**ENCRYPT-COMPRESS-METHOD**'
- We managed to get a **PDX** file related to our **IVI**, look at the files it contains
- We also captured the **firmware upgrade** resulting from the **PDX** on the CAN bus, analyse the different frames and compare it with the **PDX** content

Reverse engineering the firmware: architecture

- To reverse engineer a **firmware**, we first have to know for which **architecture** it has been compiled
- Look for the microcontroller's **datasheet** to get such information
- It is common that **automotive µC** datasheet are under **NDA**, so search for **approaching references**





life.augmented

RM0444
Reference manual

STM32G0x1 advanced Arm®-based 32-bit MCUs

Introduction

This reference manual complements the datasheets of the STM32G0x1 microcontrollers, providing information required for application and in particular for software development. It pertains to the superset of feature sets available on STM32G0x1 microcontrollers.

The devices include ST state-of-the-art patented technology.

For feature set, ordering information, and mechanical and electrical characteristics of a particular STM32G0x1 device, refer to its corresponding datasheet.

For information on the Arm® Cortex®-M0+ core, refer to the Cortex®-M0+ technical reference manual.

Related documents

- *"Cortex®-M0+ Technical Reference Manual"*, available from: *http://infocenter.arm.com*
- PM0223 programming manual for Cortex®-M0+ core[a]
- STM32G0x1 datasheets[a]
- AN2606 application note on booting STM32 MCUs[a]
- STM32G0x1 device errata sheets[a]

Reverse engineering the firmware: endianness

- Once you know the **architecture**, you may need to find out the **endianness** used
- Look for the microcontroller's **datasheet** to get such information
- Some architecture could use both endianness, using **binbloom** will help you find out which one is correct: `binbloom -a [architecture bits] file`

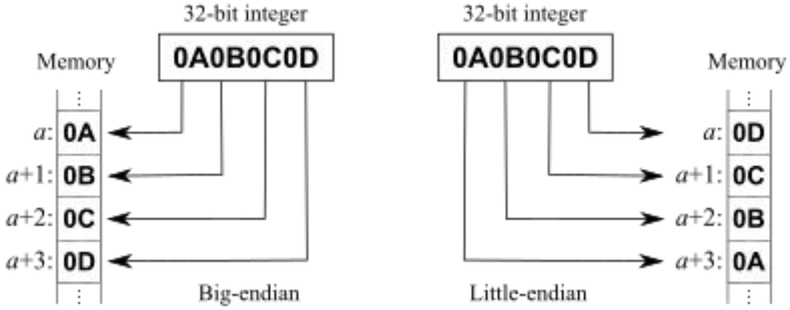
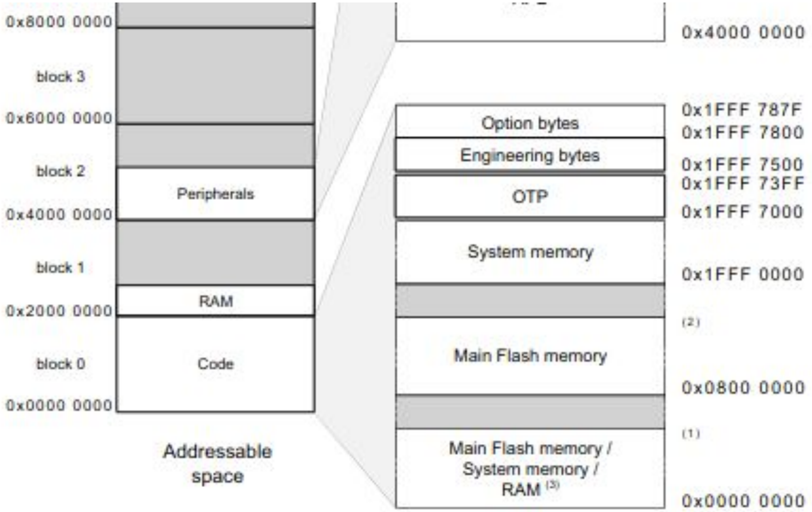


Illustration: [link]

