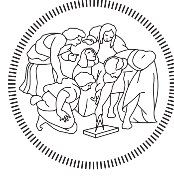
Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Master Degree in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria



POLITECNICO
MILANO 1863

CanPass: An Extensible Framework for Bypassing CAN Peripherals on Unmodified Microcontrollers

Advisor: Prof. Stefano Zanero
Supervisor: Stefano Longari, Ph.D.
Co-Supervisor: Mario Polino, Ph.D.

Master Dissertation of:
Alvise de Faveri Tron (Matr. 920882)

Academic Year 2020-2021

'You cannot pass!' he said.
With a bound the Balrog leaped full upon
the bridge. Its whip whirled and hissed.

Abstract

The Controller Area Network (CAN) is a serial communication protocol that has become a de-facto standard in the automotive industry. The basics of the protocol have been developed in the 1980s and, similarly to other protocols from that era of computing, it has been designed with little concern for security.

In particular, the CAN protocol has been discovered to be vulnerable to *bit injection* attacks, through which an adversary can disable devices connected on the bus by injecting bits at specific moments. Such attacks can be used to target safety subsystems or more generically to disrupt the normal behavior of a vehicle, and are stealthier than other types of attacks. Bit injection attacks are possible under the assumption that the attacker can manipulate the bus at a bit level.

In reality, manipulating a high-speed bus at such a low level is a challenging task to carry out on resource-constrained microcontrollers, such as those found inside automotive networks: current techniques either rely on external hardware or require very precise timings. Consequently, such attacks are generally considered feasible only in scenarios where the attacker has physical access to the bus, which makes them considerably less harmful, since physical attacks scale poorly and expose the attacker to higher risks.

In this work, we overcome these limitations by proposing a new set of techniques to manipulate the CAN data link layer, exploiting the fact that, in modern microcontrollers, many different peripherals are generally attached to the same physical pins. We demonstrate that, through these techniques, we are able to reliably read and inject bits on the CAN bus from the software layer of unmodified microcontrollers. Our approach is more reliable than existing ones and does not require any physical access to targeted device, which means it can be executed on a remotely compromised device to perform bit injection attacks.

Given the generality of the underlying mechanism, we have collected such techniques into a multi-platform, extensible framework called *CANPass*, with the aim of easing the development of such techniques and facilitating the research on this topic.

Sommario

Il protocollo CAN (Controller Area Network) è un protocollo di comunicazione seriale nato negli anni '80 per soddisfare le crescenti esigenze del settore automobilistico, ed è diventato lo standard di fatto per interconnettere centraline elettroniche all'interno di ogni tipo di veicoli. Nel tempo il suo utilizzo è cresciuto fino ad arrivare al di fuori di questo settore: è infatti oggi utilizzato anche all'interno dei sistemi elettronici di navi, di impianti di controllo industriale e persino nel settore aerospaziale.

Il motivo del suo successo risiede nella capacità di realizzare collegamenti tra un grande numero di dispositivi, anche su distanze relativamente ampie, usando solo due segnali e mantenendo un'alta immunità al rumore elettromagnetico.

Tuttavia, come molti protocolli di comunicazione progettati in quel periodo, non è stato pensato per essere al sicuro da attacchi informatici.

Il CAN bus è infatti un protocollo *broadcast*, cioè ogni pacchetto inviato viene ricevuto indiscriminatamente da tutti i dispositivi connessi. Questo aspetto, unito all'assenza di cifratura dei messaggi e di qualsiasi misura di autenticazione, rende il protocollo CAN vulnerabile a una serie di attacchi informatici che mettono in pericolo, di conseguenza, i sistemi che ne fanno uso.

Una delle prime dimostrazioni di questo problema è stata data dai ricercatori Charlie Miller e Chris Valasek, che, nel 2015, sono riusciti a spegnere remotamente il motore di una Jeep Cherokee che viaggiava a piena velocità su un'autostrada americana. Questo attacco è stato reso possibile proprio dall'assenza di sicurezza del protocollo CAN, che interconnette tutta l'elettronica interna delle automobili ed è accessibile tramite una semplice porta diagnostica, chiamata porta OBD, presente per legge in tutte le macchine in circolazione negli Stati Uniti.

Da allora, la ricerca sulla sicurezza informatica delle moderne automobili si è sviluppata enormemente, mettendone in luce l'estesa superficie di attacco e le molte vulnerabilità.

Un particolare tipo di attacchi che è possibile condurre sul protocollo CAN sono gli attacchi di tipo *bit-injection*, che sfruttano regole di basso livello del protocollo

per produrre errori di comunicazione difficilmente distinguibili dai guasti reali. La possibilità di iniettare errori sul bus in momenti arbitrari della trasmissione di altre centraline dà la possibilità a un attaccante di disturbare la comunicazione sul bus e, in certi casi, addirittura selettivamente interrompere le operazioni di uno specifico dispositivo collegato, sfruttando i meccanismi di gestione degli errori definiti dal protocollo stesso.

Per condurre questi attacchi, l'attaccante deve però essere in grado di manipolare i segnali del bus a livello di singoli bit, e in modo molto preciso. Se, infatti, negli ordinari microcontrollori la comunicazione è gestita da un dispositivo separato, chiamato *CAN controller*, che si occupa di rispettare le regole imposte dal protocollo, nel caso degli attacchi *bit-injection* deve essere possibile violare alcune di queste regole. Per questo motivo, l'esecuzione di questi attacchi su dei normali microcontrollori richiede di aggirare il CAN controller per accedere direttamente ai livelli logici del bus via software.

I metodi utilizzati finora per fare ciò soffrono di pesanti limitazioni: o necessitano di hardware esterno per essere utilizzati, oppure sono scarsamente affidabili se eseguiti su dispositivi con potenza limitata. Questo, di fatto, limita gli scenari in cui tali attacchi sono considerati possibili e, in particolare, riduce la possibilità che questi attacchi vengano eseguiti usando l'hardware già presente nelle centraline. Conseguentemente, questi attacchi sono al momento considerati improbabili in uno scenario di *compromissione remota*, ovvero una situazione in cui un attaccante abbia la possibilità di modificare remotamente il software, ma non l'hardware, di una centralina connessa al CAN bus di un veicolo.

Questa tesi si pone l'obiettivo di superare queste limitazioni. In particolare, proponiamo una nuova serie di tecniche che sfruttano il fatto che, sui microcontrollori moderni, più periferiche hardware sono generalmente collegate agli stessi segnali fisici. Se questo conflitto tra più periferiche si verifica sui pin dedicati al CAN bus, le periferiche connesse ad essi possono essere usate per inviare e ricevere bit sul CAN bus in momenti arbitrari, bypassando completamente il CAN controller.

Il nostro approccio risulta essere più affidabile dei metodi correnti usati per iniettare e leggere singoli bit sul CAN bus e può essere implementato totalmente via software anche su microcontrollori poco potenti, il che rende possibili attacchi di basso livello al CAN bus anche in scenari di compromissione remota.

Data la varietà di soluzioni che derivano da questo approccio, abbiamo riunito le tecniche proposte in un framework estendibile e multi-piattaforma, con l'intento di facilitarne l'utilizzo e incentivare la ricerca in questo ambito.

Contents

1 Introduction **1**

1.1 CANBus security . . . 1

1.2 Link-Layer Attacks . . . 2

1.3 Our Contribution . . . 3

2 Background **5**

2.1 The CAN Protocol . . . 5

2.1.1 Protocol Stack . . . 6

2.1.2 CAN Nodes . . . 7

2.1.3 Physical Layer . . . 8

2.1.4 Bit Timing . . . 9

2.1.5 Message Frames . . . 10

2.1.6 Error Handling . . . 12

2.2 CAN Bus Security . . . 14

2.2.1 CAN Vulnerability Assessment . . . 14

2.2.2 Attack Surface of Modern Vehicles . . . 16

2.2.3 Classification of CAN Attacks . . . 17

2.3 CAN Link Layer Attacks . . . 18

2.3.1 Types of Attack . . . 19

2.3.2 Relevant Research . . . 20

2.3.3 Existing Tools . . . 21

3 The CANPass framework **23**

3.1 Motivation . . . 24

3.2 Problem Overview . . . 24

3.3 Goals and Challenges . . . 26

3.3.1 Limitations Imposed by the CAN Controller . . . 26

3.3.2 Primitives Needed by Link-Layer Attacks . . . 28

3.3.3 Non-Functional Goals . . . 30

3.4 Threat Model . . . 31
3.5 Bitbanging Improvements . . . 32
3.6 The Conflicting Peripherals Approach . . . 33
3.6.1 SPI . . . 35
3.6.2 UART . . . 37
3.6.3 I2C . . . 39
3.6.4 ADC . . . 40
3.7 Framework Design . . . 41
3.7.1 Public Interface Layer . . . 41
3.7.2 Techniques Layer . . . 43
3.7.3 Platform Layer . . . 44

4 Implementation 45
4.1 General Characteristics of the CANPass Framework . . . 45
4.1.1 Language . . . 45
4.1.2 Building and Dependencies . . . 46
4.1.3 File organization . . . 46
4.1.4 Error Handling . . . 48
4.2 Framework Implementation and Usage . . . 49
4.2.1 Interfaces . . . 49
4.2.2 User Configuration . . . 52
4.2.3 Sending Example . . . 52
4.2.4 Receiving Example . . . 54
4.2.5 Adding a New Technique . . . 54
4.2.6 Adding a New Platform . . . 56
4.3 Bitbanging Implementation . . . 56
4.3.1 Reading Improvements . . . 56
4.3.2 Writing Improvements . . . 58
4.4 Techniques Implementation . . . 60
4.4.1 SPI . . . 61
4.4.2 UART . . . 63
4.4.3 I2C . . . 66
4.4.4 ADC . . . 67

5 Experimental Validation 69
5.1 Chosen Platforms . . . 69
5.1.1 Hardware . . . 70

5.1.2 Software 72
5.2 Experimental Approach 72
5.2.1 Objective of the Experiments 72
5.2.2 Simulating Pin Conflicts 73
5.2.3 Exchanged Messages 74
5.3 Test Setup 78
5.3.1 Test Bench Components 78
5.3.2 Nodes of the Test Network 79
5.3.3 Types of Experiments 81
5.4 LPC Experiments 82
5.4.1 LPC Writing Experiments 83
5.4.2 LPC Reading Experiments 85
5.5 AURIX Experiments 86
5.5.1 AURIX Writing Experiments 87
5.5.2 AURIX Reading Experiments 88
5.6 Results 89
5.6.1 AURIX Results 90
5.6.2 LPC Results 91
5.7 Final Remarks 92

6 Conclusions **93**
6.1 Conclusions 93
6.2 Limitations 94
6.3 Future Work 95

Bibliography **96**

List of Figures

1.1 The in-vehicle network of a modern car [39]. 2

2.1 CAN protocol stack, as defined in the ISO standard [16] 6
2.2 The logical components of two CAN nodes connected to the same bus. 8
2.3 Bit representation in the high-speed (left) and low-speed (right) CAN physical layer. 9
2.4 Segments of a CAN bit time. 9
2.5 Fields in a standard CAN data frame. 10
2.6 Error states of the CAN bus. 14
2.7 Attack surface of a modern vehicle. 16

3.1 Sequence diagram of a CAN write operation. 27
3.2 Sequence diagram of a CAN read operation. 28
3.3 Sequence diagram of a CAN bit injection attack implemented with CANPass. 29
3.4 The *Conflicting Peripherals* approach 34
3.5 Timing diagram of an SPI message. 36
3.6 Timing diagram of a UART message. 38
3.7 Timing diagram of an I2C message. 39
3.8 Logical layers of the CANPass framework. 42

4.1 Folder organization of the CANPass framework. 47
4.2 An example of how timer drifting might affect reading. 57
4.3 Resetting the timer on rising/falling edges improves synchronization. 57
4.4 Timing diagram of a CAN signal with injected falling edges. 59
4.5 List of techniques included in the *CANPass* framework. 60
4.6 List of peripherals provided by the *Hardware Abstraction Interface*. 61

5.1 Difference between real pin conflict (a) and simulated conflict (b) . 74

5.2 Three examples of how SPI (a), UART (b) and I2C (c) peripherals can be employed to craft valid CAN messages. 75
5.3 CAN message used for testing SPI and ADC. 76
5.4 CAN message used for testing UART. 76
5.5 CAN message used for testing I2C. 78
5.6 Picture of the test setup. 79
5.7 Wiring of the test setup. 80
5.8 Four different test scenarios. 82

List of Tables

3.1 A list of peripheral conflicts found in some popular automotive microcontrollers. 32

5.1 Comparison between the LPC11C24 and the TC399 microcontroller. 70
5.2 Maximum speed achieved for each writing technique tested. 89
5.3 Maximum speed achieved for each reading technique tested. 89

List of Algorithms

1 Pseudo-code of a selective bit injection attack. 29
2 Pseudo-code for forcing a falling edge when sending a bit. 59

List of Listings

1 The CANPass Receiver interface. 50

2 The CANPass Sender interface. 51

3 An example of application code for the NXP LPC11C24 microcontroller that uses CANPass to perform bitbanging with a hardware timer. 53

4 An example of application code for the AURIX TC399XP microcontroller that uses CANPass to read CAN messages through the SPI interface. 55

Chapter 1

Introduction

1.1 CANBus security

The automotive industry has faced a tremendous evolution over the last couple of decades.

Today, vehicles are equipped with an enormous amount of electronic devices [39], which can include WiFi access points, Bluetooth modules, cellular communication modules, gateways, telemetry systems, and dozens of Electrical Control Units (ECUs) [19]. A modern vehicle, even if not fully featured, already has 70 to 100 ECUs, with over 2500 signals to transmit internally [20]. To coordinate communication among ECUs, in-vehicle networks can be composed of several kinds of bus protocols, but the most prevalent and de-facto standard of such protocols is the CAN protocol.

The Controller Area Network (CAN) is a serial bus communications protocol developed by Bosch in the early 1980s. It defines a standard for efficient and reliable communication between sensors, actuators, controllers, and other nodes in real-time applications. CAN is also employed in a large variety of networked embedded control systems, including those found in ships, spacecraft, industrial plants, and medical equipment.

However, the CAN bus was primarily designed for reliable communication without considering cybersecurity. The lack of encryption, authentication, and integrity checking introduces several vulnerabilities in the CAN protocol, consequently making in-vehicle networks vulnerable to cyber-attacks.

This has led to a growing interest from researchers and vendors for the security of the CAN bus. As a matter of fact, research in automotive security, and in particular CAN bus security, has gained considerable traction in the last few decades, starting from the famous demonstration that Miller and Valasek gave when they

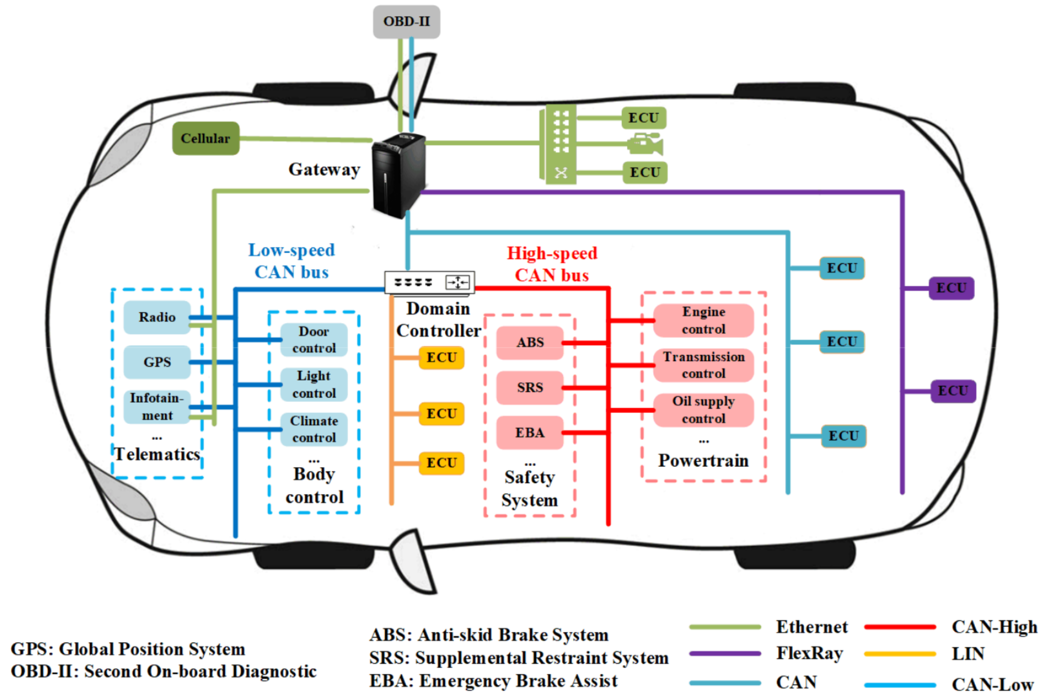


Figure 1.1: The in-vehicle network of a modern car [39].

remotely killed the engine of a Jeep Cherokee driving at full speed in the middle of a highway [27] to more recent attacks that can evade Intrusion Detection Systems [22].

Since the trend for both the connectivity of cars and their autonomy is steeply growing, we expect this problem to be more and more relevant as cars, and vehicles in general, become more and more autonomous and interconnected.

1.2 Link-Layer Attacks

Historically, the first attacks on the CAN bus, such as those analyzed by Miller and Valasek [25], have been performed by injecting malicious messages into a vehicle's internal CAN through a diagnostic port (OBD II), which is mandatory for every car in the US. The main countermeasure that has been adopted to protect already existing systems from such attacks is to employ Intrusion Detection Systems (IDSs) that scan incoming messages on the bus to detect signs of anomalous activity.

More recent attacks, however, such as the one devised by Palanca [33], have demonstrated that vulnerabilities exist also at the CAN data link layer, by exploiting which an attacker can remain undetected by previous IDSs. These new

kinds of attack are possible given that the attacker can inject bits that break the protocol rules at a low level, e.g. if it is possible to write on the bus while another ECU is writing at the same time. This is generally not directly possible on modern microcontrollers, since they access the CAN bus through a separate piece of hardware, called CAN controller, which handles the data link layer according to the protocol.

To avoid this problem, the most common approach is to inject bits on the CAN bus by forcing logical levels from software directly on the pins connected to the bus. This technique is commonly known as *bitbanging*. Since the CAN bus can reach quite high baudrates (1 Mbit/s) and the timing of the injected bits is controlled at a software level, this technique is generally imprecise and unreliable when executed on resource-constrained microcontrollers, such as those found inside ECUs.

Other solutions employ external hardware, use FPGAs or require using perfectly synchronized messages, which is even more difficult at high speeds.

All of the aforementioned techniques currently used to mount link-layer attacks have a common issue: they cannot be adopted in a scenario where an attacker does not have physical access to the CAN bus.

On one hand, performing bitbanging on a high-speed CAN bus from an ordinary ECU is generally considered difficult, as ECUs typically contain resource-constrained microcontrollers that are not built to reach such high frequencies of operations.

On the other hand, techniques that employ external hardware or FPGAs are not suitable for a remote attack, since the adversary cannot implant any additional hardware in the vehicle's electronics system.

The absence of a fast and reliable technique to perform bit injection attacks on ordinary microcontrollers has caused these attacks to be considered infeasible, or at least less likely, in a scenario in which an attacker cannot modify the hardware of an existing ECU nor introduce a new malicious node in the network by physically accessing the bus.

1.3 Our Contribution

The goal of this work is to overcome these limitations and demonstrate that reliable bit injection on the CAN bus from unmodified microcontrollers, such as those found inside typical ECUs, is possible, opening the possibility for link-layer attacks to be performed on remotely compromised ECUs.

Our approach is to bypass the CAN controller using other common peripherals

found in modern microcontrollers, instead of directly accessing the bus pins from software.

We demonstrate that this technique can be used to read and write arbitrary bits on the CAN bus directly from the software layer without any external hardware.

Finally, we present CANPass, an extensible framework that groups together the bypass techniques that we propose in this work. This framework is meant to facilitate the research on CAN link-layer attacks and provide a platform to research new techniques to bypass the CAN controller on unmodified microcontrollers.

Chapter 2

Background

This chapter is meant to provide some background on the general topic of this work, i.e. the CAN protocol and its security in the context of automotive applications.

2.1 The CAN Protocol

The Controller Area Network (CAN) is a bus standard developed by Bosch at the end of the 1980s which has become the de facto standard for connecting Electronic Units (ECUs) in vehicles over the past 40 years. Its applications today range from automotive vehicles to industrial control systems, ships, and even aviation and space [18].

It was introduced as a result of the stringent need in the automotive sector for a serial communication system, at a time when the number of point-to-point connections required by the internal electronics was greatly increasing the wiring overhead.

The progress and success of CAN are due to several factors, such as high noise immunity, robustness, a built-in, priority-based access control mechanism, and the possibility of employing only two wires to interconnect many different devices over relatively long distances.

The CAN protocol was internationally standardized in 1993 as ISO 11898- 1 [17]. The use of CAN in the automotive industry has caused mass production of CAN controllers: today, CAN controllers are integrated on many microcontrollers and available at a low cost, so much so that, according to some estimates, each year about 2 billion CAN nodes are sold [11].

Over the past years, a multitude of higher layer protocols have been developed on top of the CAN specifications to meet the needs of various industrial sectors,

e.g.: CANopen for industrial automation, SAE J1939 for commercial vehicles, UDS (Unified Diagnostic Services) for automotive diagnosis or NMEA 2000 for marine applications. While each of these higher layer protocols has its peculiarities, the underlying physical and data link layers of CAN are common to all of them.

2.1.1 Protocol Stack

The ISO 11898 standard [16] defines the layers of the CAN protocol stack in relation to the OSI model, as shown in Figure 2.1.

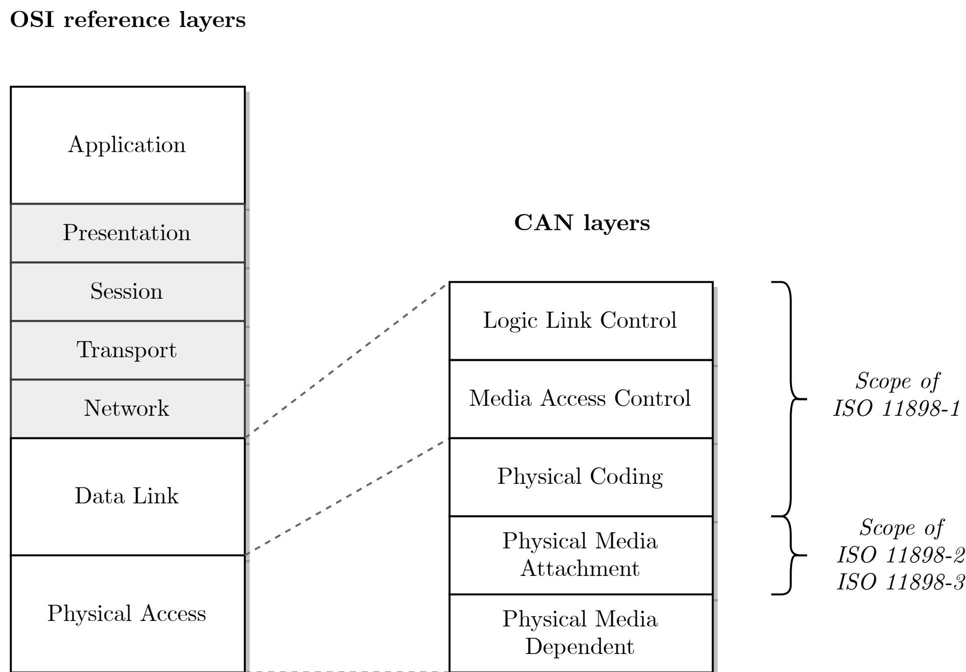


Figure 2.1: CAN protocol stack, as defined in the ISO standard [16]

These layers can be summarized as:

- A **physical layer**, which defines the electrical properties of the bus, including characteristics of the physical medium, connectors, analog signals, voltage levels and bit encoding. In particular, bits of the CAN protocol are encoded as a *differential* signal between two electrical lines, *CANH* and *CANL*, as described in Section 2.1.3.
- A **data link layer**, where various control and integrity fields inside a CAN frame are defined as described in Section 2.1.5. These fields are appended when generating a frame and controlled when receiving one. At this level,

messages are represented as a stream of bits (data frames). Communication errors and retransmissions are also handled at this layer.