Reverse engineering the firmware: base address

- The **base address** is the address in the **memory** where the firmware is loaded from
- Knowing this address will helps the **SRE** tool you're using to find **cross-references** to pointers, functions …
- Look at the **datasheet** to find it



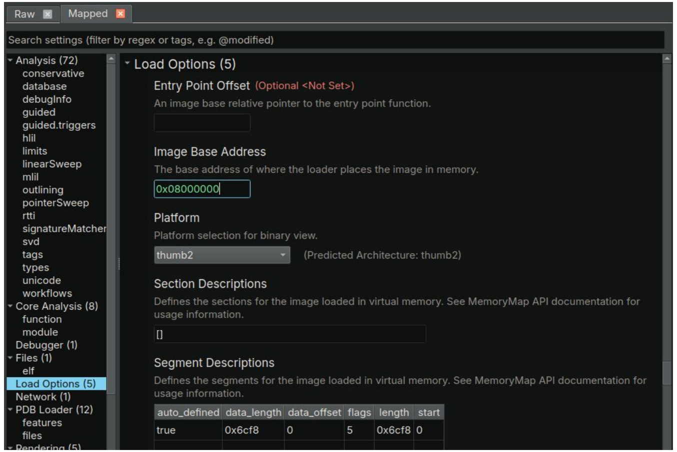
Lab 2 - Firmware analysis

Goals

- To test **binbloom**, we provide a **2013 Polo ECU** firmware. Try to identify endianness, base address, UDS database detection (it is based on a Tricore TC1766)
- For all other RE lab, we will work on the **TinyECU** firmware, available in the previous **PDX**
- Look at the **provided manual** to find the **endianness** and the **base address**

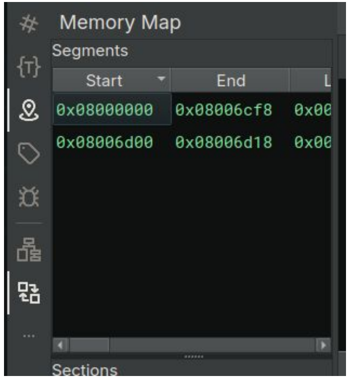
Reverse engineering the firmware: using Binary Ninja

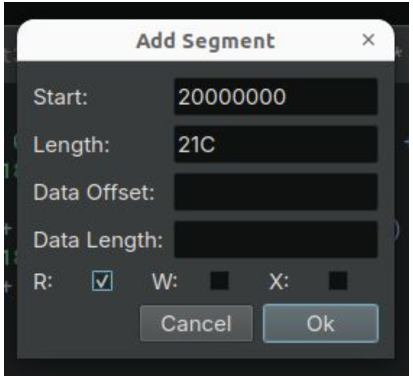
- We will use **Binary Ninja** as our **SRE** tool to introduce some automotive-specific reverse engineering techniques on the **TinyECU firmware**
- Under the menu **File**, choose **Open** and select the 'IVI_v2.1_app.bin'
- The **architecture** will be correctly detected and set to “**thumb2**”
- The base address needs to be set at **0x08000000**

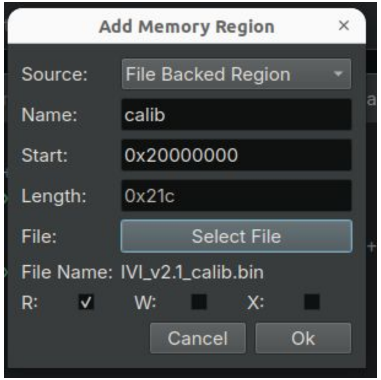


Reverse engineering the firmware: mapping the memory

- Our **PDX** provides two binary, we need to load the second one
- Using the **Memory Map** window, right click to select the **"Add Segment"** menu, set the correct start address and length.
- Once created, with a right-click on the new segment, choose "**Add Memory Region**" to load the second binary

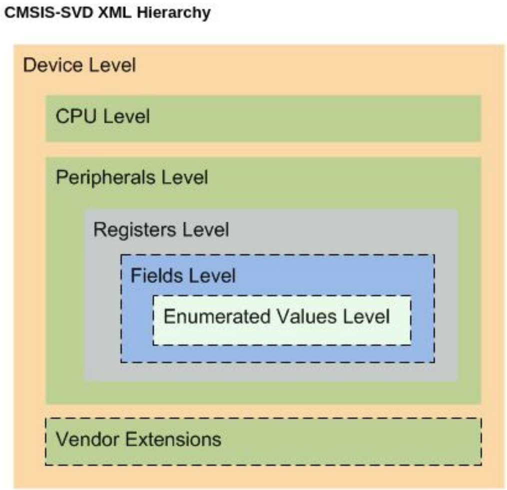






Mapping the registers

- Mapping the various registers is also helpful, but it is a time-consuming task
- **Plugins** exists to automatise such task, like **Load SVD File**. Use it to automatically map main registers using the '**STM32G0B1.svd**' file



Reverse engineering basis: CPU registers & instructions

- The CPU executes a set of **instructions**
- Each instruction is defined by an **opcode**, a hexadecimal value
- To store data, the CPU uses the **memory** or **registers** (a0-a15, d0-d15)
- As it is a **32 bits microcontroller**, data can be stored as a :
  - **Byte**, coded “b” (8 bits)
  - **Half-word**, coded “h” (16 bits)
  - **Word**, coded “w” (32 bits)
- By default, data is **signed**, if the “u” prefix is present, it means that data is **unsigned**

Instruction examples

- ldr r3, [DAT_00007670]
  Load in register r3 the byte value stored at memory address 0x00007670

- ldrh r5, [r4, #0]
  Load in register r5 the two bytes value stored at memory address present in register r4, with a 0 offset

- beq LAB_0000dbfa
  Branch to address 0x0000dbfa if previous comparison (cmp) is equal

- movw r1, #0x726
  Move 4 bytes value 0x00000726 in register r1

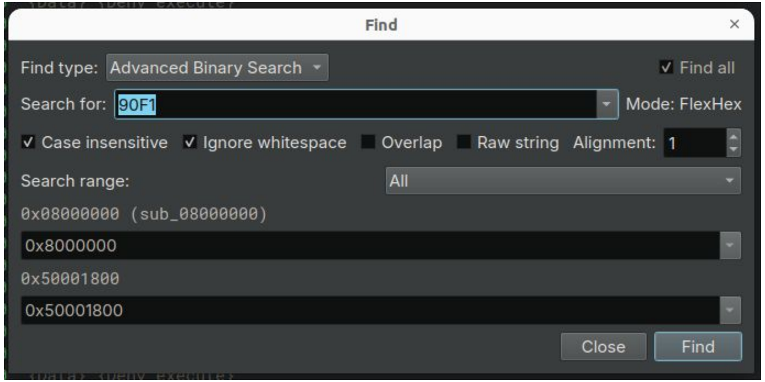
- Full instruction set can be found here:
  https://developer.arm.com/documentation/ddi0403/d/Application-Level-Architecture/The-ARMv7-M-Instruction-Set/About-the-instruction-set

Grap view & pseudo code

- A **microcontroller** has a set of **instructions** and **CPU registers**
- When analysing a **function**, do not try to understand each instruction, it is a waste of time
- Look at **pseudocode** or **graph view** instead to have a **quick understanding** of what is executed
- Do not try to analyse each function, start from a **target interrupt** or **peripheral** register instead
- Look also for some **specific values**, like UDS Services, CAN identifier, DID …