- An **application layer**, which handles incoming and outgoing messages. Each message at this layer is represented as composed of a *payload*, that can be 8 bytes long at most in standard CAN, and an *ID*, which is used as message identifier and, implicitly, as a priority tag as well, as described in Section 2.1.5.

The functionality of an ECU (e.g. engine control, driver assistance) is described via high-level software running at the application layer. To carry out actuation and sensing, messages are transmitted and received by the application layer through the lower layers of the communication stack.

In this work, we are especially interested in the *data link layer*, as bit injection attacks and synchronization disruption can be performed stealthily at this level.

2.1.2 CAN Nodes

CAN is designed as a two-wire differential bus that interconnects nodes of a physical network in a broadcast fashion. Each node that participates in CAN communication requires a CAN interface, which is comprised of a *CAN controller* and a *CAN transceiver*, as suggested in Figure 2.2.

The *CAN controller* unit can be found as a stand-alone circuit or often as a dedicated module of the host microcontroller. The controller implements the CAN protocol at the data link layer as described by the standard, generating the bit sequence that has to be transmitted on the bus or decoding incoming bits into messages.

The *CAN transceiver* is responsible for converting between logical data, coming out and going to the CAN controller, and the corresponding physical signaling, as it connects the CAN controller to the physical communication lines. Depending on whether high-speed or low-speed CAN is employed, appropriate high-speed or low-speed transceivers have to be used.

High-speed CAN transceivers support bit rates of up to 1 Mbit/s while low-speed transceivers can only provide communication speeds of up to 125 Kbit/s. An advantage of the low-speed CAN transceivers is that they can provide fault-tolerant communication.

It is important to note the difference in the signals handled by these two components: the CAN controller is in charge of converting application-level objects (messages) to a sequence of bits and vice-versa, using the *CANTX* and *CANRX*

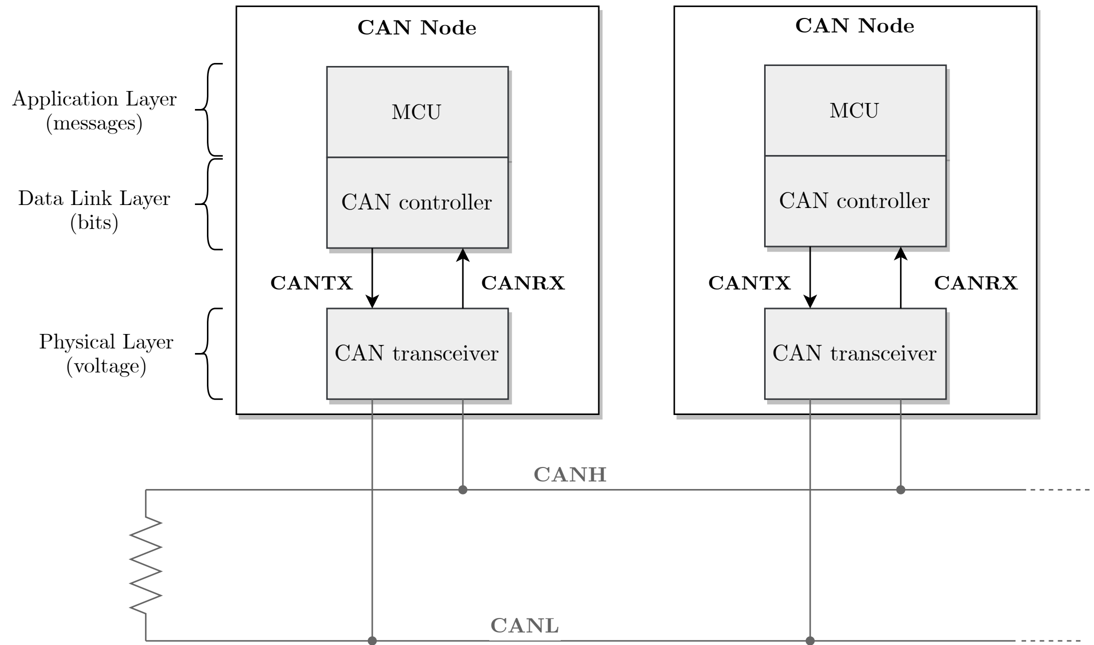


Figure 2.2: The logical components of two CAN nodes connected to the same bus.

lines. On the other hand, CAN transceivers take care of transforming each bit received from CANTX into a voltage difference between *CANH* and *CANL*, and continuously monitor the bus to output the current differential level as a '1' or '0' on the CANRX line.

In other words, CANTX and CANRX are *digital* signals that typically range from 0V (logical '0') to 3.3V (logical '1'), while CANH and CANL are *analog* signals, that employ higher voltage lines.

The main focus of this work is to provide new techniques to *bypass* CAN controllers, since they are in charge of enforcing the protocol rules and don't allow for arbitrary bits to be injected or read from software.

2.1.3 Physical Layer

At the physical level, i.e. on the differential bus, the CAN specification defines two kinds of bit: *dominant* bits, whose logical value is conventionally '0', and *recessive* bits, corresponding to logical '1'.

These logical values are encoded for transmission on the physical layer as differential signals. The specific differential voltage levels are dependant on the type of transceiver, as illustrated in Figure 2.3. A high-speed CAN transceiver (ISO

11898-2) interprets a differential voltage of up to 0.5 volts as a recessive value, while a differential voltage that exceeds 0.9 volts is considered as the dominant level. Low-speed transceivers (ISO 11898-3) will consider a differential voltage of 5 volts as a recessive bit and a typical differential voltage of 2 volts as a dominant bit.

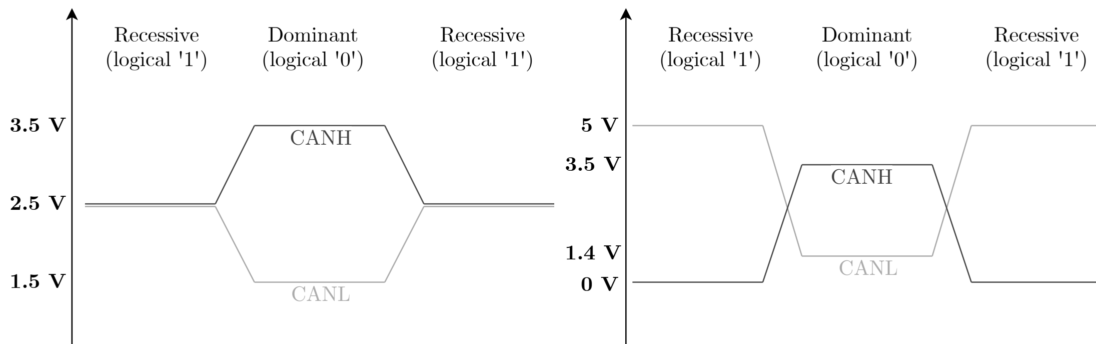


Figure 2.3: Bit representation in the high-speed (left) and low-speed (right) CAN physical layer.

According to the CAN specification, a dominant bus level must always overwrite a recessive bus level, therefore, the CAN bus is implemented as a wired-AND bus. This behavior enables the implementation of CAN protocol features such as the arbitration-based bus access, where the lowest message identifier has priority.

As a result, if any device on the CAN bus transmits a dominant bit, which is represented by a '0' at the logical level, it will overwrite any other ongoing communication. Most bit-injection attacks on the CAN bus exploit this mechanism to cause errors on the bus.

2.1.4 Bit Timing

Within the CAN protocol, the transmission time of each bit is divided into four segments, as depicted in Figure 2.4.

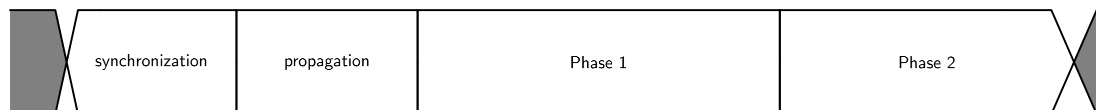


Figure 2.4: Segments of a CAN bit time.

- The **synchronization segment** is where the bit is expected to start, i.e. where a transition from one logical level to the other should occur, if any

- the **propagation segment** is introduced to compensate for the signal propagation delays
- The **Phase 1** and **Phase 2** segments are used for resynchronization by being lengthened or shortened. The bit value sampling occurs directly after the Phase 1 segment, at the *Sampling Point*

The length of each segment is an integer multiple of the *Time Quantum* (TQ), the smallest timing resolution used by a node to derive the bit time.

The CAN standard also defines two synchronization mechanisms that each device must implement:

- **Hard resynchronizations** occur on all recessive-to-dominant transitions signaling the beginning of a frame, i.e. on the Start Of Frame (SOF) bit
- **Soft resynchronization** occurs at each subsequent falling edge, and can adjust the bit time by an amount specified as the *Synchronization Jump Width* (SJW) which is limited to a maximum of 4 TQs

These synchronization mechanisms can be used to either disrupt the bus synchronization, as described in Section 2.3.1, or to enhance some bit injection techniques, as illustrated in Section 3.5.

2.1.5 Message Frames

The CAN controller is in charge of sending data on the CAN bus after transforming an application-level message into *frames*.

Each standard CAN data frame comprises the following fields, depicted in Figure 2.5:

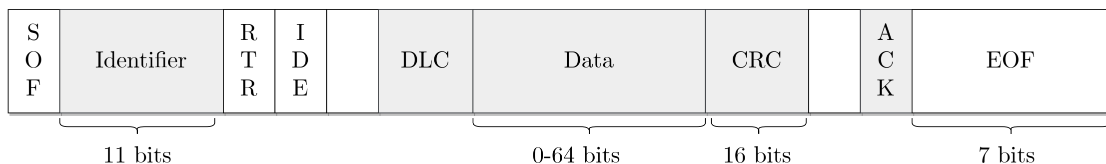


Figure 2.5: Fields in a standard CAN data frame.

- The **Start of Frame** (SOF), signalled by the transmission of a dominant bit. Since the idle state of the bus is recessive, this means that each CAN frame start is detected by CAN controllers whenever a falling edge is received on the *CANRX* line preceded by a sufficient number of recessive bits.

- The **Identifier**, which has a length of 11 bits in standard frames and 29 bits in extended frames.
- A **RTR** bit, which signals a remote request for data, if set.
- An **IDE** bit, which specifies if the current frame is a standard frame or an extended one.
- A reserved bit in which the bus should be set to a dominant state.
- The **Data Length Code** (DLC) field, containing the length of the payload (in bytes).
- The **Data** field, containing the payload.
- A **CRC** field, that contains the Cyclic Redundancy Check code computed over all the preceding fields.
- A **CRC delimiter**, which should be always set to '1'.
- A 1-bit **Acknowledgement** field, in which the sender device leaves the bus in a recessive state. Any dominant bit communicated here will be treated as a positive acknowledgment.
- A recessive delimiter
- The **End of Frame** field, composed of 7 consecutive recessive bits.

One of the great advantages of CAN is its *arbitration mechanism*, which is possible thanks to how dominant and recessive bits are defined at the physical layer. This protocol employs CSMA/CA, which stands for *Carrier Sense Multiple Access* with *Collision Avoidance*. This means that each device is expected to assure collision avoidance on the bus by strictly following the protocol rules.

In particular, the CAN arbitration mechanism is applied over the first part of the frame, i.e. the ID field. It requires the sending nodes to monitor the transmission bit by bit. If two or more nodes start sending a frame at the same time, they each continue the transmission as long as the value of the bit read out from the bus equals the value they have written on the bus. Whenever one of the two devices reads a value on the bus that is different from the one it has written, it will immediately back off. Since a node putting a recessive bit on the bus will always lose arbitration to a node writing a dominant bit, identifiers with lower values have higher priorities.

Another mechanism employed by CAN is *bit stuffing*, which is a procedure required to keep the nodes on the bus synchronized. The mechanism involves inserting additional bits of the opposite value after each sequence of 5 identical consecutive bits. The stuffing bit will be inserted even if the 6th bit in the normal transmit sequence is different in value than the previous 5 identical bits.

As a final remark, all of the protocol rules mentioned until now, including the insertion of special fields in the message frame, the CRC calculation, the collision avoidance mechanism, the resynchronization mechanisms, and the bit stuffing rule, are enforced by the CAN controller.

As a matter of fact, while the CAN transceiver is merely a translator from the digital signals to the analog bus, the CAN controller is a much more complex device that is in charge of ensuring that the protocol rules are followed. This is one of the main reasons why, for injecting arbitrary bits on the bus, we need methods to bypass the CAN controller, as discussed in Section 3.3.

2.1.6 Error Handling

Understanding how errors are handled in the CAN protocol is relevant for understanding how link-layer bus-off attacks work, which is a central topic of the present thesis.

Possible Errors

The CAN protocol defines an error detection mechanism based on bus monitoring, performed by both the sender and receiver of a message. The sender is responsible for monitoring the sent message bit-by-bit, as well as reading the acknowledge field. When monitoring the bus bit levels, the sender compares the sent bit value with the actual sampled bit value. A *bit error* exists if these values differ. Moreover, since all sent CAN messages should be acknowledged by a receiver node, the ACK field is checked by the sender. If the positive acknowledgment is missing, then the sender records an *acknowledgment error*.

The message format, bit stuffing and checksum are verified on the receiver side. A *form error* occurs whenever a message is found to be non-conformant with the specification. Breaking the bit stuffing rule results in a *bit stuffing error* while failing to verify the message CRC produces a *CRC error*.

Whenever an error is detected, the detecting node starts sending an error frame beginning from the first bit following the error detection. The CAN error frame starts with the *error flag*, which consists of 6 dominant bits.

Since this breaks the bit stuffing rule, even if the error was detected by a single node, this field is meant to ensure that all other nodes send an error flag.

The secondary error flag, which also consists of dominant bits, is meant to compensate for the later detection of errors and can be 0 to 6 bits in size.

The last segment of the CAN error frame, called the *error delimiter*, consists of 8 recessive bits. After the error frame was sent and the intermission time has elapsed, the sender of the erroneous message will try to retransmit it.

Note that this mechanism happens at the data link layer: the CAN controller automatically takes care of generating error messages and retransmitting messages after an error has been detected. Therefore, the application layer is never involved in these operations. Most embedded CAN controllers include some kind of mechanism through which the hardware can be notified of new errors or can disable the automatic retransmission of a message in case of errors, but, in general, the detection and generation of error messages are not under the application layer's control.

Fault Confinement

The CAN protocol specification describes a fault confinement mechanism to prevent faulty nodes from creating high bus loads.

According to this mechanism, each node should implement two error counters: **TEC** (Transmission Error Counter) and **REC** (Receive Error Counter). These error counters are decremented by 1 on each successful transmission or reception of a data frame.

Upon detection of an error, the sender node increments TEC by 8, while receivers increment REC by 1 unless they are the ones causing the error, in which case REC is incremented by 8.

Depending on the values of these error counters, a CAN node can be in one of three error states:

- **Error Active**: when in this state, the CAN node behaves normally without any specific restriction.
- **Error Passive**: nodes in this state can only indicate an error by sending 6 recessive bits, preventing other nodes from globalizing the error. When sending consecutive data frames, nodes in this error state must wait for an additional time equivalent to 8 bits (Suspended Transmission Time).
- **Bus-Off**: nodes that reach the bus-off state can no longer influence the bus

communication in any way. This state can only be exited after 128×11 correctly recorded recessive bits.

Figure 2.6 shows the possible transitions between these three states along with the triggering conditions.

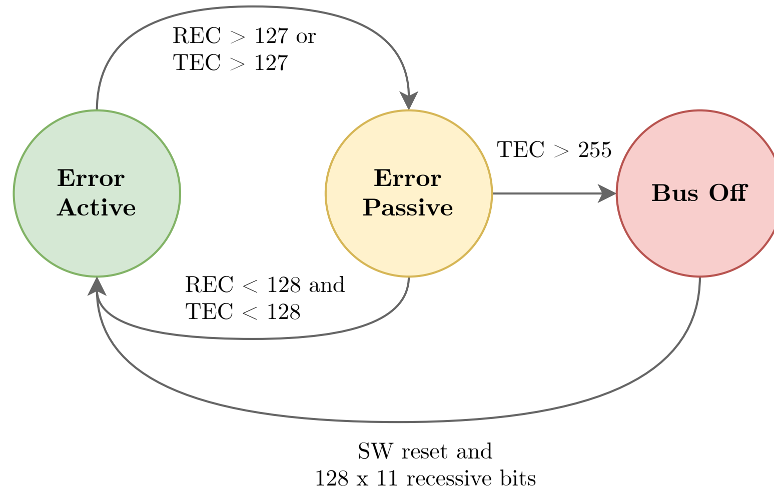


Figure 2.6: Error states of the CAN bus.

2.2 CAN Bus Security

This section provides an overview of some of the relevant security research carried out in the automotive field, and on the CAN protocol in particular, during the last decades. It is meant to give the reader a high-level idea of the past and current challenges that have been faced in this field, and an overview of the possible attacks that can be carried out against modern vehicles by means of the CAN bus.

2.2.1 CAN Vulnerability Assessment

Similar to many other protocols in the embedded world, and also older internet protocols, the CAN bus lacks some fundamental security mechanisms, which makes vehicles vulnerable to malicious adversaries.

According to *CIA* (Confidentiality, Integrity, Availability) security model, we can identify a set of vulnerabilities that are intrinsic to the protocol itself and are present in any possible implementation of such standard.

As described in [39], the following characteristics of the protocol are problematic from a security point of view:

1. **No encryption**: the absence of encryption of the CAN frame's content allows an adversary to easily analyze the behavior of target ECUs based on the historically recorded CAN frames, violating the confidentiality principle.

2. **No authentication**: since the CAN protocol has no built-in mechanism to authenticate the source of a CAN message, any transmitter can indistinguishably transmit a CAN frame to any ECUs connected on the CAN bus, which violates the integrity principle. This means that any node on the bus is able to send all available commands to any other ECU by transmitting fabricated CAN frames containing appropriate contents.

3. **Broadcast transmissions**: CAN frames are both physically and logically broadcasted to all the connected ECUs: every ECU receives the frames transmitted on the CAN bus. Although manufacturers typically segment CAN networks and use CAN firewalls to prevent communication from separate sets of nodes, a malicious ECU can easily leverage the broadcast nature of the CAN bus to eavesdrop on the CAN frames transmitted by other ECUs within one segment.

4. **Priority-based Arbitration**: The CAN protocol imposes that the ID of a CAN frame is used to designate its priority. This is far more than a conventional rule imposed in the link-layer: it directly descends from the electrical characteristics of the bus, which define a priority between bits at a physical level. The priority-based arbitration mechanism allows a CAN frame with a smaller ID (higher priority) to be transmitted on the CAN bus while forcing all the other CAN frames to back off. At the simplest level, a malicious ECU could assert a dominant state on the CAN bus indefinitely, preventing legitimate ECUs from transmitting any CAN frames. In this way, adversaries can easily produce a Denial of Service (DoS) attack against CAN networks, violating the availability principle.

5. **Limited Bandwidth and Payload**: Limited by the bandwidth and the payload, the CAN bus cannot provide strong access control. For example, in order to protect ECUs against certain operations without authorization, ECUs diagnostic services are supposed to use fixed challenges (seeds) and store the corresponding responses (keys) for the challenge-response pairs. Since the length of the challenges and the responses are too short, the adversaries can crack the key of an ECU within eight days through a brute-force attack [21].

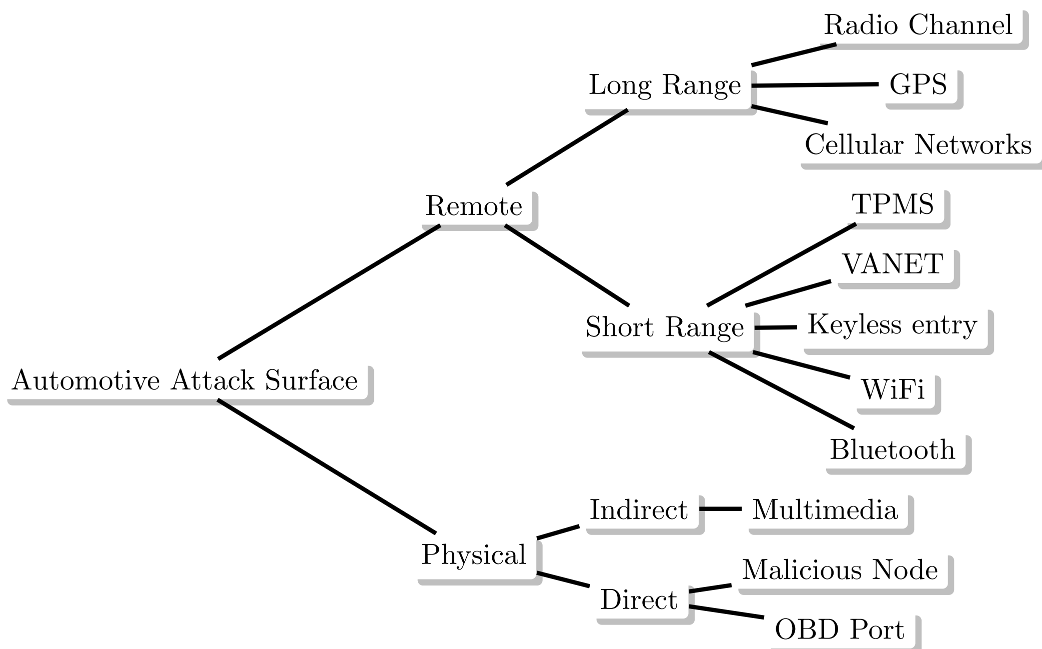


Figure 2.7: Attack surface of a modern vehicle.

These intrinsic aspects of the CAN protocol lay the foundation for many attacks to be perpetrated on the CAN bus, even with unsophisticated technology.

2.2.2 Attack Surface of Modern Vehicles

Having described the characteristics that make networks based upon the CAN protocol vulnerable, we proceed with a broader overview of the attack surface of modern vehicles and describe how such networks can be accessed from adversaries in the first place.

A considerable amount of research has been carried out on this subject [4] [5] [6] [21] [26], highlighting many aspects of modern vehicles that can be exploited by malicious actors.

Figure 2.7 illustrates an example of such analysis, as provided by [5].

The main aspect that is relevant for this thesis is how the internal CAN bus can be breached, therefore opening the possibility of exploiting the aforementioned vulnerabilities in the CAN protocol.

Some possible ways to gain access to the CAN bus are:

- A malicious node, that has been implanted in the electronic system by an adversary.
- A malicious diagnostic device connected to the diagnostic port (OBD-II port), which is mandatory for all cars sold in the USA. This can consist

of either an ECU, a malicious diagnostic device, but also a custom-made electronic device, which might be connected to the outside world.

- A compromised node, i.e. a legitimate internal node of the CAN bus that has been compromised either by physical access or remotely and is in full control of the adversary.

In this work, we will focus on compromised nodes, in particular *remotely compromised* nodes, i.e. unaltered ECUs that have been compromised by an adversary who can remotely reprogram their behavior.