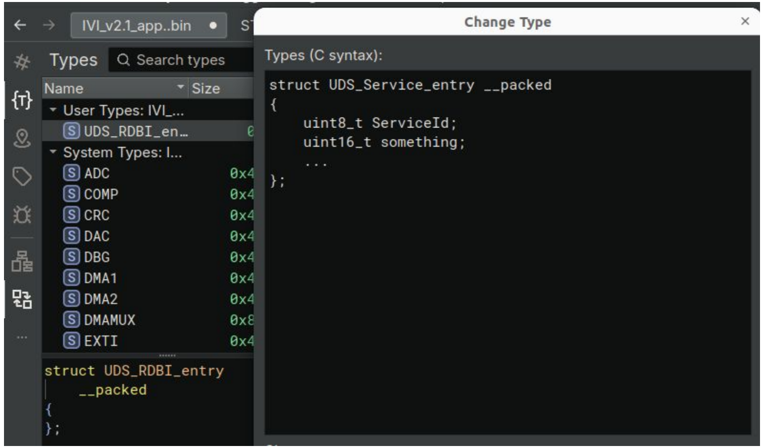
UDS reverse engineering - DID database

- **DIDs** are a good hint to find references to their functions and associated data
- It is common that **DIDs** are stored as an array of struct, including theirs DID value and a pointer to a function managing it
- Search for value **0xF190 (VIN DID)** using the appropriate endianness and analyse if you see a pattern
- Try to create the correct struct and apply it to each **DID**



UDS database

- The same applies for the **UDS Services**: they are commonly available in a DB, which is an array of struct
- Using one of the DID function you find, try to cross-ref to the DID handler and search for the start address in the memory
- **Hint:** as the arch is in Thumb mode, it's normal to find pointer you'll need to add ` | 1` to the address to find the correct pointer
- Look around to identify the different supported **UDS Services**, try to guess the correct structure and label all the functions



UDS Negative Response Code

- Remember, **UDS** protocol is verbose
- Looking at constants matching **UDS NegativeResponseCode** can help you to confirm the function you are analysing is a **UDS** one

```
if (cVar2 != '\x03') {
  if (cVar2 != '\x04') {
    if (cVar2 == '\x05') {
      iVar4 = FUN_a0154b90((undefined2 *)(param_1 + 0x54));
      if (iVar4 == 0) {
        iVar4 = return_0();
        if (iVar4 == 0) {
          return 0x2f;
        }
                    /* UDS NRC GeneralReject */
        *param_3 = 0x10;
        return 1;
      }
      if (iVar4 == 1) {
                    /* UDS NRC ConditionsNonCorrect */
        uVar5 = 0x22;
      }
      else {
        if (iVar4 == 10) {
          return 0xa;
        }
        uVar5 = GeneralReject;
      }
    }
    else {
      uVar5 = GeneralReject;
    }
    *param_3 = uVar5;
    return 1;
  }
```

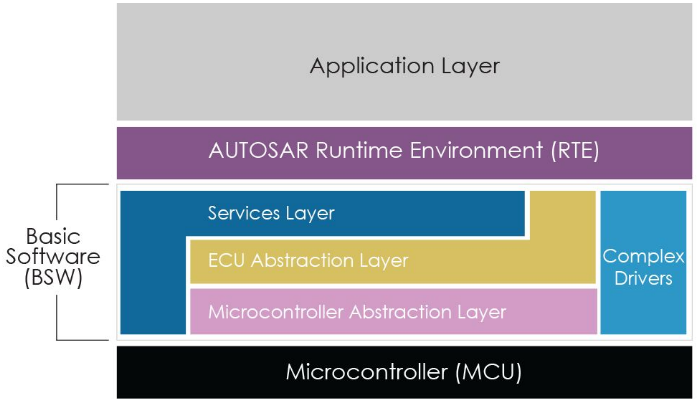
Lab 3 - Firmware analysis

Goals

- Look for known **DID** value and try to find the **DID database** and map **DID_Read** and **DID_Write** structures
- Label all the **DID functions**
- Find the **UDS database**, set the correct struct and label all **UDS functions**

Using AutoSAR specifications

- **AUTOSAR** is a “worldwide development partnership of vehicle manufacturers”
- It provides **documentation** and **standards** for the automotive industry
- It also provides a platform, the **standardized ECU software architecture**



AUTOSAR Architecture

Illustration: [link]

AutoSAR

- Most of the ECU are designed using an **AutoSAR** framework
- Studying **AutoSAR protocol specifications** helps to understand or identifying standard functions or variables of a firmware ECU

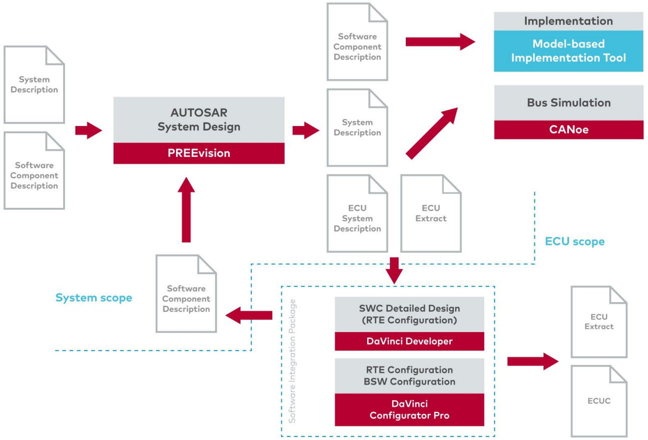


Illustration: [link]

AutoSAR: DCM module

- UDS functions are defined in the **Diagnostic Communication Manager**
- The **API specification chapter** in each specification documents define base **type**, **functions** or **enums**
- Provided functions arguments, return type and operating mode is helpful to identify AutoSAR defined functions during reverse engineering process

8.9.1 <Module>_<DiagnosticService>

[SWS_Dcm_00763] ⌈

Service Name	<Module>_<DiagnosticService>
Syntax	`Std_ReturnType <Module>_<DiagnosticService> ( Dcm_ExtendedOpStatusType OpStatus, Dcm_MsgContextType* pMsgContext, Dcm_NegativeResponseCodeType* ErrorCode )`
Service ID [hex]	0x32
Sync/Async	Asynchronous
Reentrancy	Reentrant

▽

8.9.2 <Module>_<DiagnosticService>_<SubService>

[SWS_Dcm_00764] ⌈

Service Name	<Module>_<DiagnosticService>_<SubService>	
Syntax	`Std_ReturnType <Module>_<DiagnosticService>_<SubService> ( Dcm_ExtendedOpStatusType OpStatus, Dcm_MsgContextType* pMsgContext, Dcm_NegativeResponseCodeType* ErrorCode )`	
Service ID [hex]	0x33	
Sync/Async	Asynchronous	
Reentrancy	Reentrant	
Parameters (in)	OpStatus	DCM_INITIAL DCM_PENDING DCM_CANCEL DCM_FORCE_RCRRP_OK DCM_POS_RESPONSE_SENT DCM_POS_RESPONSE_FAILED DCM_NEG_RESPONSE_SENT DCM_NEG_RESPONSE_FAILED
Parameters (inout)	pMsgContext	Message-related information for one diagnostic protocol identifier. The pointers in pMsgContext shall point behind the SID.
Parameters (out)	ErrorCode	If the operation <Module>_<DiagnosticService>_<SubService> returns value E_NOT_OK, the Dcm module shall send a negative response with NRC code equal to the parameter ErrorCode parameter value.
Return value	Std_ReturnType	E_OK: Request was successful E_NOT_OK: Request was not successful DCM_E_PENDING: Request is not yet finished DCM_E_FORCE_RCRRP: Application requests the transmission of a response Response Pending (NRC 0x78)

▽

Illustration: [link]

Lab 4 - AutoSAR calls

Goals

- Map the **ErrorCode** pointer and try to check various functions using it

AutoSAR: module interactions

- To keep **hardware dependent** functions separate from **independent ones**, a “simple” task could involve several modules, like sending a CAN message

- This improves **portability**, **reusability** and **scalability** across different ECUs and automotive networks

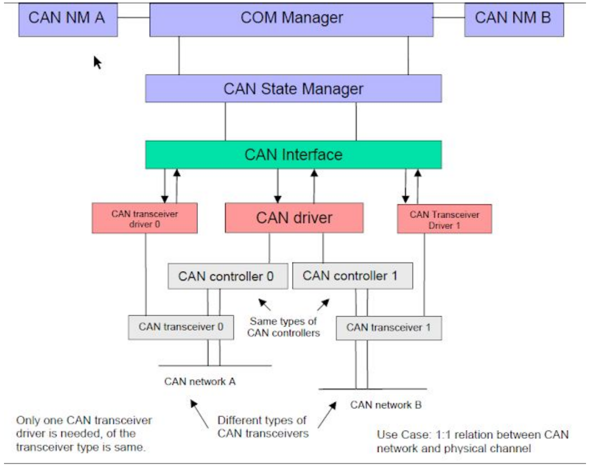


Illustration: [link]