Given the number of external connectivity devices that modern cars include, breaking into the CAN bus by remotely compromising an ECU has been shown to be theoretically and practically feasible in many different ways, for example using Bluetooth [6], the FM channel [6], wireless interfaces [26][29], cellular interfaces [29], OTA mechanisms [1] and more.

2.2.3 Classification of CAN Attacks

As mentioned in Section 2.2.1, the CAN bus protocol has no built-in encryption and authentication mechanisms, and it cannot determine whether the data was replayed by a malicious node even if a corresponding cryptographic mechanism is adopted to tackle the previous vulnerabilities.

A general classification of the possible attacks that can be carried out on the CAN bus given the aforementioned vulnerabilities is reported in [39]. The possible identified attacks are:

- **Eavesdrop Attack**: since CAN frames are broadcasted to all ECUs in the same segment without encryption, an attacker with access to the CAN traffic could eavesdrop on the CAN bus to collect and analyze CAN frames. This can be used as a way of characterizing the system or fingerprinting the vehicle's driver [9], as well as reverse-engineering the CAN IDs to prepare for a replay or impersonation attack.

- **Replay Attack**: given the lack of authentication for CAN frames, an attacker that can write messages on the bus can easily re-transmit a message sent by another To carry out this attack correctly, the adversary must have the ability to prevent the original ECU from transmitting on the bus.

- **Spoofing Attack**: under the same assumptions, an attacker can not only replay an already sent message, but completely impersonate another ECU

by sending messages on the bus with the same ID and same period as the legitimate ECU.

- **Denial-of-Service (DoS)**: an attacker that controls an ECU on the CAN bus is able to prevent other ECUs from transmitting or receiving data, either by causing an *arbitration denial* or forcing the victim ECU in a *bus-off* state. Finally, it is also possible to completely disrupt communication on the bus by sending a constant dominant bit or by forcing ECUs to de-synchronize. More details about this kind of attacks can be found in Section 2.3.1.

The aforementioned attacks can be used to target safety-related systems, such as the ABS, change the information displayed on the dashboard, either hiding an existing issue to the driver or signaling an inexistent one, disturb or take control over autonomous features, such as parking assistance or cruise control, and finally completely shut down a car, as famously demonstrated by [27].

To better understand the context of this work, we introduce a further distinction based on the technique used to carry out such attacks:

- **Message Recording** techniques are used to record the traffic on the bus by reading all messages sent and received, as in the eavesdropping attack
- **Message Injection** techniques can be used to carry out replay attacks and spoofing attacks by accurately crafting CAN messages
- **Bit Injection** techniques consist in injecting single bits on the bus instead of full messages, which can be used to carry out DoS attacks as described in Section 2.3.1

This distinction is useful to identify the scope of this thesis, which is mostly focused on enabling *bit injection* attacks in remotely compromised ECUs.

2.3 CAN Link Layer Attacks

This section provides some background on *link-layer attacks*, i.e. attacks that can be carried out at the CAN data link layer. All of these attacks, and possibly future ones, can benefit from our research, as this work aims at providing a way to unboundedly access the data link layer from software

Data link layer attacks are relevant from a security perspective because they are generally harder to detect for Intrusion Detection Systems, as they operate below the level of messages and are indistinguishable from a genuine bus fault.

Currently, the common consensus is that such attacks can be carried out with physical access to the CAN medium, but not remotely, as they are too unreliable and require too much precision to be correctly performed by automotive microcontrollers without any modification.

The present work aims at changing this conception of link-layer attacks.

2.3.1 Types of Attack

Many attacks to the CAN link layer have been recently devised. The possible attacks that can be carried out by manipulating the data link layer of the CAN protocol can be summarized as:

- **Complete Denial Of Service:** an electrical property of the CAN bus is that dominant bits, i.e. '0's, have a priority over recessive bits. This means that keeping the bus constantly in a dominant state will prevent any further communication from being performed on the bus. Although modern CAN transceivers are typically designed to prevent this from happening, this can usually still be done by keeping the bus in a dominant state for long intervals, effectively disrupting any possible communication.

- **Selective arbitration denial:** since messages with lower IDs have higher priority in the CAN protocol, injecting dominant bits on the bus while a message ID is being communicated will cause the transmitting device to lose the arbitration, which forces it to back off and stop transmitting. This can be done repeatedly during the transmission of specific messages to prevent an ECU from ever winning bus contention.

- **Selective bus-off attacks:** if, instead, a dominant bit is injected in the *payload* of a message, while a transmitting device is sending a recessive bit, the transmitting device will detect an error on the bus, increasing its internal error counter and immediately terminating the transmission. Repeating this process a certain number of times will cause the device to accumulate too many errors, which forces it to go into a bus-off state. This mechanism can be used to completely shut down the communication of any of the nodes connected to the CAN bus.

- **Synchronization disruption:** finally, both the synchronization mechanisms and the sampling point settings of the CAN protocol can be used to cause a desynchronization between nodes on the CAN bus, and, in some cases, this can cause different nodes to read differently the same message on

the bus. This can be particularly useful to evade IDS message inspection, or simply disrupt communication on the CAN bus.

In general, these kinds of attack can be used in conjunction with other described in Section 2.2.3 to shut down a critical ECU, to disable its functions or impersonate it.

2.3.2 Relevant Research

While being a relatively new topic in automotive security, a considerable amount of possible link-layer attacks have been studied by researchers in the field.

Some implementations of such attacks that we consider as a reference in this work are:

- **Error Handling of In-vehicle Networks Makes Them Vulnerable**, Kyong-Tak Cho and Kang G. Shin, 2016 [7]. This work shows a simple bus-off attack in which a compromised device is used to communicate on the bus at the same time of the victim ECU. If the messages IDs are identical and the compromised ECU communication is perfectly synchronized with the victim ECU, neither of the two nodes will lose arbitration. Therefore, if the attacker injects a recessive bit in the payload, the victim ECU will consider it as a bus error, increasing its TEC by 8 as previously described and, eventually, going in bus-off state. Even if, strictly speaking, this attack is performed at the application layer, since it requires injecting a complete message on the bus, it can be seen as a first attempt to inject a dominant bit in the place of a recessive bit.

- **A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks**, Andrea Palanca, Eric Evenchick, Federico Maggi and Stefano Zanero, 2017 [33]. This work presents a proper link-layer attack on the CAN bus, performed on an Arduino board by bypassing the CAN peripheral and using direct GPIO access instead. It also contains a real-world experiment on a 2012 Alfa Romeo Giulietta, with a reported CAN speed of 50Kbit/s.

- **DoS Attacks on Controller Area Networks by Fault Injections from the Software Layer**, Pal-Stefan Murvay and Bogdan Groza, 2017 [28]. This paper contains an experimental analysis of all the aforementioned bit-injection techniques and their impact on proposed CAN authentication pro-

tocols. It reports the reliability achieved by using bitbanging on a real-world microcontroller for each technique analyzed.

- **CANCloak: Deceiving Two ECUs with One Frame**, Li Yue, Zheming Li, Tingting Yin, Chao Zhang, 2021 [38]. This paper shows a new attack that can be carried out when two devices have different sampling points. It creates a message that is read differently by the two devices, thanks to the fact that the level on the bus is switched *in the middle* of the CAN bit, between the two sampling points. The reported experiments have been performed using a laptop processor (Intel Core i7-8550U) to achieve the precision needed.
- **WeepingCAN: A Stealthy CAN Bus-off Attack**, Gedare Bloom, 2021 [3]. This work presents a variant of [7] in which the faulty bits are injected at random positions inside the payload, to evade possible IDS inspection.
- **CANNON: Reliable and Stealthy Remote Shutdown Attacks via Unaltered Automotive Microcontrollers**, S. Kulandaivel, S. Jain, J. Guajardo and V. Sekar, 2021 [22]. This paper presents a novel approach to bit injection. It employs the CAN peripheral to inject arbitrary bits with extreme precision by pausing its clock during transmission and resuming it when the next bit needs to be injected. Unfortunately, this method does not provide a way to read bits while the CAN peripheral is frozen, and relies on message periodicity to know the right moment to inject bits.

2.3.3 Existing Tools

Apart from [22], all the other aforementioned papers rely on a common technique for injecting bits: either they inject CAN messages at the exact time as another ECU, or they inject bits using *bitbanging*.

The term bitbanging is commonly used to identify a set of techniques that aim at emulating in software the behavior of a certain protocol by forcing logical levels on the bus pins. These techniques are commonly employed in situations where there is no dedicated hardware peripheral to handle a certain protocol, such as SPI and I2C, but also to take control over these buses, as reported in [37].

While implementing a complete set of the CAN bus rules in software is a challenging task, being able to at least read and write bits on the bus with the correct timing is enough for mounting link-layer attacks, which is the scope of the present work.

In particular, tools for carrying out CAN bus bitbanging already exist, namely:

- **CANT** [2]: The CANT project is a piece of code designed to run on ST Microelectronics Nucleo-H743ZI boards that allows user code to directly interact with the CAN bus bypassing the CAN controller. A strong disadvantage of this library is that it also comes with a custom external shield, that is needed for the hardware to be able to force logical levels on the CAN bus.

- **CANHack** [36]: the CANHack library is another open-source project that aims at enabling low-level manipulation of the CAN bus, and also features a considerable amount of already prepared link-layer attacks that can be tested. It is mainly designed to run for a Raspberry Pi, but extensions to other platforms are also possible.

This said, there are several downsides to bit-banging: for example, it increases the power consumed by the microcontroller since the processor is busy for the whole transmission time. Moreover, if compared to a dedicated hardware peripheral, more communication errors like glitches and jitters occur when bitbanging is used, especially when data communication is being performed by the microcontroller at the same time as other tasks. Finally, communication via bitbanging usually happens at a fraction of the speed with which it occurs when dedicated hardware is used.

Chapter 3

The CANPass framework

In this chapter, we propose a new set of software-only techniques that can be used to read and write bits at the CAN data link layer. In particular, we demonstrate how, on modern MCUs, peripherals that share pins with an on-chip CAN controller can be exploited to bypass the CAN peripheral and read or write arbitrary bits at arbitrary moments directly on the CAN bus.

The reliability of these techniques and the fact that they do not need any external hardware makes them suitable for performing bit injection even from simple microcontrollers, without the need for physical access to the bus. This means that, given an adversary who is able to remotely compromise an ECU connected to the CAN bus, which has been demonstrated to be possible, it can also perform link-layer attacks from that ECU remotely, which is generally considered impractical with current techniques.

This approach is, to the best of our knowledge, completely novel, since previous work on this topic either employs bitbanging, which is slow and unreliable, or uses the CAN peripheral itself [22], which cannot be used to read the ID of the current message before executing the injection.

The ultimate goal of this work is to build an extensible framework, called *CANPass*, around the concept of conflicting peripherals. This framework includes an implementation for the standard bitbanging techniques, which can be improved as described in Section 3.5, as well as some new techniques that are listed in Section 3.6. From a practical point of view, the CANPass framework can be seen as a swiss-army knife for CAN link-layer manipulation, whose applications are especially relevant in the context of the CAN protocol's security.

3.1 Motivation

As discussed in Section 2.3.1, an adversary with physical access to the CAN bus can bypass the CAN data link layer and inject bits directly on the bus, breaking the protocol's specification. This approach can be used to carry out bus-off attacks in a stealthy way, as the produced behavior is difficult to distinguish from a genuine bus fault, which in turn can be used to attack safety systems, cruise control systems, perform car ransom attacks, and much more.

Since many researchers have highlighted a growing attack surface for vehicles, which includes many wireless vectors, we are interested in verifying the possibility of carrying the aforementioned link-layer attacks also *remotely*. These attacks are, in fact, currently considered impractical in a remote scenario, since the current techniques lack the sufficient reliability to be performed on an ordinary microcontroller, or require additional hardware to be installed, which implies having physical access to the electronics system.

With our work, we want to change this concept of link-layer attacks, by demonstrating new techniques that can carry out reliable and stealthy link-layer manipulation from software on unmodified microcontrollers. As a consequence, such techniques allow for link-layer attacks to be performed remotely.

Remote attacks are important from a security perspective for two main reasons:

- They are generally more scalable since the simultaneous control of many devices is much easier in a remote context
- They impose less burden on the attacker, since the presence of a remotely compromised device can be effectively hidden from the user and requires less exposure for an attacker

For this reason, enabling link-layer attacks on remotely compromised devices, which is the main consequence of our work, makes such attacks more dangerous. A detailed discussion of our threat model is provided in Section 3.4.

3.2 Problem Overview

Obstacles to Remote Link-Layer Attacks

As already mentioned, although link-layer attacks have been proven to be possible in scenarios where the attacker has physical access to the CAN bus, they are generally not considered feasible in remote scenarios.

The main reason for this is the fact that modern microcontrollers, which are the main ”brain” inside automotive ECUs, do not communicate on the CAN bus directly from the CPU, but instead use separated, on-chip devices called *CAN controllers*, described in Section 2.1.2. These components are in charge of filtering the IDs of incoming messages, detecting bus errors, performing collision avoidance, and enforcing all the protocol rules that would otherwise need to be implemented in software. As a consequence, the CPU does not have direct access to the data link layer, as the CAN controller takes care of transforming bits into CAN messages and vice-versa, forwarding them to or from the CPU.

The fact that a separate piece of hardware takes care of enforcing the protocol's rules limits what a microcontroller can do on the CAN bus. For example, it cannot interrupt another ECU's transmission, nor can it read a message while it is being transmitted. As a result, the software layer has little control over the timing of reading and writing operations at the bit level: only full, valid packets can be sent and received on the CAN bus, and only when the protocol permits it.

These limitations make link-layer attacks such as the one devised by Palanca [33] and others mentioned in previous chapters substantially impossible in a scenario where the communication is performed through a CAN controller. The only way to mount these attacks is to bypass the controller and access the bus directly.

Limits of the Current Methods

Currently, direct access to the CAN bus can be done in microcontrollers equipped with an embedded CAN peripheral via *bitbanging*, i.e. controlling via software the pins connected to the bus and replicating the peripheral's behavior by reading or forcing the digital level on the pin at specified, hand-crafted intervals.

Implementing everything in software has some major drawbacks in real-world applications, which have been highlighted in Section 2.3.2. As a matter of fact, it is very difficult to achieve reliable and high-speed communication on microcontrollers with this technique.

For this reason, link-layer attacks are generally considered impractical in a scenario where the attacker has access to a remotely compromised ECU, since their limited hardware resources make software-only communication on a high-speed CAN bus extremely cumbersome and unreliable.

Our Approach

The purpose of this work is to move beyond these limitations and demonstrate that it is possible to perform reliable CAN data link layer manipulation even in the absence of physical access to the CAN bus.

We propose some improvements to existing bitbanging techniques, making them more reliable even at higher speeds, and, most importantly, we extend the concept of bitbanging to a more general solution, which employs other commonly used peripherals, such as I2C, SPI, UART and ADC peripherals.

Since this approach opens up a multitude of possible solutions, both general and specific to a certain microcontroller or vendor, we want to provide a common interface for these proposed techniques, so that they can be used interchangeably on multiple hardware platforms. For this reason, each technique has been implemented in the context of a more general framework, called CANPass.

This is intended to ease both the research of low-level CAN protocol vulnerabilities and the development of new techniques to bypass the CAN peripheral.

3.3 Goals and Challenges

In order to provide a better understanding of the contributions of this work, this section contains a technical analysis of the problem of bypassing the CAN peripheral and a discussion of the goals and challenges of our approach.

3.3.1 Limitations Imposed by the CAN Controller

As already mentioned, the presence of CAN controllers limits what can be done by the software layer on the CAN bus. Typical limitations that CAN controllers impose by design are:

1. **The software cannot write arbitrary bits on the bus:** the software layer can set the ID and payload of the message it wants to send, but the controller takes care of adding stuff bits, calculating the CRC, and setting control bits according to the standard.

2. **The software cannot control when a bit will be sent:** since the controller is responsible for avoiding collisions on the bus, it will send the packet only when there is no other ongoing communication, and it will automatically back off if another message with higher priority is being sent in the same exact moment.

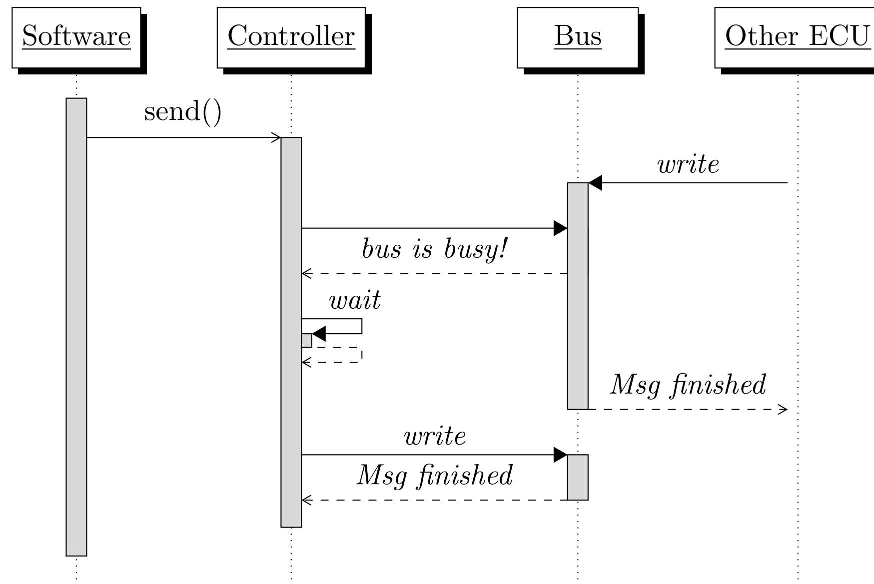


Figure 3.1: Sequence diagram of a CAN write operation.

If the bus is occupied by another transmission, the CAN controller will delay the message until the bus is free.

3. **The software cannot control the timing of each bit independently:** in order to generate a valid communication, bit timings are defined by the current baudrate and cannot be individually tweaked.

4. **The software cannot read bits in real-time:** the controller will signal the arrival of a new message to the CPU only after it has received the message in its entirety, and no error has been detected. It is generally not possible for the software to reliably read bits while they are being sent on the bus.

To give an intuitive understanding of what these limitations mean in practice, Figure 3.1 provides a high-level description of what a typical *write* operation looks like from the controller point of view: if the CPU wants to send a message on the bus, the controller has to first make sure that no other participant on the bus is transmitting. If there is a conflict, it will wait until the end of the message, check that no other connected transceiver is willing to initiate a communication, and only then it will send the desired packet on the bus.

Figure 3.2, on the other hand, illustrates a situation in which a device on the bus sends an erroneous message: the error is communicated, the transmission is interrupted, and the messages is re-sent by the device, without the software layer ever being notified.

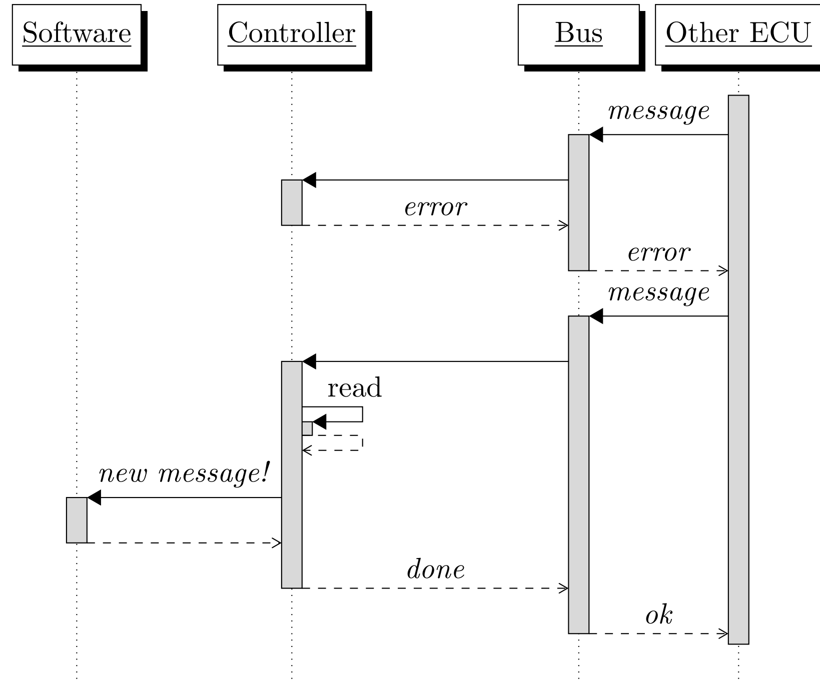


Figure 3.2: Sequence diagram of a CAN read operation.

The CAN controller notifies the software layer only when a complete message is received without errors.

3.3.2 Primitives Needed by Link-Layer Attacks

Link-layer attacks to the CAN bus are typically based on the violation of one or more of the protocol's formal rules. For example, selective bus-off attacks such as Palanca [33] consist in causing errors during the transmission of a target ECU in a way that forces the target to eventually go in a *bus-off* state, therefore becoming unresponsive.

Algorithm 1 contains a high-level procedure that implements the core of such attack, which can be summarized as: read the ID of each incoming message, compare it against a chosen ID, and inject a series of bits at a specific point of the packet in case the two are identical.

Figure 3.3 graphically represents the sequence of events that would happen in such a scenario. It is evident how essential it is for the software layer to have control over the timing of each operation, in order to be able to inject bits during another transmission. This cannot be achieved using the CAN peripheral.

In general, to be able to effectively perform low-level attacks to the CAN protocol we need at least the two following primitives:

Algorithm 1 Pseudo-code of a selective bit injection attack.

1: **procedure** SelectiveInjection(ID, BITS)
2: **while** true **do**
3: $waitForFirstBit()$
4: $ReceivedID \leftarrow$ **ReadBits**$(size_of(ID))$
5: **if** $ReceivedID == ID$ **then**
6: **for** $i \leftarrow 0$ to $size_of(BITS)$ **do**
7: **writeBit**$(BITS[i])$

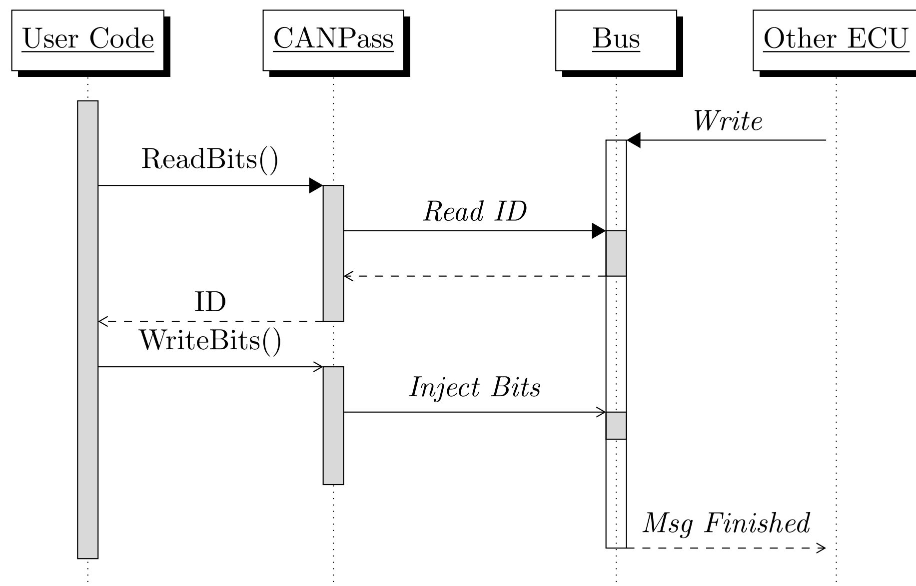


Figure 3.3: Sequence diagram of a CAN bit injection attack implemented with CANPass.

- **Read Primitive**: Reads a specified amount of bits, then immediately returns to the caller.
- **Write Primitive**: Immediately starts writing bits on the bus.

The main goal of CANPass is to provide a collection of techniques that implement the aforementioned primitives so that they can be used to communicate on the bus bypassing the CAN peripheral.

3.3.3 Non-Functional Goals

Finally, it is important to note that our solution is also designed to meet some non-functional goals, which are essential in order to make it usable in real-world applications.

As a matter of fact, while the various bitbanging techniques presented in the previous chapter can be used to bypass the CAN peripheral, they are extremely fragile solutions, that typically need to be fine-tuned to the specific microcontroller in order to obtain good reliability at higher speeds, and sometimes cannot be used at all, if the microcontroller's CPU clock is not fast and stable enough.

Message-based techniques, which involve sending a CAN message at the exact same time as another one, are even more subject to timing issues.

In contrast, some important features that CANPass offers are:

- **Speed:** Modern CAN is designed to reach a maximum bitrate of 1Mbit/s. We want our techniques to be able to keep up with these speeds, even on slower microcontrollers.
- **Reliability:** We must be able to correctly read bits that are being sent on the CANBus and write bits with accurate timing in such a way that we can mimic the functioning of a real CAN peripheral most of the time.
- **Extensibility:** Adding a new technique or substituting one inside already existing code should be as easy and frictionless as possible. For this reason, we define a set of common interfaces that each technique must adhere to, as well as a set of abstract peripherals that each platform should implement.
- **Cross-Platform:** We want to provide some kind of hardware abstraction layer that enables the techniques to be developed in a hardware-agnostic way.

3.4 Threat Model

As a reference threat model, we consider a scenario in which a remote adversary has targeted a non-safety-critical ECU (e.g. the head unit or navigation system), which often have remote wireless interfaces to handle multiple high-performance functions, and aims to utilize the compromised ECU to influence the functionality of a different (typically safety-critical) ECU in the vehicle without being detected by any deployed network security mechanisms, e.g. IDSes.

In this situation, ad adversary can use the techniques presented in this thesis to reliably perform link-layer attacks from the compromised ECU to the target ECU, e.g. performing a selective bus-off attack, given the following conditions:

1. The remotely compromised microcontroller can be reprogrammed by the adversary
2. The remotely compromised microcontroller is equipped with an on-chip CAN controller, also known as *CAN peripheral*
3. The pins that connect the CAN peripheral to the bus can also be accessed by other peripherals on the same microcontroller

Requirement 1 is necessary for any change in the microcontroller's behavior, hence we consider it a consequence of the microcontroller being compromised and take it as granted by definition. This assumption is also consistent with prior work [6] [7] [21] [3].

Requirement 2 is verified by the vast majority of modern automotive microcontrollers: since the CAN standard has been defined in the 1980s, microcontroller vendors have had plenty of time to develop and refine on-chip peripherals that handle CAN communication, and, on the other side, having a microcontroller with plug-and-play CAN connectivity makes it more desirable for ECU vendors, as they have fewer components to route and more integrated software support for CAN applications.

Requirement 3 might seem the most restrictive one, but in modern microcontrollers it is very common that different peripherals share the same pins. This has been done to pack more and more functionality inside microcontrollers that have to handle increasingly heterogeneous applications while staying as general-purpose as possible. It also enables embedded systems engineers to have more choice over where to put each external device or connector when designing and routing PCBs that contain the microcontroller.

Microcontroller	Vendor	# CAN devices	Conflicting Peripherals
V850ES/JC3-H	Renesas	1	UART, I2C
MPC5554	NXP	3	SPI
AT90CAN32	Atmel	1	Timer
SPC564A80B4	ST Microelectronics	3	SPI, eSCI
C8051F50x	Silicon Labs	1	SPI, I2C, LIN
TMS570LS313	Texas Instruments	3	Unknown

Table 3.1: A list of peripheral conflicts found in some popular automotive microcontrollers.

As a demonstration of our claims on Requirements 2 and 3, we considered the list of automotive MCUs recommended by *Digi-Key* [8], one of the top global electronics distributors in the world [23]. Each of the 6 microcontrollers cited has at least one on-chip CAN peripheral, and conflicts with SPI, I2C and serial peripherals are common to most of the sample. Table 3.1 lists the number of CAN peripherals and the types of conflict found on these microcontrollers.

As a final remark, it is important to note that one of the main design goals of our solution is specifically to develop multiple techniques that use different peripherals, so that these can be mixed together to match the exact hardware setting of the intended target, which improves the probability of Requirement 3 to occur.

3.5 Bitbanging Improvements

As discussed in Section 2.3.3, a standard way of performing link-layer manipulation is *bitbanging*, which consists in accessing the logical levels of the bus pins from software. Practically, this means reading and writing some memory-mapped registers, in which the voltage of the pin is represented as a single bit: '0' (low voltage) or '1' (high voltage).

The most challenging task when bitbanging a high-speed protocol such as CAN is keeping the communication synchronized with the other bus participants. Software techniques can in fact suffer from timing glitches and jitters due to the varying CPU load and the preciseness of the underlying clock, which means that the sampling or writing operation performed at each bit can happen with slightly different timings. This ultimately results in bits being read or written on the bus either too soon or too late with respect to the nominal bitrate, causing errors.

Moreover, CAN is designed to work smoothly in noisy environments and with potentially slight drifts in the transceivers' clocks, so even ensuring precise timing from the software side does not guarantee a perfect synchronization.

To overcome this problem, we can rely on the CAN protocol's built-in synchronization mechanism, which forces each transceiver to resynchronize on each falling edge. A detailed description of this aspect of the protocol is provided in Section 2.1.4, but it can be summarized as the fact that, at the start of each bit time, the CAN device will briefly wait for a falling edge to be detected on the *CANRX* line. If this falling edge is detected, the point in which the signal is sampled is slightly anticipated or postponed, in order to match the new bit start.

This property can be used to improve both reading and writing in bitbanging techniques with some minor adjustments.

In particular, the reading bitbanging routine can be improved by mimicking this behavior in software. This implies taking into account rising and falling edges in the incoming packet to reset the internal timer used for the communication. This technique is used for example in CANHACK[36].

Writing, on the other hand, can be improved by artificially injecting a brief falling edge at the start of each bit, which causes a *soft resynchronization* of all transceivers listening on the bus. This means that, even if the CPU clock is slow or glitchy, the receiving devices will be able to keep up with small clock differences by resetting their internal timer at each bit. This technique is, to the best of our knowledge, a novelty in the context of CAN bitbanging techniques.

A more in-depth description of the aforementioned methods to improve standard CAN bitbanging techniques can be found in Section 4.3.

3.6 The Conflicting Peripherals Approach

Even if bitbanging techniques can be improved in various ways, they still have many weak points. First of all, they are of no use when the clock of the microcontroller is too slow, and secondly, if a hardware timer is used, they can generate a high interrupt load on the CPU, which might introduce a considerable overhead and makes the reliability of this technique highly dependent from the other tasks that the CPU is carrying out.

To avoid these limitations, we introduce a set of new techniques, based on the idea of *conflicting peripherals*.

Many microcontrollers nowadays are packed with on-chip peripherals, which often need more signals than the available physical pins. As a consequence, many

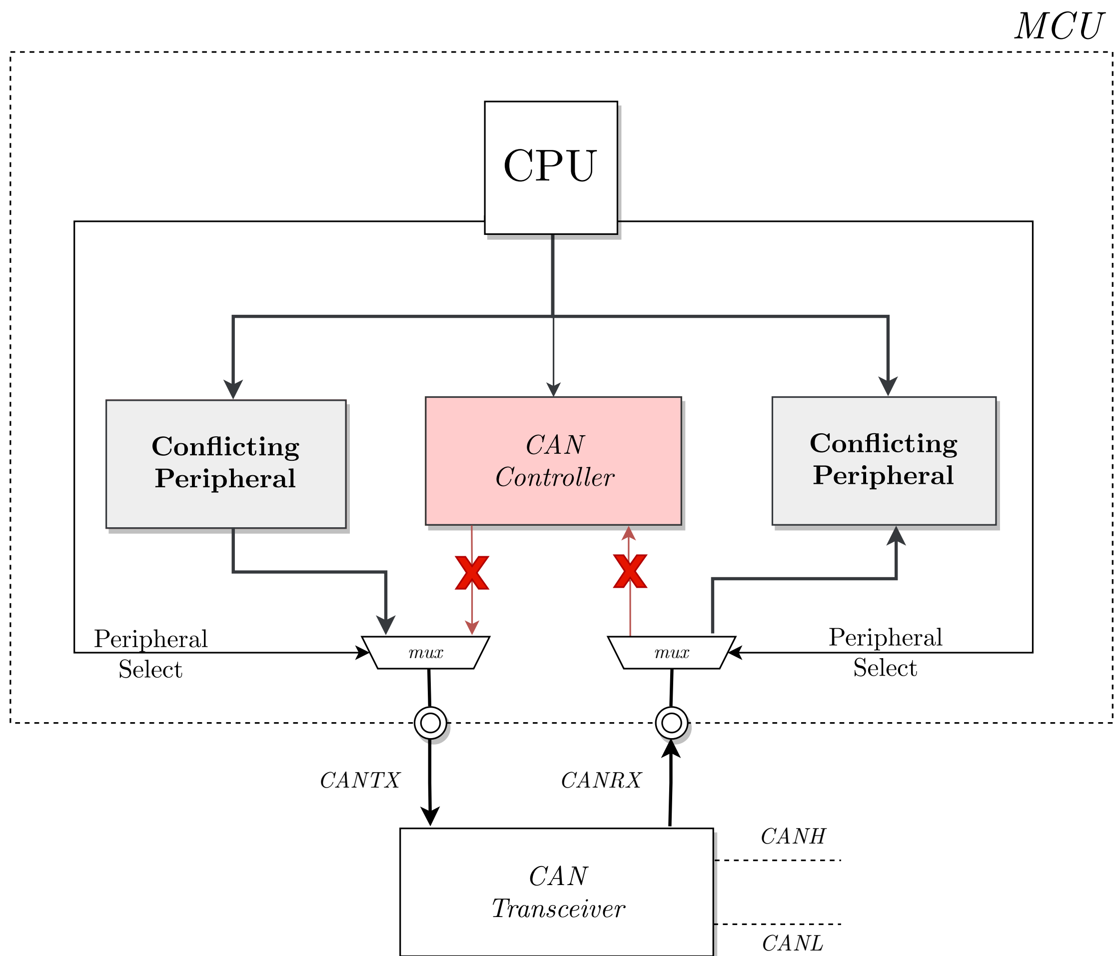


Figure 3.4: The *Conflicting Peripherals* approach

peripherals share the same set of pins, that are internally multiplexed and redirected to the chosen peripheral. The most common peripherals found on any microcontroller, from high-end automotive ones to low-power, simpler ones, are by far I2C, SPI and UART peripherals, as well as ADCs and DACs.

Most microcontrollers are equipped with multiple instances of such commonly-used peripherals, and sometimes offer the possibility to configure which pins are used by each specific device. This means that these peripherals end up sharing some or all of their signals with many other peripherals, including the CAN controller.

With this insight, we can use conflicting peripherals to generate or read CAN bus activity, and, provided that we can translate a message from one protocol to the other, effectively read and write CAN messages without using the CAN peripheral. A schematic description of this solution is provided in Figure 3.4.

By using this approach, we can benefit from all the advantages that come with hardware peripherals: low CPU load, high timing precision, high clock frequencies, and more overall reliability.

The specific peripherals chosen for this purpose are described in the next paragraphs.

Note that, since the piece of hardware that enables software control of a pin's logical level is a peripheral itself, commonly known as the *GPIO* peripheral, we can consider bitbanging a special case of the conflicting peripherals approach, in which we are exploiting the conflict between the GPIO peripheral and the CAN TX/RX signals.

3.6.1 SPI

Protocol Description

The SPI protocol is a master-slave serial protocol that typically employs four lines:

1. a clock signal (*CLK*), which is generated by the master and sets the bit timing of the communication

2. a transmission line (*MOSI*), where bits are sent from the master device to the slave device

3. a receiving line (*MISO*), with which the slave device can send bits to the master device

4. a chip select line (*CS*) for each slave, which in normal SPI communication is used by the master to select which slave it wants to communicate with

Figure 3.5 represents the timing diagram of a typical SPI communication.

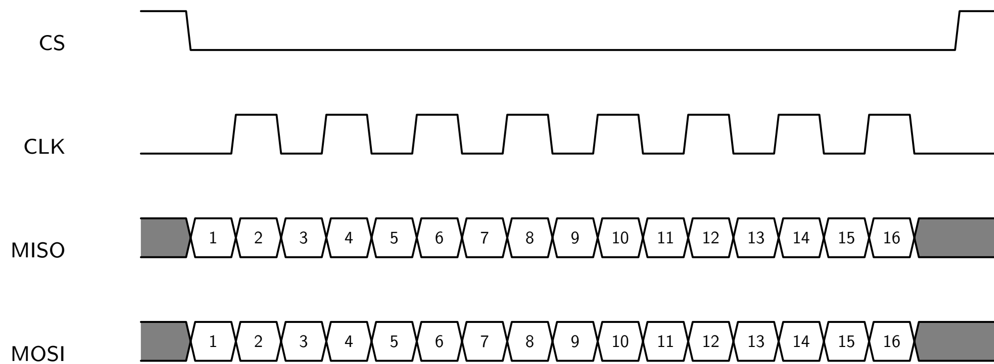


Figure 3.5: Timing diagram of an SPI message.

When in master mode, the SPI device automatically takes care of selecting the right slave, generating the clock signal, sending bits on the MOSI line, and reading bits on the MISO line.

Communication from slave devices to the master device is also initiated by the master, which decides which device can communicate on the *MISO* line by setting the *CS* line of the corresponding slave low and generating the clock signal.

The protocol does not define any intrinsic limitation on the shape of the packets that can be sent and received by the master.

Finally, SPI devices typically expose some mechanism to modify the *Clock Polarity* and *Clock Phase* of the signal, which determine how the bits are encoded during the communication. These details are out of the scope of this work, as their setting might be different among different devices, and the user manual of the specific microcontroller or SPI device in use should be consulted for a complete understanding of these settings.

Conflict Exploitation

In order to employ the SPI peripheral as a sender or receiver device on the CAN bus, we can exploit the fact that, in the SPI protocol, all communications are initiated by the master device. Therefore, setting up the SPI peripheral to behave as an SPI master is enough to have complete control over the bus communication.

Since no specific rules apply when transmitting SPI packets from a master device, other than generating the clock signal, any bitstream that is provided to the peripheral will be transmitted as-is on the MOSI line. This means that, if the *MOSI* line conflicts with the *CANTX* signal coming out from the microcontroller, the SPI peripheral can be used to send an arbitrary number of bits of the bus.

Similarly, if the *MISO* line is sharing the same pin as the *CANRX* signal, we can force the SPI peripheral to start reading the incoming packet by simply activating it in master mode.

Section 4.4.1 provides a detailed description of how this is done in the CANPass framework.

Note that, since the *CS* line and the *CLK* signal are generated automatically by the SPI peripheral, we can completely ignore their presence since neither of these signals is relevant for sending arbitrary bits on the CAN bus or is ever read by the SPI device.

Figure 5.2a shows an example of how a sequence of bits transmitted by the SPI peripheral can be interpreted both as a CAN message and an SPI message.

The conversion from and to SPI packets must be done taking into account that bitstreams that are read and written with the SPI peripheral contain every single bit transmitted on the bus, including the start bit and stuff bits.

3.6.2 UART

Protocol Description

The UART protocol is another very common serial protocol used in many embedded applications. Different from SPI, the UART protocol does not use a clock signal to synchronize the transmitter and receiver devices; it transmits data asynchronously.

Similar to the signals coming out of the CAN peripheral, the two main signals of a UART peripheral are the transmission line (*TX*) and receiving line (*RX*).

In the UART protocol, each packet must have a predefined form, which consists of a start bit, data frame, a parity bit, and stop bits, as summarized in Figure 3.6.

- The *start bit* is always '0'
- The *data frame* content can be 5 to 9 bits long
- The *parity bit* is optional and is used for error detection; it represents the parity of the number of '1's in the data frame
- The *stop bit(s)*, which consist of one or two consecutive logical '1's, depending on the peripheral configuration

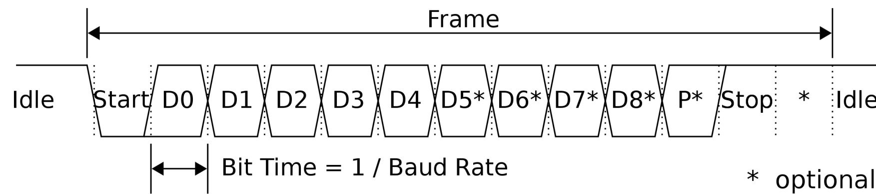


Figure 3.6: Timing diagram of a UART message.

Source: https://commons.wikimedia.org/wiki/File:UART_Frame.svg

Conflict Exploitation

Employing a UART peripheral to emulate CAN bus activity is considerably more complex than using SPI for the same purpose.

Since the UART protocol does not provide a separate communication line for the clock signal, synchronization is done by enclosing each packet between a *start bit* and one or more *stop bits*. These special bits are expected to have a specific value, i.e. the start bit of each packet must be '0' and the stop bits must be '1'.

One important characteristic of UART packets is that they can contain at most 8 bits as a payload. Therefore, since UART packets are so small, sending and receiving long CAN messages through the UART interface must be done over multiple consecutive packets.

Here, the start and stop bit values come into play, since the presence of multiple consecutive UART packets forces the resulting CAN message to have predefined values at precise locations, corresponding to the start and stop bits of each UART packet. Figure 5.2b shows an example of this, illustrating how the bits of a CAN message composed of multiple UART packets are interpreted by the UART peripheral.

It should be noted that, since our goal is to communicate on the CAN bus, the bitstream handled by the UART peripheral must not only comply with the UART protocol rules but also with the CAN protocol rules. For instance, the value of the CRC field in CAN messages is not arbitrary: it contains the CRC calculated over all the preceding fields, therefore it is much more difficult to ensure that the UART rules are followed inside this field. Another example is stuff bits, which force the message to have '1's and '0's in specific locations of the message, which might be in conflict with the positions of start and stop bits of the UART packets.

All of the above-mentioned restrictions and possibly conflicting rules make sending and receiving complete CAN messages over a UART peripheral an ex-

tremely complex task. However, it is important to consider that many attack scenarios do not involve transmitting and receiving a complete message on the bus, but rather small portions. For example, it may be enough to read the ID of the incoming message and inject just a small sequence of bits on the bus. In these cases, the UART approach can be used with fewer restrictions.

For more details on the implementation of this technique, the reader can refer to Section 4.4.2

3.6.3 I2C

Protocol Description

Another popular communication protocol in modern embedded systems is I2C, which is a multi-master, multi-slave, synchronous protocol. Being a serial protocol, data is transferred bit by bit along a single wire, called the *SDA* line.

Like SPI, I2C is synchronous, so the emission and sampling of bits is synchronized by a clock signal shared between the master and the slave. The clock signal is controlled by the master.

In the I2C protocol, messages are broken up into two types of frame: an address frame, where the master indicates the peripheral to which the message is being sent, and one or more data frames, which are 8-bit data messages passed from master to peripheral or vice versa.

The two lines are called *Serial Data* (SDA) and *Serial Clock* (SCL). Data is placed on the SDA line after SCL goes low, and is sampled after the SCL line goes high. The time between clock edge and data read/write is defined by the devices on the bus and will vary from chip to chip.

The timing diagram of a typical I2C communication is provided in Figure 3.7.

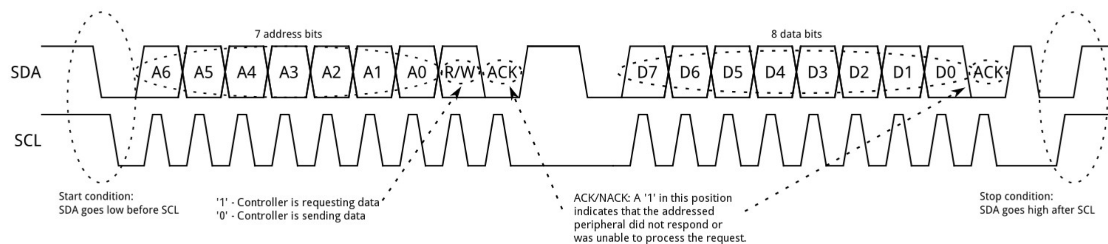


Figure 3.7: Timing diagram of an I2C message.

Source: https://learn.sparkfun.com/tutorials/i2c/all#protocol

Packets in the I2C protocol are characterized by:

- A *Start Condition*, where the SDA line is pulled low while SCL is high
- A payload, consisting of 8 controllable bits, both for address and data frames. Here, the SDA line is used for data while the SCL line is used for clocking
- An ACK slot, in which SDA is held high by the master and the slave is expected to place a '0' to indicate a positive acknowledgment
- A *Stop Condition*, where the SDA line goes high, then low, then high again after the SCL line is pulled to a high voltage

Conflict Exploitation

If a conflict between the *SDA* line and the *CANTX* line is present in the target microcontroller, this protocol can be used to send bits on the CAN bus.

However, the protocol rules are even more restrictive than UART, since the beginning and ending of each packet are signaled by a low voltage on the *SDA* line, whose duration depends on the specific device.

Moreover, each packet sent by the master is expected to be acknowledged by the intended slave in the *ACK slot*, during which the master leaves the *SDA* line in a high voltage state.

Finally, each frame in the I2C protocol is formed by multiple packets and starts with an *address* packet, in which the master communicates the address of the slave, the direction of communication (read or write), and waits for the ACK signal from the slave. If the message is not acknowledged, the communication is interrupted by the master.

All of these aspects of the protocol interfere with the necessity of sending arbitrary bits on the bus, since the voltage of the bus during the start condition, stop condition, ack slot, and inter-frame space cannot be controlled.

Nevertheless, as for UART packets, being able to control at least some of the bits that are sent on the bus can be enough to inject small sequences of bits, and, with enough knowledge of the I2C device characteristics, it is even possible to craft complete valid CAN messages, as shown in Figure 5.2c.

3.6.4 ADC

Analog-to-Digital Converters (*ADC*s) are the last type of peripherals examined in this section. The capabilities of such peripherals may strongly vary among different platforms and vendors, but the basic idea is that these peripherals can be used to perform fast and repeated analog-to-digital conversions without the

intervention of the CPU, meaning that they can be used to sample the *CANTX* signal as if it was an analog signal.

Many microcontrollers include one or more ADC devices as on-chip peripherals. Typically, such devices expose a mechanism to regulate the *resolution* of the conversion, i.e. how small can the difference between two analog values be before they become indistinguishable.

Clearly, since we are interested in the digital value of the bus, we can select the lowest possible resolution for these conversions, which typically also means higher sampling frequency, and then compare the result with a constant value, corresponding to half of the full-scale range of the ADC.

A detailed overview of how ADC sampling can be employed to read incoming bits on the CAN bus is provided in Section 4.4.4.

3.7 Framework Design

The CANPass framework is designed to group together all the aforementioned techniques under a common interface. It is built to be easily extensible and hide platform-specific details, such as registers location and peripherals settings, from the upper layers. In this way, it is possible to implement all CAN bypass techniques using an abstract representation of the hardware behavior, and develop hardware-specific code separately.

To achieve this goal, the framework is logically split into three different layers, each in charge of handling a different level of abstraction:

1. the *Public Interface Layer*, which provides the read and write primitives needed for manipulating the CAN link layer
2. the *Techniques Layer*, which contains all the techniques developed to bypass the CAN peripheral
3. the *Platform Layer*, which contains all the hardware-specific code

Figure 3.8 illustrates how these layers interact with each other.