AutoSAR: locate useful functions

- Knowing commonly used **AutoSAR** functions may help to understand part of the firmware
- **AutoSAR DET_ReportError** is a great candidate, as it is generally used to trace errors
- It also contains useful arguments: **ModuleID** and **ApiID**
- This function has a **lot of calls** with **static arguments** for the **ModuleID** and **ApiID**, allowing a quick identification

Illustration: [link]

8.1.3.2 Det_ReportError

[SWS_Det_00009] ⌈

Service Name	Det_ReportError
Syntax	Std_ReturnType Det_ReportError (uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId)

```
void FUN_000c68da(void)

{
  undefined4 *puVar1;
  int unaff_gp;
  int unaff_tp;
  uint uVar2;
  undefined4 *puVar3;

  if ((&DAT_ffff8004)[unaff_gp] == '\0') {
    DET_ReportError(0x65,0,7,0xf);
  }
  else {
```

Finding Module ID and Api ID

- **AUTOSAR** provides a list of all base module for the **Basic Software (BSW)** layer
- Each module has a **16-bits ID** and a **dedicated Specification document** where all base Service ID are listed, which correspond to the **ApiID**
- Using this information will allow us to quickly identify **AutoSAR** functions inside our firmware

Module short name	Module abbreviation (API service prefix)	Module ID (uint16)	Specification document
GPT Driver	Gpt	100	AUTOSAR_SWS_GPTDriver.pdf
MCU Driver	Mcu	101	AUTOSAR_SWS_MCUDriver.pdf
Watchdog Driver	Wdg	102	AUTOSAR_SWS_WatchdogDriver.pdf

8.3.8 Mcu_PerformReset

[SWS_Mcu_00160]⌈

Service Name	Mcu_PerformReset
Syntax	`void Mcu_PerformReset ( void )`
Service ID [hex]	0x07

Illustration: [link] & [link]

Identifying AutoSAR functions

Illustration: [link]

- If we have a match regarding **AutoSAR specifications** for a **Module ID** and a **Service ID**, we can compare arguments and return value to identify the target function

```
void EthIf_Init(int param_1)
{
  int iVar1;

  iVar1 = 0;
  if (DAT_7000415c == '\0') {
    if (param_1 == 0) {
      iVar1 = 5;
    }
    else {
      DAT_701008f8 = param_1;
      FUN_a0163ad8();
      DAT_7000415c = '\x01';
    }
  }
  else {
    iVar1 = 7;
  }
  if (iVar1 != 0) {
    DET_ReportError(0x41,0,1,iVar1);
    return;
  }
  return;
}
```

Module short name	Module abbreviation (API service prefix)	Module ID (uint16)
LIN Interface	LinIf	062
LIN Transceiver Driver	LinTrcv	064
Ethernet Interface	EthIf	065

[SWS_EthIf_00024] ⌈

Service Name	EthIf_Init	
Syntax	`void EthIf_Init ( const EthIf_ConfigType* CfgPtr )`	
Service ID [hex]	0x01	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	CfgPtr	Points to the implementation specific structure
Parameters (inout)	None	
Parameters (out)	None	
Return value	None	
Description	Initializes the Ethernet Interface	
Available via	EthIf.h	

Lab 5 - Finding AUTOSAR Det_ReportError

Goals

- Look at identified **UDS** functions if you see a pattern that looks like **Det_ReportError**
- Using cross-references find various calls
- Compare arguments of functions of interest with the **AutoSAR BSW Module List**: https://www.autosar.org/fileadmin/standards/R19-11/CP/AUTOSAR_TR_BSWModuleList.pdf
- Once found, try to locate functions tied to the **PDU Router** module

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: www.quarkslab.com

@quarkslab

Quarkslab