A more complete description of each layer is provided in the next sections.

3.7.1 Public Interface Layer

The Public Interface Layer is in charge of defining a unified interface for reading and writing bits on the CAN bus. This interface is meant to be the main point

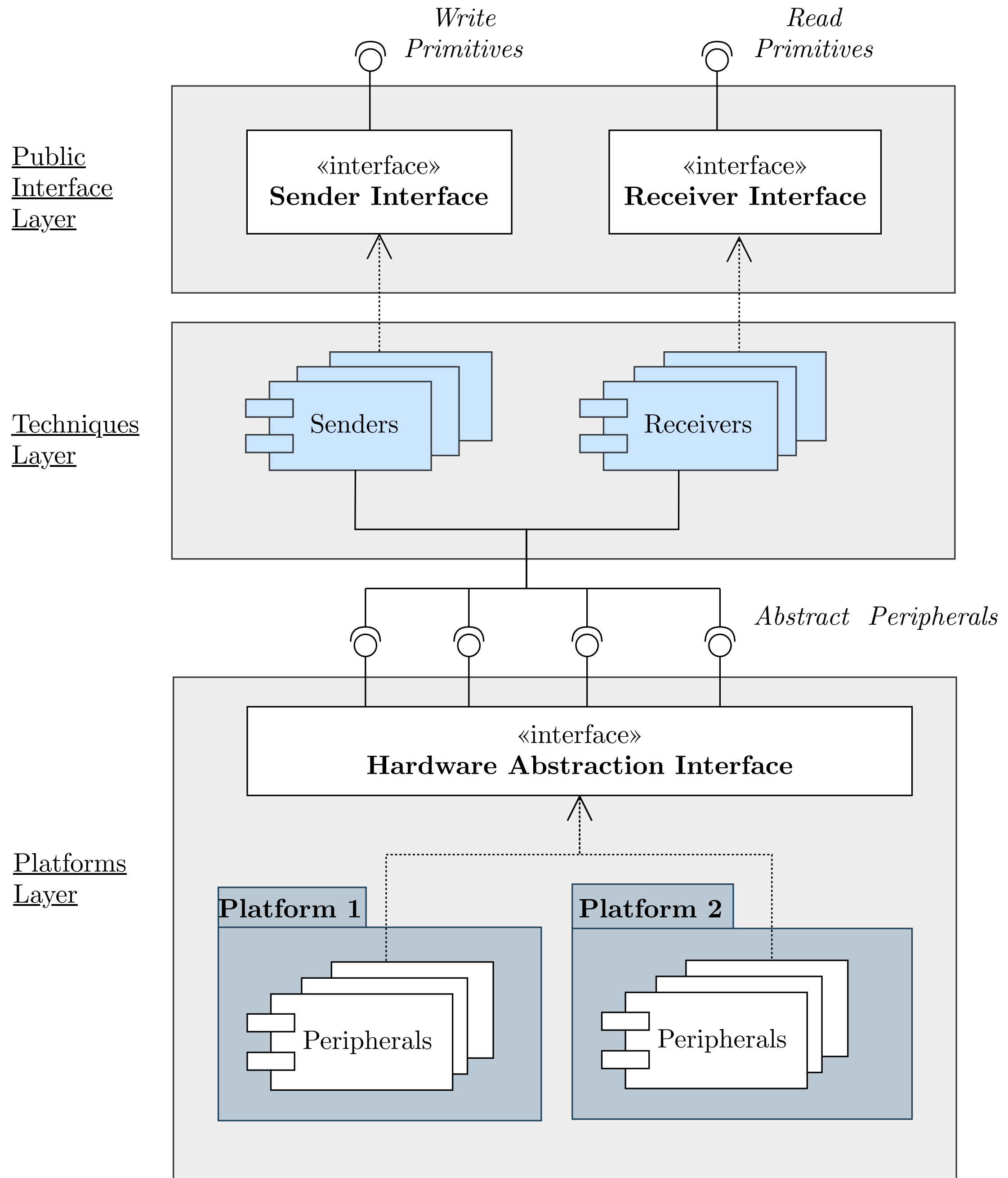


Figure 3.8: Logical layers of the CANPass framework.

of contact between user code implementing a specific attack and the underlying technique used to mount the attack.

More specifically, the reading and writing primitives are provided by two different interfaces: the *Sender* interface and the *Receiver* interface. This division serves two main purposes:

- It enables the user to use one peripheral for sending and a different peripheral for receiving bits on the bus. This means that it is not necessary to have a full conflict between a peripheral and the CAN controller in order to use its related technique, but also partially overlapping peripherals are allowed.
- It gives the possibility to implement techniques that provide only one of the two primitives, e.g. the ADC peripheral can only provide a read primitive, since its purpose is to sample incoming analog signals.

For both readers and writers, the corresponding interface provides an *initialization* and *de-initialization* primitive, which typically takes care of the initialization and de-initialization of the used peripheral(s) as well as some custom configuration that can be chosen by the user.

The Receiver Interface provides a simple *Read Primitive*, which reads a certain number of bits on the bus and returns them in a chosen buffer.

The Sender Interface's *Write Primitive* is instead divided into two steps: a preprocessing phase, in which the bitstream that the user wants to send gets encoded in a peripheral-compatible way (e.g. bing-endian bytes), and a write phase, that concretely performs the write operation. This split has the advantage of providing a standard way to concentrate the writing overhead before the writing operation is performed. In this way, the user can decide to perform the message encoding ahead of time and obtain a pre-cooked message that can be immediately injected into the bus.

3.7.2 Techniques Layer

Techniques are divided into *Senders* and *Receivers*, each of which implements one of the two public interfaces.

Each Sender and Receiver technique might use one or more peripherals to interact with the bus, for example the SPI peripheral, or the GPIO peripheral and a hardware timer.

Since we don't want to rewrite each technique for each possible platform, the Platform Layer provides a set of *abstract* peripherals that define a minimal inter-

face to interact with each peripheral. Techniques can use these abstract peripherals to ensure compatibility with all the supported hardware, which completely decouples techniques development from platform-specific code.

3.7.3 Platform Layer

Finally, the Platform Layer provides all the code relative to the interaction with the hardware. As already mentioned, the Platform Layer exposes a set of abstract peripheral interfaces via the *Hardware Abstraction Interface* layer.

Here, some of the most common peripherals are defined, such as I2C, SPI, GPIO and many others. Each abstract peripheral defines an abstract structure that is implemented by platform-specific code, through which it can be identified and passed to the related functions.

Each function in the abstract peripheral reflects a basic functionality that is expected to be common across all the devices of that type: for instance, timers should have a function to set the frequency, GPIOs should have a function to set the logical level to high or low, and so on for each peripheral.

The abstract peripherals are then implemented in the Platform code. Platforms are the final targets that will be executing the code, typically microcontrollers in our case. Different vendors provide different functionalities for each of their microcontrollers, so each platform might have a slightly different implementation for each peripheral's functionality.

Platforms are not expected to implement all the peripherals defined in the Hardware Abstraction Interface. Clearly, the more peripherals are implemented, the more techniques will be available for that platform.

Chapter 4

Implementation

4.1 General Characteristics of the CANPass Framework

Concretely, the CANPass framework is implemented as a self-contained, header-only C library that can be easily included in any embedded application. The only requirement is that the user must be already able to program the intended target, i.e. any vendor-specific toolchain, including cross-compilers, build systems and hardware-specific libraries should be obtained separately.

As already mentioned in Section 3.7, the framework is divided into three layers that decouple user code, CAN peripheral bypass techniques, and hardware implementations. This division is reflected by the folder structure of the code, described in Section 4.1.3.

The remainder of this section provides some details and justifications for various implementation choices that have been made in the realization of the framework.

4.1.1 Language

The programming language chosen for implementing the CANPass framework is the C language, in particular the *C99* standard [15], which features a few important additions to the ANSI C standard, such as `inline` functions and the `stdint` library.

The C language is by far the most commonly used language [10] for embedded applications: its simplicity and "low-level" features, such as bit manipulation and manual memory management, make it a perfect match for highly specialized, real-time applications that have to run in a highly constrained hardware environment. It is also the most commonly supported language by proprietary compilers, which are very common in embedded applications and automotive in particular.

It is worth noting that, nowadays, other alternatives to C exist for embedded software development, such as C++, Python and Rust, which provide an easier way to express abstraction without introducing excessive overhead, but the lack of support from freely available compilers, for instance in the case of TriCore architectures [35], make them less fit for our purpose. Furthermore, since C is a simple and widely studied language, especially for embedded applications, we expect a wider audience to be able to understand and contribute to the framework.

4.1.2 Building and Dependencies

To avoid any additional steps in the building process of any application already written for a specific target, the CANPass library is *header-only*, i.e. it consists only of header files.

It is important to note that, while all files of the framework are headers, only some of them are designed to be included by user code, as described in Section 4.2. In this sense, there are some internal dependencies in the framework, which are nevertheless transparent for the user.

The only external dependency that is present in the techniques' code is standard library headers, such as `stdint` and `stdbool`, which are used for better bit manipulation. Platform code, on the other hand, generally depends on vendor-specific libraries that are not provided in the framework and must be obtained separately.

From the building point of view, the main consequence of being a header-only library is that no source file has to be separately compiled and then linked to the application code: the CANPass headers can be directly included by the user and the contained functions will be available out-of-the-box.

4.1.3 File organization

The internal file organization of the CANPass framework is illustrated in Figure 4.1.

The outer folder contains the two main interface headers, called `CANPassSender.h` and `CANPassReceiver.h`. Each available technique must include one of the two headers and provide an implementation for each structure and function defined there. User code should not include directly such headers.

From a practical point of view, these files serve as a reference for the user, who can rely on the fact that the functions declared there will be implemented by every technique. Moreover, any technique that includes one of the public interface

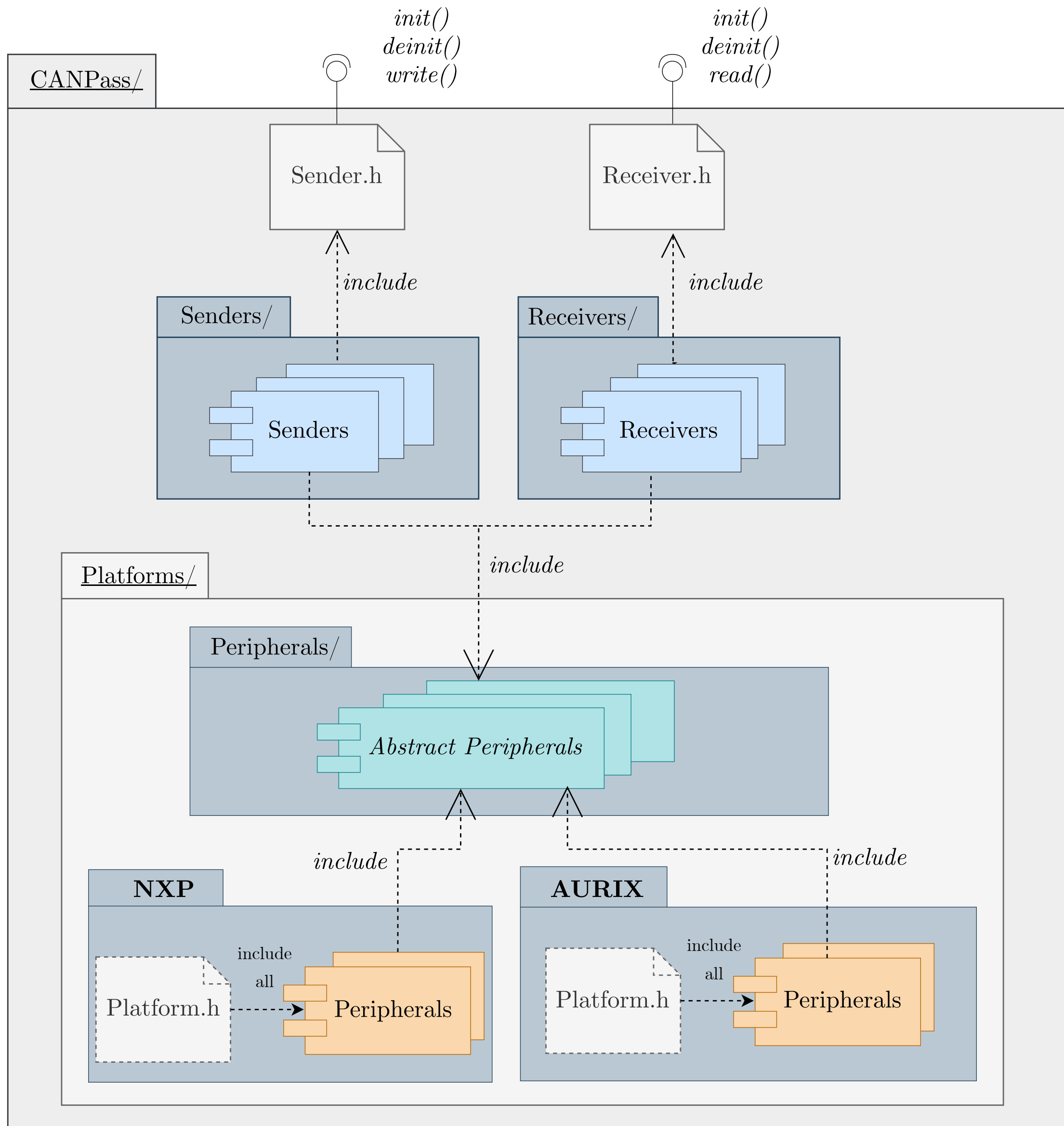


Figure 4.1: Folder organization of the CANPass framework.

headers must define *all* of the declared functions, otherwise the compiler will generate missing definition errors.

In addition to public interface headers, the top folder contains three subfolders: `senders/`, `receivers/` and `platforms/`.

Senders and Receivers are the core of the framework: each of them contains the implementation of a specific technique to read or write bits on the CAN bus, respectively. The user can choose which technique to use by including the corresponding header, as described in Section 4.2. Each sender can be included alongside any receiver and vice-versa: this is done to maximize the flexibility of the framework with respect to the specific hardware configuration of the chosen platform. However, multiple senders or receivers cannot be included in the same translation unit, as they would cause a multiple-definition error for the same set of interfaces.

Finally, platform code is divided into *abstract peripherals' files*, which declare a set of functions for each peripheral but do not provide any definition, and *platform-specific files*, which implement the above-mentioned functions for a specific microcontroller.

To ease the platform selection procedure, all peripheral headers of a specific target are gathered in a `platform.h` file. In this way, the user can swap the implementation that is being used by CANPass techniques by simply including a different `platform.h`.

As already mentioned, it is not mandatory for a specific target code to implement all supported peripherals: in this case, a missing definition error will be generated by the compiler when using a technique that is not supported by the target.

An exhaustive list of currently supported techniques and platforms can be found in Figure 4.5 and Figure 4.6.

4.1.4 Error Handling

Every function defined in the framework either returns a boolean value, indicating the success or failure of the given operation, or it returns an integer.

No exceptions or hard faults are generated, which means that the user is in charge of checking the return value of each operation and performing an appropriate action in case of errors.

4.2 Framework Implementation and Usage

When using *CANPass*, the user must choose at compile-time which technique to adopt for reading or writing on the bus. This is done by simply including the appropriate technique header in the application code. Reading and writing techniques can be chosen independently from each other, which makes this tool adaptable to various hardware configurations: for example, the user might choose to transmit on the CAN bus with the I2C peripheral and read incoming messages through the SPI peripheral instead.

The possible configurations are still bounded to which peripheral signals are conflicting in the actual target, i.e. which peripherals have pins in common. Conflicts are not hardcoded in advance inside the platform code: the user must check which signal conflicts are present in the targeted microcontroller, and set up the CANPass configuration accordingly. More details about how CANPass techniques can be used and configured from the application code are provided in the following subsections.

4.2.1 Interfaces

The interface exposed to the user is pretty straightforward, consisting of two simple *read* and *write* primitives, that take an array of bits as an argument, and *initialization* and *de-initialization* functions for taking control of the bus and releasing it.

To show how minimal the interface is, Listing 1 and Listing 2 contain the whole reading and writing interface, respectively.

It is worth noting that the *Sender* interface exposes a `setMsgToSend()` function alongside the `sendMsg()` function. The former is meant to include any preprocessing step that the technique should perform just before sending a message, which might include encoding the message in a peripheral-specific form or set some registers to prepare the peripheral.

The rationale behind this choice is that, in a typical attack, the content of the message that must be sent on the bus is known in advance, while the moment in which the sending should start is to be chosen carefully. Dividing the message preparation from the actual sending provides an easy solution for this case, since the `setMsgToSend()` function concentrates all the writing overhead and can be called ahead of time, to maximize the responsiveness of the `sendMsg()` operation.

```c
#pragma once

///\file CANPassReceiver.h
///\brief This file defines the public interface of CANPass receivers.
///
/// Each receiver must provide an implementation for all the functions and
/// structs defined here.
/// \note This file is meant to be included by receivers, not by user code.

#include <stdint.h>
#include <stdbool.h>

///\brief Configuration struct, defined by the specific receiver
struct CANPass_ReceiverConf_t;

///\brief Initialize the peripheral
///\return true if the initialization was successful
static inline bool CANPass_receiver_init(struct CANPass_ReceiverConf_t *conf);

///\brief Start recording incoming traffic on the CAN bus
///\return how many bits were effectively received
static inline uint64_t
CANPass_receiver_rcvBits(struct CANPass_ReceiverConf_t *conf, bool buf[],
                         uint64_t nBits);

///\brief De-initialize the peripheral
///\return true if the de-initialization was successful
static inline bool CANPass_receiver_deinit(struct CANPass_ReceiverConf_t *conf);
```

Listing 1: The CANPass Receiver interface.

```c
#pragma once

///\file CANPassSender.h
///\brief This file defines the public interface of CANPass senders.
///
/// Each sender must provide an implementation for all the functions and structs
/// defined here.
/// \note This file is meant to be included by senders, not by user code.

#include <stdint.h>
#include <stdbool.h>

///\brief Configuration struct.
///
///\brief Configuration struct, defined by the specific receiver

struct CANPass_SenderConf_t;

///\brief Initialize the peripheral
///\return true if the initialization was successful
static inline bool CANPass_sender_init(struct CANPass_SenderConf_t *conf);

///\brief Prepare the sender for the next message.
///\return how many bits will be sent (0 means error).
///
/// This function can be called ahead of time to minimize the overhead when
/// injecting bits.
static inline uint32_t
CANPass_sender_setMsgToSend(struct CANPass_SenderConf_t *conf,
                            const bool bitstream[], const uint32_t nBits);

///\brief Start sending the message on the bus as soon as possible.
///\return how many bits have been sent (0 means error).
static inline uint32_t
CANPass_sender_sendMsg(struct CANPass_SenderConf_t *conf);

///\brief De-initialize the peripheral.
///\return true if the de-initialization was successful.
static inline bool CANPass_sender_deinit(struct CANPass_SenderConf_t *conf);
```

Listing 2: The CANPass Sender interface.

4.2.2 User Configuration

Public interfaces and abstract peripherals each expose, alongside some functions, an abstract configuration structure that is defined differently by each technique or concrete peripheral. For example, the `CANPassSender` interface declares a structure called `CANPass_SenderConf_t`, which contains different fields depending on what information is needed by the specific technique.

In this way, it is possible to adapt the same interface to potentially very different uses, since the variables that are needed by each technique are implementation-defined.

Generally, each function accepts the aforementioned structure as a first parameter, which is then used to identify the peripheral and configure the bitrate, the peripheral pins, and other information that highly depend on the specific technique and target.

However, there are some cases in which configuration structs aren't enough to express what the user can choose, in particular in the case of interrupts, where the name of the function to implement is different for each peripheral. In this case, some preprocessor `define`s are expected to be defined by the user before including the CANPass code, as illustrated in Listing 3. The possible values of these macros are typically located in a `config.h` file inside the platform's folder.

4.2.3 Sending Example

Listing 3 shows an example of how to concretely use these interfaces. This example was tested on the NXP LPC11C24 microcontroller.

First of all, the `platform.h` file corresponding to the desired target must be included, preceded by any `#define` macro needed by the platform. For example, in Listing 3 we can see that, in order to use the hardware timer bit-banging technique on the NXP platform, we must, first of all, define which timer we are going to use, by defining the `CANPASS_LPC11_TIMER` preprocessor macro. The possible values for this macro can be found in `CANPass/platforms/NXP/config.h`. After that, we can include the platform header for the NXP microcontroller.

As a second step, the sending technique is chosen by including the corresponding header (`GPIOSender_timer.h`). After this, all sender functions listed in Listing 2 are available.

The only thing that differentiates one technique from another is the configuration struct, called `CANPass_SenderConfig_t`. Before passing it to the initialization function, the user must take care of initializing its fields to the desired values.

```c
// Define which timer to use: possible values are listed in config.h.
#include <platforms/NXP/config.h>
#define CANPASS_LPC11_TIMER CANPASS_LPC11_TIMER32_0

// Choose platform
#include <platforms/NXP/platform.h>

// Choose technique
#include <senders/GPIOSender_timer.h>

int main(void) {
  // Define which GPIO to be used for sending
  struct CANPass_GPIO_t tx_gpio;
  tx_gpio.gpioPeriph = LPC_GPIO;
  tx_gpio.port = 2;
  tx_gpio.pin = 0;

  // Define which timer peripheral to use
  struct CANPass_TIM_t tim;
  tim.timer = LPC_TIMER32_0;

  // Build the configuration structure
  struct CANPass_SenderConf_t sndConf;
  sndConf.gpio = &tx_gpio;
  sndConf.tim = &tim;
  sndConf.blocking = true;
  sndConf.freqHz = 100000;

  // Define the message to send
  const bool MSG_BITS[8] = {0, 1, 1, 1, 0, 1, 1, 1};

  // Initialize the CANPass sender with the configuration struct
  CANPass_sender_init(&sndConf);

  while (1) {
    // Send the message with CANPass
    CANPass_sender_setMsgToSend(&sndConf, MSG_BITS, 8);
    CANPass_sender_sendMsg(&sndConf);
  }
}
```

Listing 3: An example of application code for the NXP LPC11C24 micro-controller that uses CANPass to perform bitbanging with a hardware timer.

Finally, the `setMsgToSend()` and `sendMsg()` operations are used to concretely send the message on the bus.

4.2.4 Receiving Example

The receiving interface is fairly similar to the sending one, except that no preprocessing is needed before starting the receive operation.

Platform choice and configuration are performed in the same way described in Section 4.2.3, as well as technique choice.

Listing 4 shows a concrete example of reading the CAN bus using the SPI peripheral. The bus is sensed through the GPIO peripheral until a falling edge is detected, which signals the start of a CAN packet, then the SPI peripheral is activated to read the first 16 bits of the message and store them in a chosen buffer.

4.2.5 Adding a New Technique

Sending and receiving techniques are the core of CANPass, and are logically interposed between the user code and hardware peripherals.

Any technique inside the `senders/` and `receivers/` folder can be used as a reference for developing new techniques.

To create a new technique accessible from the user code, the first step is to create a new header file inside the `senders/` or `receivers/` folder, depending on the goal of the technique. If the technique can be used for both sending and receiving, the two capabilities must be implemented in two separate files.

Once the technique file is created, the right public interface header must be imported, i.e. either `CANPassSender.h` or `CANPassReceiver.h`. Since these two headers consist of forward-declarations only, all the contained structs and functions must be implemented, otherwise, a compiler error will be generated. This ensures that every technique complies with the public interface.

After including the public interface header, each technique must include the header of the peripheral(s) it wants to exploit for communicating on the CAN bus. Techniques are expected to be hardware independent, therefore they should use *abstract peripherals*, which are located in the `platforms/peripherals/` folder.

Once the correct peripherals have been included, the configuration struct must be defined, i.e. `CANPass_SenderConf_t` for senders and `CANPass_ReceiverConf_t` for receivers. This struct must contain all the information needed to configure and initialize the hardware peripheral, which typically means at least a pointer to the selected peripheral and the desired bitrate.

```c
/*
 * \brief Example of how to use the SPIReceiver on the TC399XP TriBoard.
 */
// CANPass headers
#include <platforms/AURIX/platform.h>
#include <senders/SPIRecei.h>

#define CAN_RX_PIN (&IfxCan_TXD00_P20_7_IN)
#define MSG_BIT_LEN 112
#define N_SAMPLES 1
#define CAN_BAUDRATE 1000000 // = 1 MBits/s

extern IfxCpu_syncEvent g_cpuSyncEvent;

void core4_main(void) {
  IfxCpu_enableInterrupts();
  IfxScuWdt_disableCpuWatchdog(IfxScuWdt_getCpuWatchdogPassword());

  IfxCpu_emitEvent(&g_cpuSyncEvent);
  IfxCpu_waitEvent(&g_cpuSyncEvent, 1);

  // CANPass Setup
  struct CANPass_SPI_t spi;
  spi.spi = &MODULE_QSPI0;
  spi.clkPin = &IfxQspi0_SCLK_P22_8_OUT;
  spi.mosiPin = &IfxQspi0_MTSR_P22_5_OUT;
  spi.misoPin = &IfxQspi0_MRSTC_P22_6_IN;
  spi.chipSelectPin = &IfxQspi2_SLSO0_P15_2_OUT;

  struct CANPass_ReceiverConf_t conf;
  conf.spi = &spi;
  conf.freqHz = CAN_BAUDRATE;
  conf.nSamples = N_SAMPLES;

  static const bool msg_bits[16];

  CANPass_receiver_init(&conf);

  // Read with CANPass
  while (1) {
    while (IfxPort_getPinState(CAN_RX_PIN) == 0)
      /* Wait for falling edge */;
    CANPass_receiver_rcvBits(&conf, msg_bits, 16);
  }
}
```

Listing 4: An example of application code for the AURIX TC399XP microcontroller that uses CANPass to read CAN messages through the SPI interface.

Non-Blocking Implementations

As for the current version of the framework, all techniques have been implemented as *blocking* functions, i.e. the read or write function invoked from the user code will not return to the caller until the operation has been carried out completely.

In future implementations, the configuration struct might contain a pointer to the buffer to be used, as well as a counter for the current bit that is being read or written. These can be used to implement *non-blocking* techniques, i.e. techniques in which the sending/receiving operation is performed in parallel to CPU instructions, therefore allowing the software to perform other operations while the hardware peripheral is active. In this case, the counter field of the configuration struct can be used to check the state of the hardware peripheral and verify when the desired operation has been completed.

4.2.6 Adding a New Platform

Platforms represent a specific microcontroller or board supported by CANPass. All the code for each platform is contained in a separate folder, and a `platform.h` file is generally provided to include all the platform code at once.

Implementing a new platform can be done incrementally, adding one file for each supported peripheral. Each peripheral file must include the corresponding abstract peripheral header and implement all structures and functions declared there. Since abstract peripherals are designed to be as minimal as possible, implementing their capabilities should be straightforward in the general case.

4.3 Bitbanging Implementation

4.3.1 Reading Improvements

As discussed in Section 2.3.3, the main way in which the CAN link layer is accessed by current techniques is through bitbanging. A typical robust bitbanging implementation uses a hardware timer to execute a certain routine at a fixed period, e.g. every 1μs if the bitrate of the bus is 1 Mbit/s.

However, this technique is very sensitive to the timer's initial offset, and relies on the fact that the timer's resolution can exactly match the requested interval. If, for example, the minimum interval available for a specific hardware timer is not 1 μs but 1.1 μs, after 10 bits the cumulated error will be a full 1 μs, leading to a desynchronization with the other devices on the bus.

This phenomenon is called *clock drift*: an example is shown in Figure 4.2, where the signal is sampled every time the hardware timer expires, i.e. when the internal counter of the timer reaches a given threshold. In this case, it can be easily seen how the cumulated error due to clock drifts can cause the receiver to completely skip one bit sent on the bus.

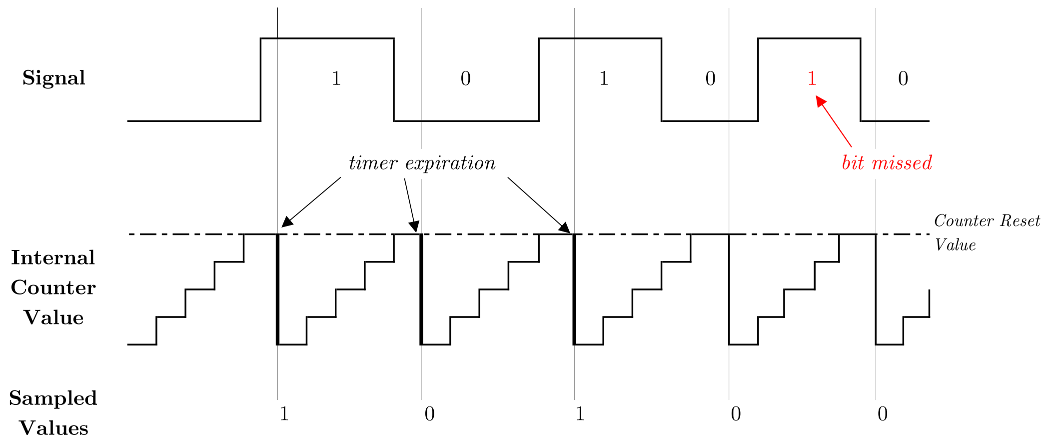


Figure 4.2: An example of how timer drifting might affect reading.

A solution to this problem is *timer resynchronization*. This technique consists in detecting the rising and falling edges in the incoming CAN signal, either making the CPU check the current pin level at each clock cycle or setting up an interrupt on the GPIO's level change. Whenever an edge is detected, the timer must be forced to expire, effectively resetting its state and canceling out the accumulated clock drift, as illustrated in Figure 4.3.

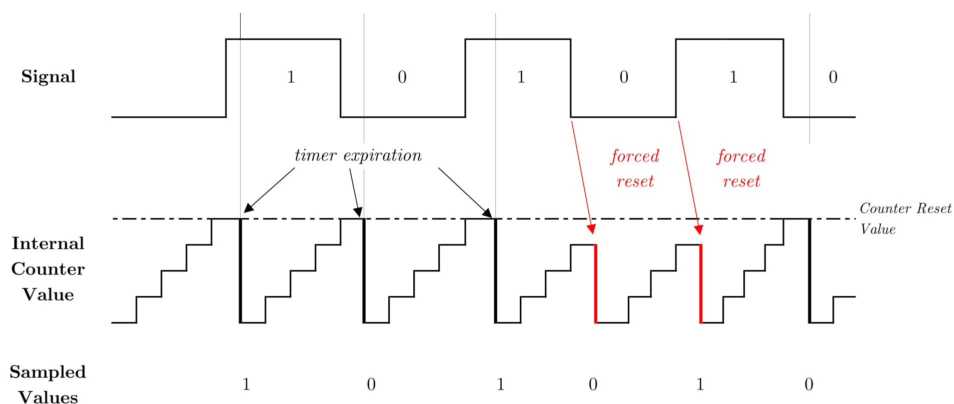


Figure 4.3: Resetting the timer on rising/falling edges improves synchronization.

In this way, even with a considerable timer drift, the hardware timer will be resynchronized every time a '0' is transmitted on the bus after a '1' or vice-versa.

The only drift accumulated is during sequences of consecutive '1's and '0's, but since CAN messages can have at most 5 consecutive equal bits, because of the bit-stuffing rule, we can theoretically have a hardware timer that is 20% slower than the bus baudrate and still be able to correctly read bits on the bus.

4.3.2 Writing Improvements

Writing techniques based on bitbanging have the same drifting problems as the reading techniques just described.

More notably, if the timings of a transmitted message are incorrect, the receivers on the bus might read a different message than the one being sent, therefore detecting bit stuffing or CRC errors that were not present in the original message. This causes the transmission to fail and error messages to be signaled on the bus by receivers.

To avoid this situation, we can rely on the following properties of the CAN protocol:

- If a falling edge is detected during the *synch* segment at the start of each bit, the device must resynchronize considering the moment in which the falling edge was detected as the start of the bit
- Only the first falling edge detected during the synchronization segment will cause the device to resynchronize
- The value on the bus is sampled at the *sampling point*, which is located at the end of the bit; typical values of the sampling point are 70-80% of the bit time

The fact that resynchronization on falling edges is mandatory for all devices and causes the sampling point to be moved further, effectively stretching the bit time, can be exploited to increase the reliability of bitbanging writing techniques in case the available timer is noisy or slower than the requested baudrate. This can be done by inserting an artificial falling edge at the start of each bit, which will cause the devices to reset their internal timers.

Algorithm 2 describes this procedure in case the bit to send is a '0' or a '1': in the first case, the signal is briefly held high before returning to low voltage, while in the second case we must inject a brief high signal followed by a brief low signal to ensure the falling edge, then the signal is pulled high to communicate the '1'.

Algorithm 2 Pseudo-code for forcing a falling edge when sending a bit.

1: **procedure** SEND(BIT):
2: **if** $Bit == 0$ **then**
3: $BusWrite(1)$
4: $BusWrite(0)$ // falling edge
5: **if** $Bit == 1$ **then**
6: $BusWrite(1)$
7: $BusWrite(0)$ // falling edge
8: $BusWrite(1)$

An example of the shape of the resulting CAN TX logical signal is illustrated Figure 4.4.

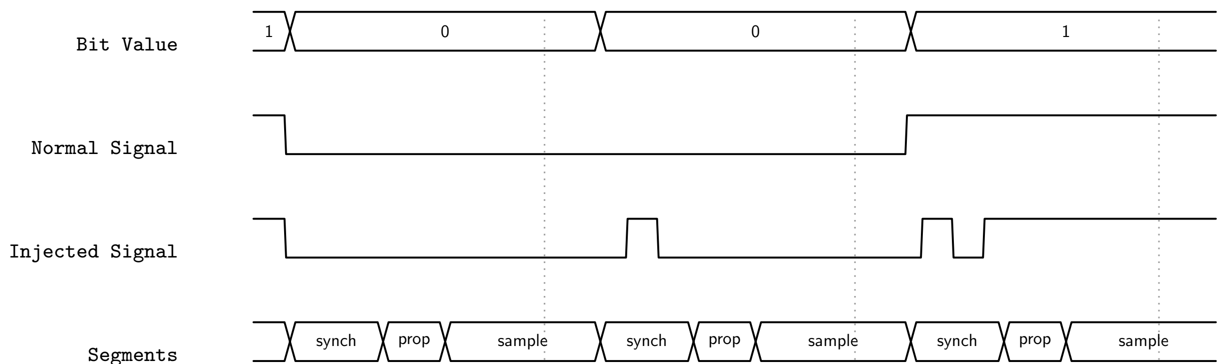


Figure 4.4: Timing diagram of a CAN signal with injected falling edges.

With this method, we also exploit the fact that, after the injected falling edge, no other edge will be detected until the sampling point, which means that we can artificially introduce a falling edge on each bit without the risk of compromising the value being read.

Note that, while this technique can effectively improve the probability that a given bitbanged message is correctly received by a legitimate CAN device, it might be easily detected by any Intrusion Detection System that can analyze the voltage patterns on the bus, and is therefore not suitable for situations in which the injection must be stealthy.

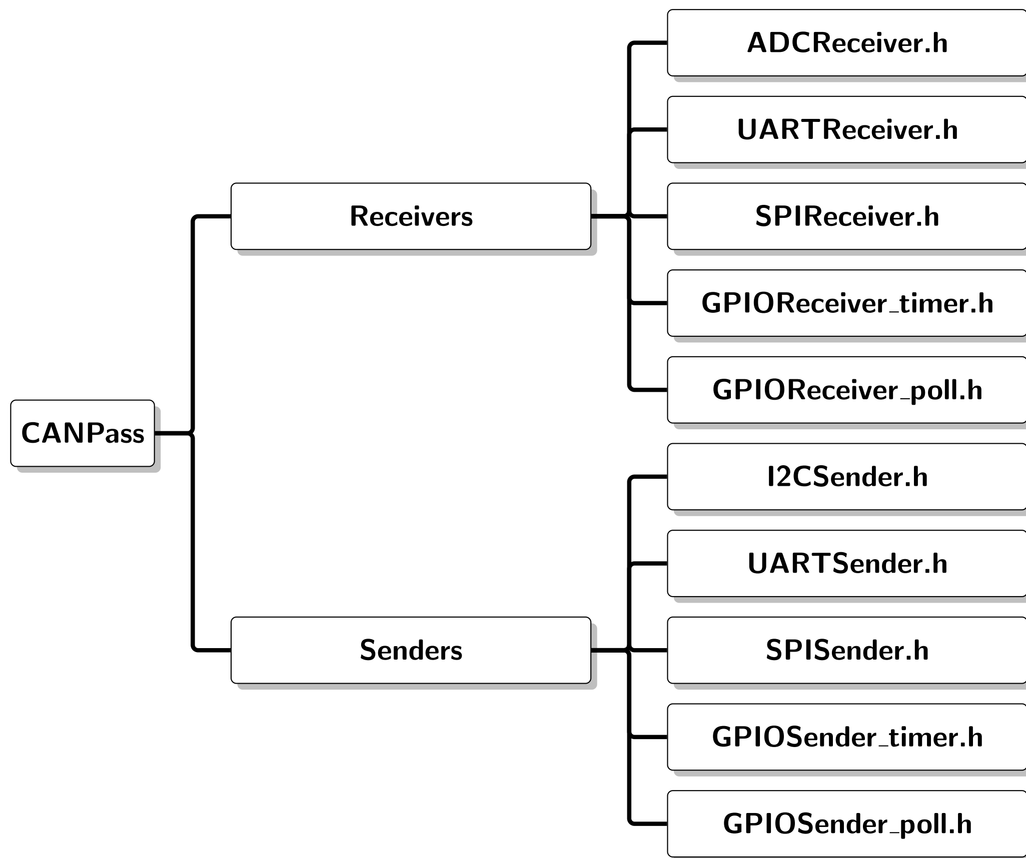


Figure 4.5: List of techniques included in the *CANPass* framework.

4.4 Techniques Implementation

To overcome the limitations intrinsic to bitbanging, each peripheral conflict described in Section 3.6 can be exploited to either write or read arbitrary bits on the CAN bus in a significantly more reliable way. The CANPass framework provides an implementation for each reading and writing technique that employs the aforementioned peripheral conflicts to communicate on the CAN bus.

Figure 4.5 contains a list of all the techniques included in the framework. Each technique makes use of one or more of the *abstract peripherals* provided by the framework, which are listed in Figure 4.6.

The next sections describe in detail how each technique has been implemented. Each peripheral's interface is briefly described, to ease the reader's understanding of how it might be used to communicate on the CAN bus.

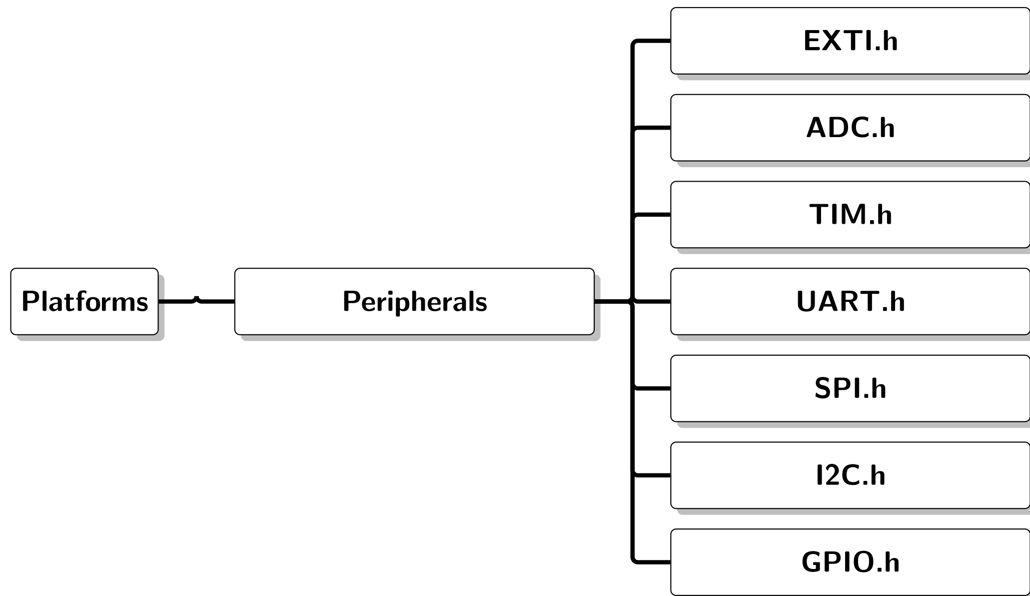


Figure 4.6: List of peripherals provided by the *Hardware Abstraction Interface.*

4.4.1 SPI

Peripheral Primitives

The SPI peripheral modeled in the Hardware Abstraction Interface exposes the following primitives:

- `setBaudrate()`: sets the speed at which the SPI peripheral sends and receives bits. This should be equal to the expected CAN bitrate when communicating on the CAN bus.
- `read()`: starts an SPI *read* operation as if the microcontroller was a master of the SPI bus: the SPI peripheral generates a clock signal on the *CLK* line and reads the *MISO* line at every falling edge of the clock. The bits are stored in the provided buffer.
- `write()`: starts an SPI *write* operation as if the microcontroller was a master device on the SPI bus: the SPI peripheral generates a clock signal and sends a bit on the *MOSI* line at each clock tick. Typically, SPI peripherals expect the software to put the data of the message to send in an 8-bit data register in big-endian form. After the data register is read, the software can overwrite its content with the next byte to send while the SPI peripheral is sending the current byte.

Given this interface, emulating CAN communication using an SPI peripheral is quite simple: the SPI peripheral behaves as if it was operating on a normal SPI bus, but input and output bits are physically redirected on the CAN pins.

Also, the fact that the SPI peripheral is used as a master device means that there are no particular limitations for when the peripheral can start reading or writing.

Reading

If the *MISO* line of the SPI peripheral and the *CANRX* line of the CAN peripheral share the same pin, the incoming signals from the CAN bus can be redirected to the SPI peripheral.

In CANPass, this task is carried out by the `SPIReceiver` technique, which is in charge of taking control of the *CANRX* signal and reading the bus levels at the desired bitrate through the SPI peripheral. The technique configuration must be initialized by selecting the appropriate baudrate, which is the frequency at which the bus is sampled, and the target's SPI peripheral in which the conflict occurs.

Once the `SPIReceiver` has been initialized, which causes the SPI peripheral to take control over the *CANRX* pin, the software can start the `read()` operation at any time.

Typically, in order to read a CAN message from the start, the application code needs to wait for a falling edge to occur on the bus, corresponding to the SOF bit. When the falling edge is detected, the SPI peripheral can be activated to read each bit at a fixed bitrate. In order to provide maximum flexibility, falling edge recognition is not done directly by the technique, therefore it should be carried out by the application code before starting the `read()` operation. Application code can also decide to start reading at fixed intervals, in which case no falling edge recognition is needed.

Note that, since the `read()` primitive accepts the length of the message to read as a parameter, the software must decide in advance how many bits to read.

Writing

Similarly, writing on the CAN bus from the SPI peripheral is possible if the *MOSI* line of the SPI peripheral and the *CANTX* line of the CAN peripheral share the same pin. The CANPass technique in charge of performing writing through the SPI peripheral is called `SPISender`.

Since SPI peripherals generally send messages byte-by-byte, the `setMsgToSend()`

function is in charge of encoding the provided bitstream into a sequence of correctly encoded bytes. The encoding procedure can be described as follows:

1. Divide the bitstream into bytes
2. Encode each byte in the big-endian form
3. Pad the last byte with '1's (recessive bit) if needed
4. Perform the send operation byte-by-byte

Finally, since SPI peripherals can generally reach a very high bitrate, the `SPISender` also provides a way to perform *oversampling*, i.e. having the SPI peripheral run at a higher frequency than the bus. In this way, each bit length can be controlled at a sub-bit duration level.

4.4.2 UART

Peripheral Primitives

The UART peripheral modeled by the Hardware Abstraction Interface of the CAN-Pass framework exposes the essential primitives needed for communicating on the CAN bus. The provided functions are:

- `setBaudrate()`: sets the baudrate of the peripheral, should be equal to the CAN bus baudrate.
- `configure()`: lets the user adjust two additional parameters: the `wordLen` parameter, i.e. the *length* of the payload of each packet, which can range from 5 to 8 bits, and the `stopBits` parameter, which represents the number of *stop bits* at the end of each packet and can be either 1 or 2. Other functionalities, such as parity bits and CTS/RTS signals, are disabled and cannot be modified by the user code, since they serve no purpose in the context of CAN signal emulation.
- `read()`: starts the *read* operation: any following falling edge will be treated as the start of a new message, which is expected to contain `wordLen` bits and end with 1 or 2 stop bits with the logical value '1'. The result of a single read operation is a byte containing the payload of the packet. As start and stop bits have constant values, the corresponding bit can be directly inferred by the software.

- `write()`: given a sequence of bytes, this function performs a sequence of write operations interpreting each byte as the payload of a UART message. The peripheral itself will insert start and stop bits at the beginning and end of each transmitted packet, therefore these should be omitted when using this primitive.

Reading

If the *RX* line of the UART peripheral conflicts with the *RX* line of the CAN peripheral, we can read CAN packets using the former. This task is carried out by the `UARTReceiver` technique in the CANPass framework.

The incoming CAN packets must have some specific characteristics in order to be recognized as valid packets:

1. The first bit must be '0', which is already the case in the CAN protocol.
2. The subsequent 5 to 8 bits, depending on the configured `wordLen`, can have any values, and will be stored by the peripheral as the result of a single read operation.
3. The following 1 or 2 bits, depending on the value of the `stopBits` configuration, must be '1'. If not, the peripheral will discard the incoming message as faulty.
4. The following bit must be a '0', which forces the peripheral to listen for a new packet.
5. The following 5 to 8 bits are read as before.
6. So on until the end of the message.

As we can see, the technique is not fit for reading any long CAN message, but might be enough for reading the ID of a message, as long as it follows the aforementioned format.

Since in real-world CAN systems many messages have a fixed period, this technique can be fired at fixed intervals to check the content of the bus at a specific moment, which could be the message ID but also inside the message's payload.

Writing

If the *TX* line of the UART peripheral conflicts with the *TX* line of the CAN peripheral, it can be used to send CAN messages.

The same formatting rules that have been highlighted for receiving also apply to messages being sent. In this case, the peripheral itself sets the voltage of the bus at the desired level during start and stop bits. Since a single CAN message cannot be enclosed in a single UART packet, multiple successive packets are needed to create a CAN message.

The `UARTSender` technique is in charge of checking that a given bitstream is fit for being sent on the UART peripheral, i.e. that every sequence of `wordLen` bits starts with a '0' and ends with a '1'. It also takes care of splitting the bitstream to send into a sequence of UART packets, which will then be written on the bus by the UART peripheral one after the other, recreating the original bitstream.

As previously mentioned, the coexistence of UART protocol rules and CAN protocol rules in the same packet severely limits the set of messages that can be sent with this technique. It is nevertheless possible, given a fixed part of the message that we want to send, to craft a sequence of bits that is both a valid CAN message and a valid sequence of UART packets by bruteforce.

This procedure can be summed up as:

1. Take the message that needs to be sent on the CAN bus
2. Choose a part that is fixed. It must not already violate the UART rules.
3. Force the bits that correspond to UART start and stop bits to be 0 and 1, respectively.
4. Randomly assign the other bits of the CAN message (except for the CRC field).
5. Check the message validity, i.e. that the resulting CRC does not break the UART rules and that the generated bitstream follows the bit-stuffing rule (no more than 5 consecutive bits with the same value).
6. Repeat from *3.* if the message is not valid.
7. Try with another UART packet length if no message was found.

4.4.3 I2C

Peripheral Primitives

Similar to the other peripherals of the Hardware Abstraction Interface, the I2C peripheral interface exposes methods to initialize and de-initialize the peripheral, set the baudrate and start the write operation.

Reading CAN packets through the I2C peripheral is not possible due to the restrictions that the protocol imposes.

Writing

The `I2CSender` technique implements the possibility of writing a given bitstream to the CAN bus through an I2C peripheral.

However, bitstreams sent with this technique cannot be arbitrary, since the I2C peripheral follows the set of rules mentioned in the previous paragraphs.

In particular, the shape of each packet coming out from the I2C peripheral is illustrated in Figure 3.7.

The `setMsgToSend()` function of the technique is in charge of converting the given bitstream into packets that can be sent on the I2C peripheral, as well as checking if the format is compatible with the above-mentioned shape.

In order to carry out this task, the technique must be initialized with the following parameters:

1. `startCondDuration`: the bit length of the start condition, during which the signal is held low by the I2C device
2. `stopCondDuration`: the bit length of the stop condition, during which the signal is held low by the I2C device
3. `interFrameDuration`: the bit length of the inter-frame time, during which the signal on the *SDA* line is high

Note that, since the duration of these segments in the I2C packets are constant and depend on the specific device, the bit duration of the CAN bus that is being targeted must divide the duration of these segments, so that they can be expressed in terms of a number of bits.

Reading

Due to how the reading operation is defined by the I2C protocol, it has been deemed not suitable for intercepting CAN packets. The main reason for this

choice is that the read operation must be initiated by the master, which sends the address of the slave as a packet on the *SDA* line and signals each start and stop condition to the slave.

Therefore, even if there is a conflict between the *CANRX* line and the *SDA* on the chosen target, either the I2C device is set up as the bus master, which would cause bits to be sent on the *CAN* line, or it is set up as slave, in which case it needs an external clock signal to start reading. In both cases, correctly receiving bits from the CAN bus using an I2C peripheral is not possible.

4.4.4 ADC

Peripheral Primitives

Analog-to-Digital Converters (ADCs) generally expose a simple interface for sampling analog signals at precise intervals.

The ADC primitives provided by the CANPass peripheral interface are:

- `setFrequency()` to set the sampling frequency
- `setResolution()` to choose the resolution of the conversion; for our purposes, we generally want the lowest resolution possible, since we are not interested in small conversion errors
- `getFSR()` to calculate the full-scale range of the converted value, i.e. the maximum value that will be outputted by the ADC; this can be used to calculate the middle value of the range
- `setCallback()` to execute a function on each conversion
- `setRxBuf()` to indicate where the converted values should be stored
- a `start()` and `stop()` function to initiate and terminate the conversion

The main usage of this interface, in the context of CANPass techniques, is to either check at each converted bit if the value is greater or lower than the mid-range value, which can be done by specifying this behavior through a callback function, or simply save the converted values in a chosen buffer as raw analog reads and convert them later into bits.

Reading

Clearly, the only action that can be carried out through an ADC peripheral is reading. Therefore, if the *CANRX* signal conflicts with an analog input of the

chosen target, the ADC can be used for sniffing bits on the bus.

This task is carried out by the `ADCReceiver` technique, which can either listen on the bus until a given sequence of bits is received or record the bus activity for a fixed number of conversions and store the results in a buffer.

Chapter 5

Experimental Validation

To validate our work, we have chosen to implement the CANPass primitives for two different microcontrollers, a high-end, automotive-graded one and a low-end, cheap one, to prove the effectiveness of our techniques across different targets. Section 5.1 describes in detail the hardware platforms chosen for our experiments.

Our objective is to measure the reliability that we can achieve with each technique, i.e. how precisely can we craft and read arbitrary bits on the CAN bus, without being attached to a specific attack implementation. For this reason, we chose as the main metric of our evaluation the maximum speed at which a given technique is able to read or write *entire messages* on the CAN bus.

The fact that we use entire messages instead of single bits for our experiments poses more constraints on the reliability of the methods employed, since small timing errors and clock drifts in a single bit are summed with those of the preceding ones. We are confident that, if we can show that a given technique is able to handle whole CAN messages without errors, we can convince the reader that the reliability of the method also applies to shorter sequences of bits. More details about our experimental approach are provided in Section 5.2.

Finally, Section 5.3 shows how the experiments have been set up, Section 5.4 and Section 5.5 list all the experiments performed, and Section 5.6 discusses the results achieved for each proposed technique.

5.1 Chosen Platforms

All of the CAN controller bypass techniques analyzed in this work have been implemented for two completely different platforms, from different vendors and with different toolchains. This section describes the hardware and software components of these platforms.

5.1.1 Hardware

The microcontrollers chosen for our experiments are the NXP **LPC11C24** [30], a low-end microcontroller equipped with an ARM Cortex M0 processor, and the Infineon **AURIX TC399XP** [13], a high-end, automotive-grade microcontroller with a 6-core processor from the TriCore family. A comparison between the two microcontrollers is provided in Table 5.1.

	LPC11C24	**TC399XP**
Vendor	NXP	Infineon
Cores	1	6
SRAM Size	8 kB	2.9 MB
Flash Size	32 kB	16 MB
CAN Peripherals	3*	1**
SPI Peripherals	2	6
I2C Peripherals	1	2
UART Peripherals	1	12
ADC Channels	8	12
CPU Speed	50 MHz	300 MHz
Price Range	20$	150$

* The LPC11C24 comes with an integrated CAN transceiver, therefore the CAN pins cannot be used by other peripherals. ** The TC399XP internally supports assigning 4 CAN nodes to each CAN peripheral, for a total of to up to 12 CAN nodes.

Table 5.1: Comparison between the LPC11C24 and the TC399 microcontroller.

As the comparison shows, the AURIX microcontroller is more complex, advanced and fast, and includes several peripherals of the same type for each bus analyzed, while also having many more dedicated peripherals (e.g. FlexRay, LIN, Ethernet, Sigma-Delta ADCs) that are not shown here. Each peripheral itself in the AURIX platform is more complex than the corresponding LPC one, providing much more configuration and flexibility to the user. This microcontroller comes also with high-end security features, such as a Hardware Security Module, and is designed specifically for automotive applications. It is equipped with 3 CAN interfaces, each of which can be internally routed to up to 4 different nodes, making a total of 12 possible logical CAN nodes. Moreover, having a multi-core processor, it is able to carry out many real-time tasks in parallel, which is important in advanced applications such as those performed by ECUs.

The LPC11, on the other hand, has a minimal set of features and includes a few simple peripherals, that can communicate with some of the basic protocols. This is a low-end ARM Cortex-M0 microcontroller mainly suited for simple, low-power applications, and the price point at which it is sold reflects the absence of advanced features. A peculiarity of this board is that it contains not only an embedded CAN controller, but also an *embedded CAN transceiver*, which needs an external 5V power supply to work. This means that the signals coming out from the MCU are `CANH` and `CANL`, instead of `CANTX` and `CANRX`. This means that, internally, these pins are completely separated from the rest of the MCU, being analog high-voltage signals, which means that, unfortunately, they cannot be used to produce pin conflicts. For this reason, in our setup, we had to employ an external CAN transceiver and connect the peripherals under test to it as if they were conflicting with `CANTX` and `CANRX` pins.

To test the LPC11C24 MCU, we used the LPCXpresso board provided by Embedded Artists [31], which includes the LPC11C24 microcontroller as well as a CMSIS-DAP Debug Interface, accessible from USB, to program the microcontroller. It also comes with 2.54 mm square headers, which can be used to easily access each of the MCU's pins.

To test the TC399XP MCU, we used the TriBoard Starter Kit provided by Infineon [14], which too provides a complete platform suitable for programming and testing this target. It is equipped with a debugger interface, various LEDs and buttons, an external clock source, two external CAN transceivers already mounted on the board, a small connector for each pin and an expansion board, that can be seen at the bottom of Figure 5.6, through which the MCU pins can be accessed in groups of 80 with standard 2.54mm square headers.

The choice of these platforms has a twofold aim. First of all, we want to demonstrate that the techniques presented in this work can be effectively used on a real, automotive-grade microcontroller, such as the AURIX TC399, that can be found on real ECUs in the wild.

Secondly, we aim at covering also the opposite side of the microcontroller spectrum, composed of small and inexpensive microcontrollers such as the LPC11C24, which can be found on simpler systems.

We consider these two platforms as representatives of the high-end and low-end categories of microcontrollers currently found on the market. Since they differ in many aspects, including the vendor, CPU architecture, clock speed, peripheral chips and overall performance, they are a perfect fit for demonstrating the flexibility of our implementation.

5.1.2 Software

Since both boards already comprise a debugging and programming interface, which is accessible from a PC or laptop via USB, no external hardware was needed to program the boards.

In order to program the selected platforms, we used the standard libraries and build toolchains provided by each vendor:

- For the AURIX board, we used the *AURIX Development Studio IDE* [12]. This tool is a free-of-charge, Eclipse-based Integrated Development Environment (IDE) for the TriCore-based AURIX microcontroller family. It is a comprehensive development environment, that includes a C-Compiler from TASKING [35] and a multi-core debugger. It also comes with the Infineon low-level driver (iLLD) library, which contains all the code needed to interact with the hardware peripherals. In our implementation of the platform layer for the TC399XP microcontroller, we make heavy use of these drivers to implement the peripheral primitives needed by CANPass.
- For the NXP board, we used the MCUXpresso IDE [32], which is another free Eclipse-based IDE that includes a compiler and debugger for the LPC11C24 microcontroller. It comes with its own set of low-level drivers, which are included in the `lpc_chip_11cx_lib/` and `nxp_lpcxpresso_11c24_board_lib/` folders.

Both of these tools were installed on a Windows 10 laptop and used to program the two boards. More details on the hardware and software setup of our tests are provided in Section 5.3.

5.2 Experimental Approach

This section provides an overview of the approach used to test the techniques proposed in this work, as well as the CANPass implementation, on the chosen platforms.

5.2.1 Objective of the Experiments

In our experiments, we want to prove that bit injection and sniffing can be done using new techniques, namely the ones described in Section 3.6. In particular, we want to prove that such techniques are highly *reliable*, in contrast with bitbanging

techniques. Moreover, we do not want to be bound to a specific attack implementation. As a measure of each technique's reliability, we want the timing of reading and writing operations to be predictable and consistent through time.

Hence, our approach for these experiments is to read and write *complete messages* on the CAN bus with the aforementioned techniques. In this way, we can prove that even long sequences of bits are injected and received in a way that is indistinguishable from a real CAN peripheral, proving that the timing of such operations is correct.

On the other hand, we are not implementing a real bit injection attack, since we want to measure the general properties of our techniques, and we did not simulate a complete, realistic environment, as this would introduce unhelpful complexity to our setup. With these tests, we are interested in proving that our techniques are possible, validating our implementation and measuring the reliability of our techniques at a finer grain than a single bit injection, therefore a realistic attack scenario, which will be tested in future work, is out of the scope of these experiments.

For this reason, for each of the techniques proposed in this thesis, we try to answer the following question: *can this method be used to read or write a complete CAN message with the same reliability as a legitimate CAN controller?*

For writing techniques, this question can be formulated as: *can this technique be used to write a complete CAN message on the CAN bus such that it will be accepted as valid by a legitimate CAN device listening on the same bus?*

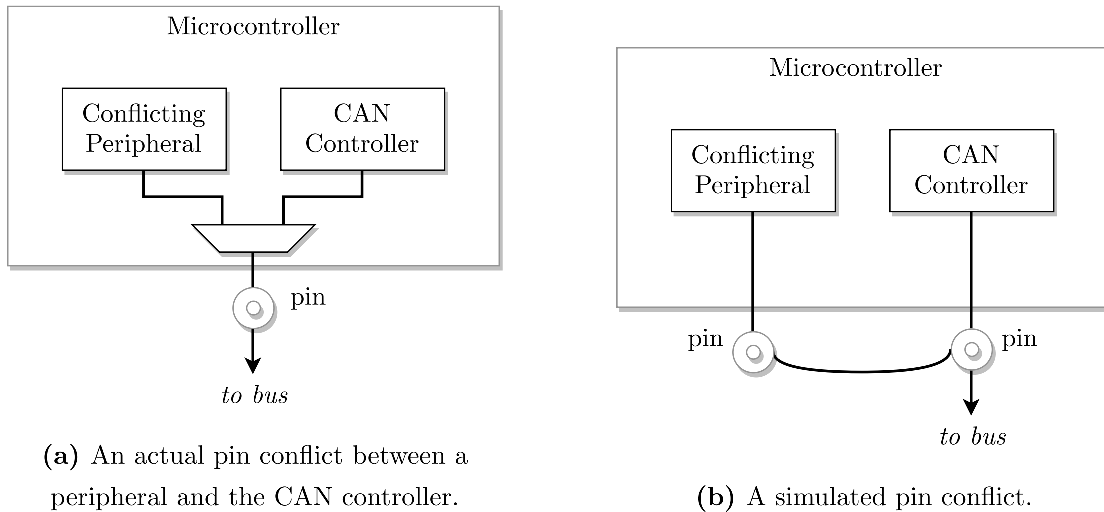
For reading techniques, we ask: *can this technique read bit-by-bit a complete CAN message sent by a legitimate CAN device, without skipping or duplicating bits?*

5.2.2 Simulating Pin Conflicts

Since not every technique described can be replicated on the chosen targets due to the lack of conflict between the CAN peripheral and all the other peripherals, we instead decided to simulate such conflicts by wiring together the signals coming out from the peripheral under test to the *CANRX* and *CANTX* signals, that are connected to the CAN peripheral.

In other words, we are simulating with external wiring the behavior that is normally displayed by internal signal multiplexing. Figure 5.1 illustrates the difference between real pin conflict and simulated pin conflict.

Even if this is not the hardware setup that we have considered when devising the conflicting peripheral techniques, we are confident that our setup closely mim-



(a) An actual pin conflict between a peripheral and the CAN controller.

(b) A simulated pin conflict.

Figure 5.1: Difference between real pin conflict (a) and simulated conflict (b)

ics a situation in which two peripherals have a pin conflict in the chip, without the need of using more than two hardware platforms for testing.

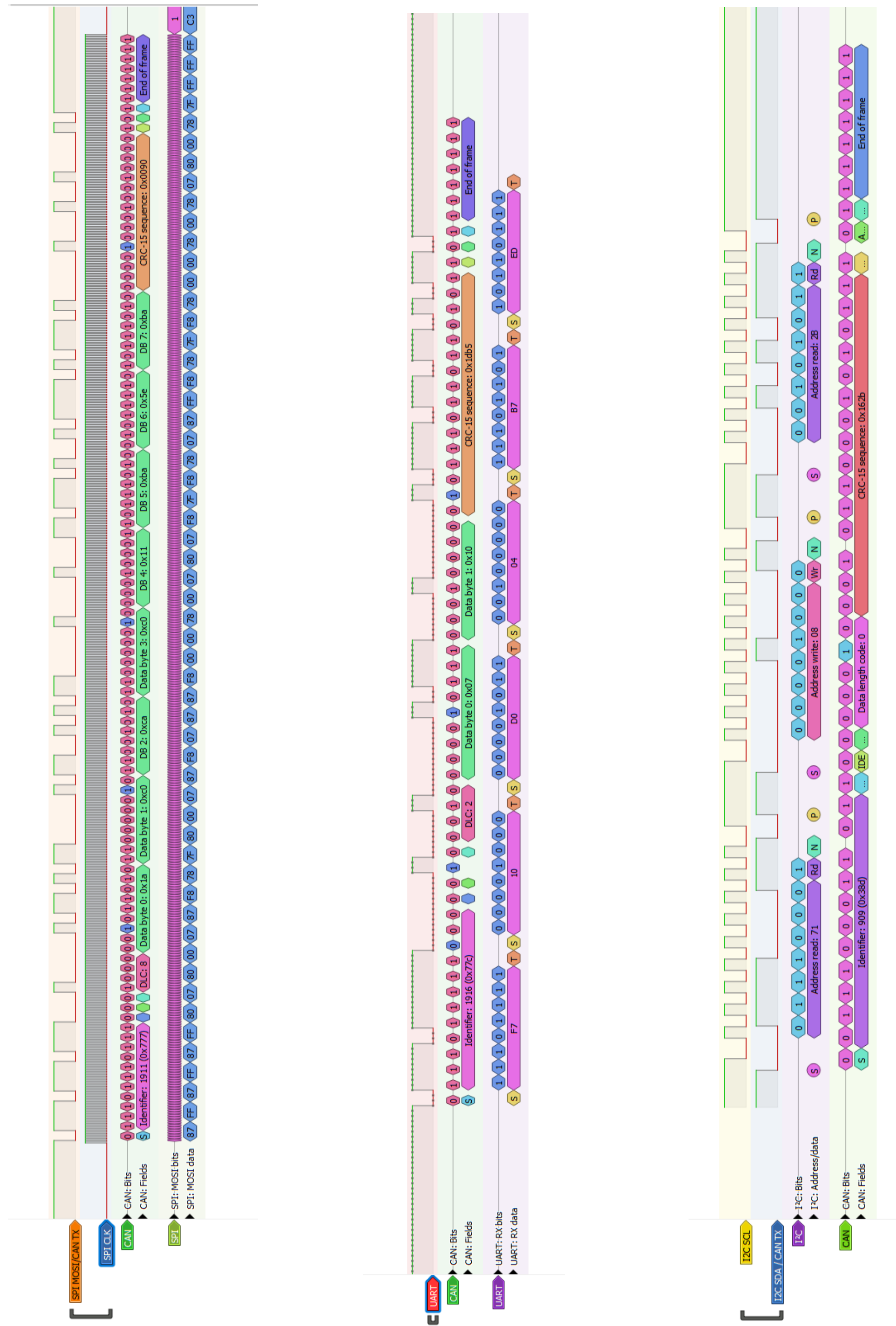
5.2.3 Exchanged Messages

To perform our experiments, we decided to use three different CAN messages: a generic CAN message, which is common to all techniques that do not have any particular limitation, and two specifically crafted messages to meet the requirements of the UART techniques and the I2C technique.

The bit representation for each message can be found in Figure 5.2, which was generated by capturing the signals on the bus with a logic analyzer.

Generic Message

The message sent and received through the SPI interface is a generic CAN message with the longest possible payload (8 bytes) and a generic content, shown in Figure 5.3. This message has also been used to validate the ADC read technique, as well as for bitbanging techniques. This message has no particular feature, it includes stuff bits and represents a randomly picked, standard CAN frame. The bit representation of this message, as well as a graphical plotting of its logical levels, can be found in Figure 5.2a.



(a) A SPI packet that is also a valid CAN message.

(b) A UART message that is also a valid CAN message.

(c) A I2C message that is also a valid CAN message.

Figure 5.2: Three examples of how SPI (a), UART (b) and I2C (c) peripherals can be employed to craft valid CAN messages.

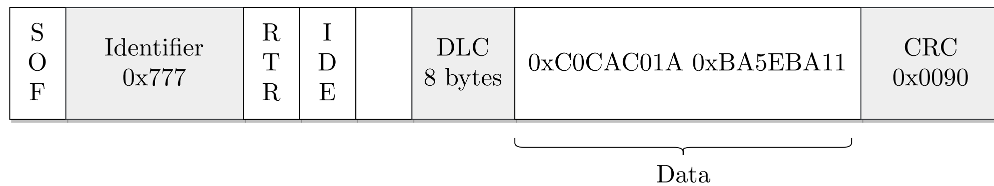


Figure 5.3: CAN message used for testing SPI and ADC.

UART Message

The CAN message chosen for the UART peripheral has been instead carefully crafted to respect both the restrictions of CAN packets and those of the UART protocol. A precise description of the shape that CAN messages received and sent through the UART peripheral must have can be found in Section 3.6.2. In particular, the message shown in Figure 5.4 has a 2-byte payload, and the ID and content of the payload have been carefully chosen in such a way that all the fields, including the CRC field, do not break the UART rules.

Note that is not the only message that can be sent with this peripheral, but since the UART protocol poses restrictions on every packet sent, a greater length of the message also means more constraints. A visual representation of the message bitstream and its constraints is provided in Figure 5.2b. Each packet of the UART protocol, represented as a pink segment in the lowest row, must start with a '0' (yellow hexagon) and terminate with a '1' (orange hexagon).

This means that the bitstream sent and received through the UART peripheral cannot have arbitrary bits in positions corresponding to start and stop bits. The procedure to generate and check valid CAN packets that can be sent through the UART technique can be found in Section 4.4.2.

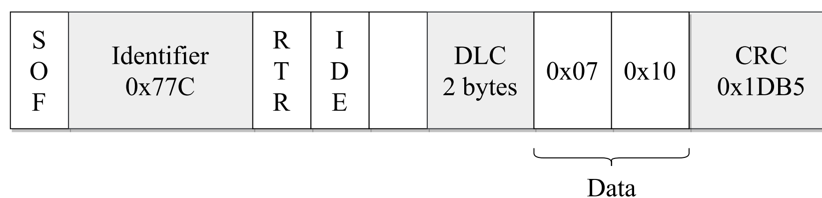


Figure 5.4: CAN message used for testing UART.

I2C Message

The CAN message sent by the I2C peripheral is described in Figure 5.5: it is a simple remote frame request with a null payload and an ID of `0x38d`.

These values too have been carefully chosen to be compliant with the more restrictive rules imposed by the I2C protocol on the shape of each message, which are described in Section 3.6.3 and Section 4.4.2.

In particular, the I2C writing technique suffers from the severe limitation of having fixed durations for the start and stop conditions of each packet, which can be seen in the second row of Figure 5.2c. In this figure, each packet's payload is colored in violet, while the start condition duration is represented by the space between the pink circle (`S`) and the start of the payload. Other fixed bits are the ACK bit (green segment), which will always be '1' since no I2C slave is connected to send an acknowledgement, and the stop condition, which is the space between the end of the ACK bit and the yellow circle (`P`). The inter-frame time, which is the space between the stop condition of a packet and the start condition of the following one, is also fixed and must be '1'.

The fact that the start and stop bits have a fixed duration means that the content of the message must be chosen together with the bitrate of the bus. In fact, the same duration of 10 μs can be seen as two bits, if the baudrate is set to 200 kbit/s, or 1 bit, if the baudrate is 100 kbit/s. For this reason, we chose a baudrate that was compatible with the length of each start and stop condition of the LPC microcontroller (5 μs, which corresponds to 200 kbit/s) and crafted each I2C packet so that, when connected together, they formed the CAN message illustrated in Figure 5.5. This technique was not implemented for the TC399XP microcontroller.

Although this is not the only valid CAN message that can be sent with the I2C technique, it is important to keep in mind that, on longer messages, the constraints of the UART protocol together with those of the CAN protocol pose some challenges to finding a valid message.

As a side note, we can see that the message space for the SPI technique is strictly bigger than that for the UART technique, which is still bigger than the possible message space for I2C techniques. This should be taken into account when choosing which technique to use on a given target.

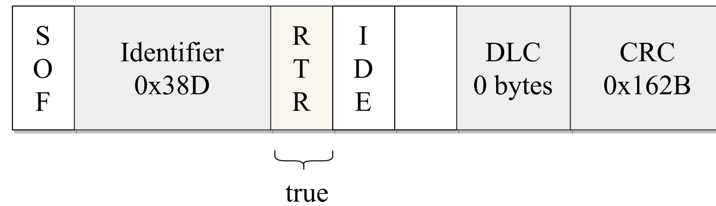


Figure 5.5: CAN message used for testing I2C.

5.3 Test Setup

The setup and wiring of the test bench used in our experiments is described in this section.

5.3.1 Test Bench Components

To carry out our tests, we have created a simple test bench, composed of:

- A Windows 10 laptop, with the following software installed:
  - The AURIX Development Studio IDE, needed to program the TC399XP microcontroller
  - The MCUXpresso IDE, needed to program the LCP11C24 microcontroller
- The LPCXpresso board, connected via USB with the laptop
- The TriBoard board, connected via USB with the laptop
- The MCP2551 CAN transceiver [24], which is used to interface the LPC11C24 board with the CAN bus, since the only CAN controller on this board is internally connected to an on-chip CAN transceiver and cannot be accessed by other peripherals

To monitor the communication on the CAN bus we also:

- Attached to the laptop, via USB, a *Logic Analyzer*, which is a small device that is able to capture logical signals sent on the CAN bus
- Installed the Pulseview [34] software, a free, open-source, Qt-based GUI to visualize and decode logical data recorded by the logic analyzer

Figure 5.6 contains a picture of the final setup, while Figure 5.7 illustrates the connections between components.

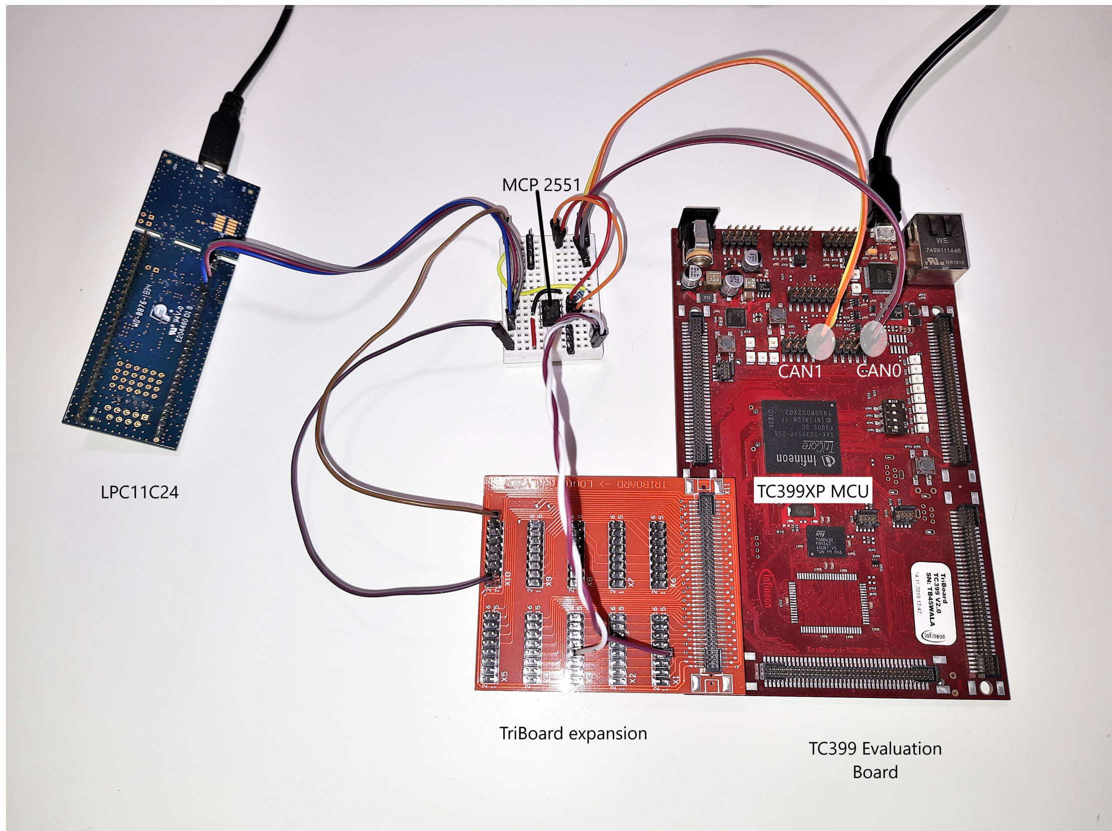


Figure 5.6: Picture of the test setup.

5.3.2 Nodes of the Test Network

From a high-level perspective, the depicted setup connects on the same CAN bus three logical CAN nodes:

- The **LPC node**, composed of the LPC11C24 MCU and the MCP2551 CAN transceiver. In this node, the CAN transceiver can be connected to different peripherals of the LPC microcontroller, simulating a peripheral conflict.
- The **CAN0 interface** of the AURIX MCU, which uses the CAN0TX and CAN0RX signals. As these signals are already routed to the on-board CAN transceiver on the board in use, connecting other peripherals to the CAN signals is possible by connecting the peripheral pins to the CAN0TX and CAN0RX pins, as shown in Figure 5.7.

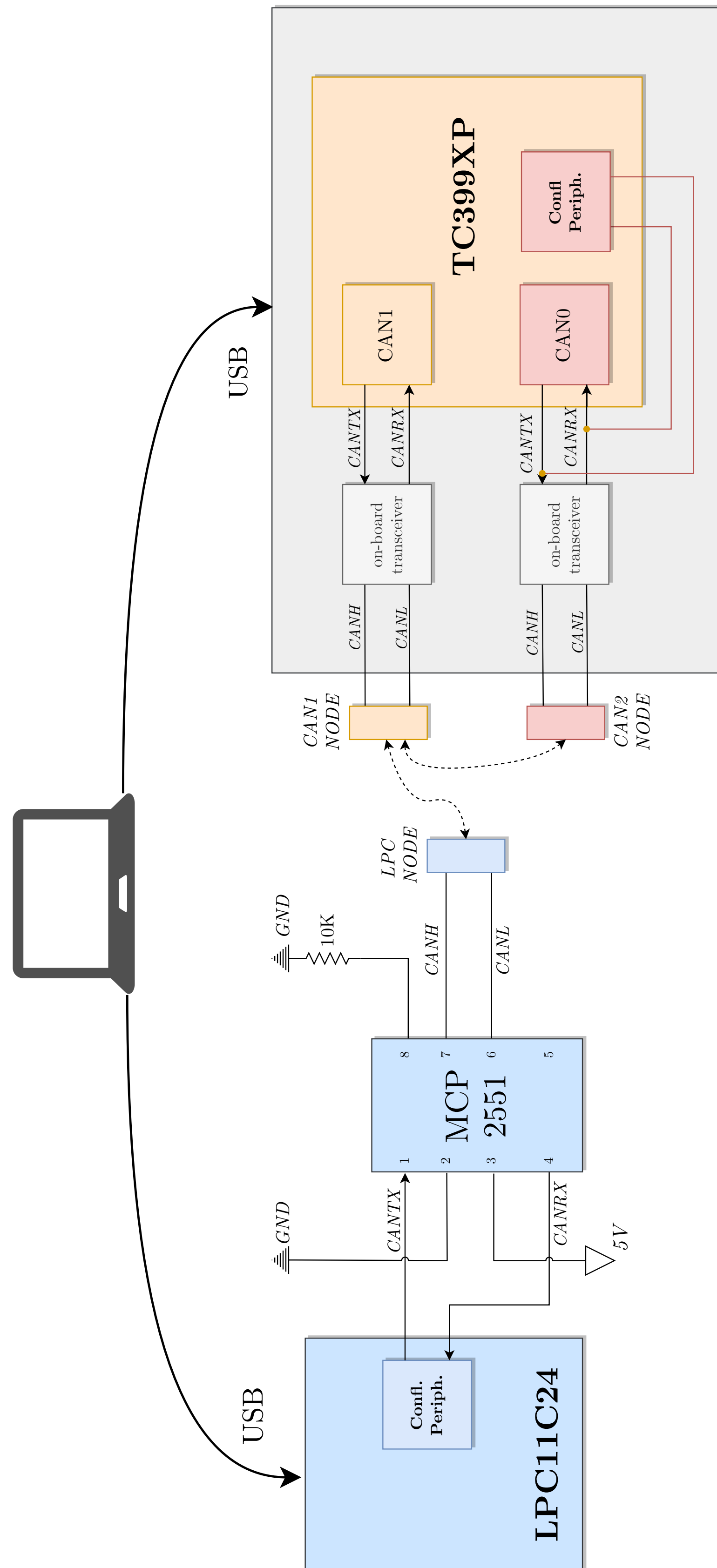


Figure 5.7: Wiring of the test setup.

- The **CAN1 interface** of the AURIX MCU, which uses the CAN1TX and CAN1RX signals and is connected to a separate CAN transceiver. The CAN1 signals are internally routed to the CAN1 hardware peripheral of the microcontroller, and act as separate CAN node.

These nodes can be used to simulate the communication between a *legitimate* CAN node, representing an ECU that has to be compromised or shut down, and an already *compromised* CAN node, which has been reprogrammed by an attacker to access the CAN data link layer using CANPass.

In particular, the LPC node and the CAN0 node of this network act as compromised ECUs. On these nodes, the peripheral conflict is simulated as described in Section 5.2.2. The CAN1 node, on the other hand, acts as a legitimate CAN node.

It has to be noted that, even if the CAN0 and CAN1 peripherals of the AURIX microcontroller are both located on the same chip, they are effectively two independent pieces of hardware inside the microcontroller and, since the TC399XP has as many as 6 cores running in parallel, if one is entirely dedicated to the CAN1 interface, as in our case, the two interfaces can be effectively considered two separated nodes on the bus.

5.3.3 Types of Experiments

The test configuration described in Section 5.3.2 can be used to perform four types of experiments:

1. **LPC writing** experiments: a peripheral of the LPCXpresso board is connected to the bus, simulating the conflict with CAN pins, and sends bits to the CAN transceiver. On the other end of the bus, the CAN1 interface of the TC399 MCU receives CAN messages through the usual CAN controller.

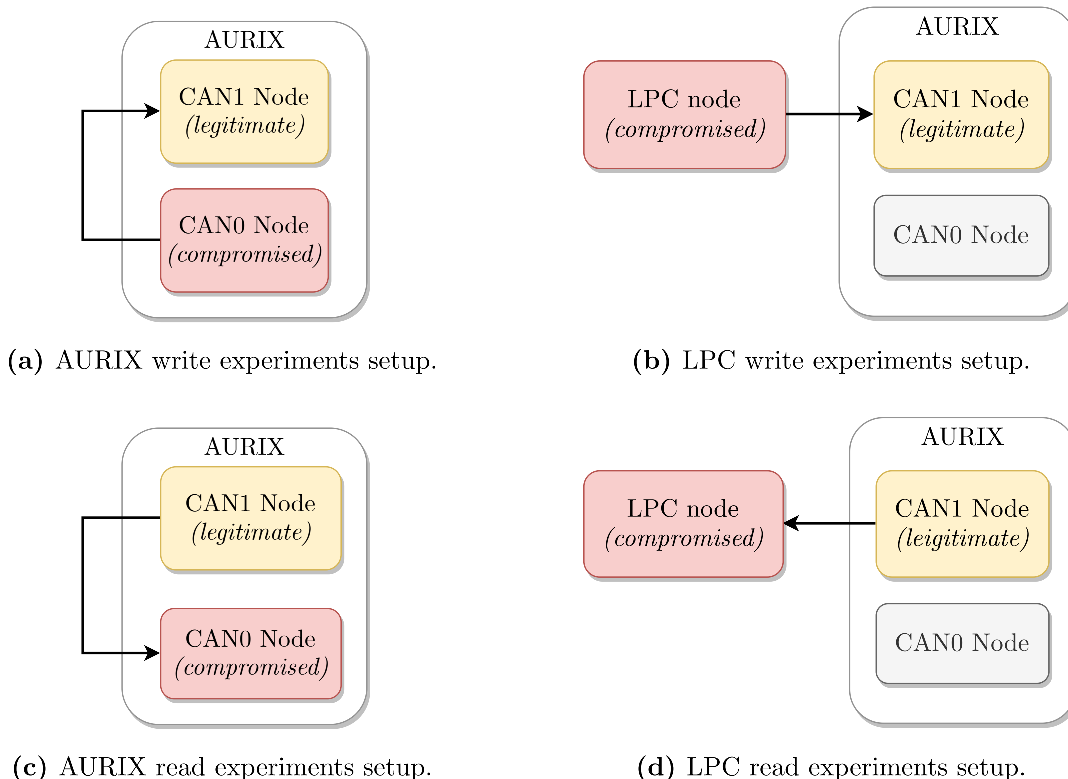
2. **LPC reading** experiments: the CAN1 interface of the TC399 MCU sends legitimate messages on the bus, while the LPC board, connected through a simulated conflicting peripheral, reads the incoming bits.

3. **AURIX writing** experiments: the CAN0 interface of the AURIX microcontroller is connected to another peripheral on the same board, simulating the pin conflict. The peripheral is used to write on the bus, while the CAN1 interface reads the CAN bus as a legitimate node.

4. **AURIX reading** experiments: the CAN0 interface is used to read through a simulated conflicting peripheral, while the CAN1 interface sends normal messages on the CAN bus.

Figure 5.8 depicts the four different test scenarios just mentioned. In a nutshell, the CAN1 node always acts as the legitimate node, while the LPC node and the CAN0 node are used to test the techniques of this work.

When the CAN0 node is being tested, the LPC node is disabled, and vice-versa.



(a) AURIX write experiments setup.

(b) LPC write experiments setup.

(c) AURIX read experiments setup.

(d) LPC read experiments setup.

Figure 5.8: Four different test scenarios.

5.4 LPC Experiments

To perform experiments on the NXP microcontroller, the LPC11C24 board was connected to an MCP2551 CAN transceiver as follows:

- Pin 2 of the MCP2551 to the board's `GND`
- Pin 3 of the MCP2551 to an external `5V` source
- Pin 8 of the MCP2551 to ground with a 10KΩ resistor

- Pin 6 and 7 of the MCP2551 to `CANL` and `CANH` lines of the network, respectively

Pin 1 (`CANTX`) and Pin 4 (`CANRX`) have been connected the pins of each peripheral under test, simulating a peripheral conflict.

The `CANL` and `CANH` lines were then wired to the corresponding signals of the AURIX CAN1 node, which acted as a legitimate node on the network.

5.4.1 LPC Writing Experiments

For the LPC microcontroller, we conducted experiments on the following writing techniques:

1. *Bitbanging*, both using a timer and busy-wait loops
2. *SPI writing*, using the SPI1 peripheral
3. *UART writing*, using the only UART peripheral available
4. *I2C writing*, using the only I2C peripheral available

For each of these techniques, we sent 100 messages on the bus, and verified the number of packets correctly received by the legitimate node.

Bitbanging

Bitbanging has been implemented with two different approaches: one that uses CPU only and one that uses a hardware timer. The pin used for both techniques was `PIO2_0`.

For the CPU-only technique, the timing between each bit has been regulated with *busy-wait loops*, which are a way to make the CPU spin for a certain number of cycles. After some experimenting, we have found that the minimum number of cycles needed to correctly craft CAN bits on this platform is equal to 3, which corresponds to a CAN bitrate of 200 kbit/s. Lower quantities were found to generate an incompatible timing with respect to typical CAN bitrates.

For the timer technique, we set up a hardware timer, in particular `TIMER32_0`, to execute a callback function at fixed time intervals. This function was in charge of writing the next bit of the message to send and increase the counter of sent bits, until the end of the message was reached. The reload value of the timer has been set to the lowest possible value of 1, which generated a measured interval between bits of 8.3 μs, corresponding to a baudrate of 120 kbit/s.

SPI

For communicating on the SPI from the LPC microcontroller, we employed the Synchronous Serial Port (SSP) peripheral, which can perform communications on Motorola SPI, 4-wire TI SSI, and National Semiconductor Microwire buses, by connecting the board's `PIO0_9` pin (`SPI1_MOSI`) to the MCP2551 `CANTX` pin.

For our implementation of the SPI abstract peripheral in CANPass, we chose the `TI` frame format, as it allows for subsequent bytes to be sent immediately one after another with this peripheral. We then set the peripheral's baudrate to 1 Mbit/s and performed repeated writes using the SPI peripheral, which correctly produced valid CAN messages at the same baudrate.

UART

UART capabilities on the LPC are provided by the `LPC_USART` peripheral. To simulate a peripheral conflict, we connected the board's `PIO1_7` pin (`UART_TX`) to the MCP2551 `CANTX` pin.

The peripheral has been configured with the following settings:

1. data length of the payload: 8 bits
2. number of stop bits: 1
3. number of parity bits: 0
4. baudrate: 1 Mbit/s

As a consequence of this, we were able to craft a correct CAN message through the `setMsgToSend()` function and send it successfully on the bus for 100 times with a speed of 1 Mbit/s.

I2C

For testing the I2C writing technique on the LPC, we connected the transceiver's `CANTX` line to `PIO0_5` (`I2C_SDA`). As already discussed in Section 5.2.3, with this technique the message and the baudrate have to be chosen together, as they are interdependent. The target baudrate of the CAN bus was set to 200 kbit/s, as illustrated in Section 5.2.3.

Due to slight differences between the clocking of this peripheral and the length of fixed portions, as the start and stop conditions, we had to compensate for small bit timing errors in order to make the CAN receiver accept the transmitted

message. In particular, bits sent on the bus were slightly slower than the target bitrate. To solve this problem, we set the peripheral's baudrate to 210 kbit/s in the LPC's firmware, which generated the desired baudrate of 200 kbit/s on the CAN bus.

5.4.2 LPC Reading Experiments

With a similar method we tested the following reading techniques:

1. *Bitbanging*, both using busy-wait cycles between reads and using a timer
2. *SPI read*, using the SPI1 peripheral
3. *UART read*, using the only UART peripheral available

The *ADC* technique was tested only on the TC399XP microcontroller.

For each of these techniques, we programmed the AURIX microcontroller to send 100 CAN messages on the bus through the CAN1 interface, and verified the bitstream on the receiving end by comparing the bits read by the chosen technique with the expected bitstream.

In order to detect the start of messages, for each of the tested techniques we performed *polling* on the pin connected to the transceiver's `CANRX`, i.e. we continuously read the logical value of the pin until a falling edge, indicating the *Start of Frame* bit, was detected After the falling edge was detected, we fired the technique's `read()` function and checked the received bitstream.

Bitbanging

Similarly to what has been described for writing experiments, we used as a baseline for our comparisons two possible implementations of the bitbanging technique: one employing busy-wait and the other employing the `TIMER32_0` hardware timer. Both techniques used the `PIO2_0` pin, which was connected to the `CANRX` signal of the CAN transceiver.

The configuration of the hardware timer was the same as writing techniques, and led to the same results. The busy-wait implementation was instead affected by small deviations that ultimately led to wrong readings. More details on the reasons for this phenomenon are provided in Section 5.6. As a consequence, we doubled the number of cycles between one bit and the following one and decreased the CAN bus speed to 100 kbit/s, which lead to a correct transmission on the CAN.

SPI

For testing the SPI reading technique on the LPC we connected the board's PIO0_8 pin (SPI1_MISO) to the MCP2551 CANRX pin and used the same SSP peripheral employed for writing techniques, using the same settings.

After detecting a falling edge on the bus, we read with the SPI peripheral a number of bytes corresponding to the expected message's length and confronted the received bits with the expected bitstream. Note that, since the SPI peripheral reads 8 bits at a time, the length of the received message had to be a multiple of 8. Another important aspect to be kept into consideration is that the SPI technique's implementation stores the bits read with this peripheral as big-endian bytes, so some postprocessing was needed in order to compare the received bits in the right order.

UART

In a similar fashion, we used the UART peripheral to read CAN messages on the bus after a falling edge was detected. The number of bytes to read were known in advance, as the expected message was fixed. In more realistic scenarios, the bus could be read one byte at a time to verify the first bits of the incoming message.

To perform this experiment, we connected the PIO1_6 pin (UART_RX) to the CANRX signal coming out of the CAN transceiver and set the bus's baudrate to 1 Mbit/s.

As with SPI, this technique stores received bits with a peripheral-specific encoding, i.e. it stores only the payload of each received packet, but since the other bits (start condition, stop condition, nack) are fixed, the full bitstream of the message can be recovered with some postprocessing.

5.5 AURIX Experiments

For conducting our experiments on the AURIX microcontroller, we wired together the CANH and CANL lines of the two CAN interfaces (CAN0 and CAN1) available on the board. To simulate the pin conflict, we attached the signals coming out from each writing peripheral under test to CAN0TX (pin P20.8) and each signal going to reading peripherals under test to CAN0RX (pin P20.7). The CAN1 peripheral was instead used as a legitimate node connected to the bus, as already done for the LPC experiments.

5.5.1 AURIX Writing Experiments

For the AURIX microcontroller, we conducted experiments on the following writing techniques:

1. *Bitbanging*, both using a timer and busy-wait loops
2. *SPI writing*, using the QSPI0 peripheral
3. *UART writing*, using the ASCLIN2 peripheral

The *I2C writing* technique was tested only on the LPC microcontroller.

For each of these techniques, we sent 100 messages on the bus, and verified the number of packets correctly received by the legitimate node.

Bitbanging

As already described in Section 5.4, bitbanging techniques have been tested both with and without the use of a hardware timer.

The pin used for both bitbanging techniques was exactly the `CAN0TX` pin (`P20.8`), while the timer used was the `T3` timer of the `GPT12` module set to *timer* mode, which provides a way to perform high-speed operations at fixed intervals.

After some empirical observations, the reload value of the timer was set to 18, while the number of cycles that was empirically found to be fit for producing consistent bit timings was 90.

With these settings, we were able to successfully produce CAN messages at a speed of 1 Mbit/s which could consistently be read by the receiving device.

SPI

The AURIX platform provides a Queued SPI (QSPI) device, whose aim is to provide fast and flexible communication, either point-to-point or master-to-many slaves. This peripheral is quite complex and can be configured to use different FIFO queues, different modes of operation and up to 32 bits per packet.

For our purposes, we used the default setup for the *SPI master* mode and configured the baudrate to 1 Mbit/s, which was enough to correctly generate CAN messages at 1 Mbit/s after connecting the `P20.8` (`CAN0TX`) to `P22.5` (`SPI MOSI`).

UART

For performing UART communication, we employed the *ASCLIN* peripheral, which supports many serial communication protocols within one peripheral: LIN, 3- and 4-wire communication, and many more. We used the internal ASC module (Asynchronous/Synchronous Communication) interface to replicate UART messages with this peripheral. We then set the data length to be 8 bits and the number of parity bits to 1.

The pin conflict was simulated by connecting the `CAN0TX` pin to the `P10.4` (`ASCLIN TX`) pin, and the baudrate was set to 1 Mbit/s.

Note that, in the AURIX platform, UART frames can also be sent using another peripheral, called `PSI5`, which has not been tested in this work but could be employed for the same purpose.

5.5.2 AURIX Reading Experiments

With a similar method we tested the following reading techniques:

1. *Bitbanging*, both using busy-wait cycles between reads and using a timer
2. *UART read*, using the ASCLIN2 peripheral
3. *SPI read*, using the QSPI0 peripheral
4. *ADC read*, employing the special *ADC FC* (Fast-Channel) peripheral available on this platform

For each of these techniques, we programmed the AURIX microcontroller to send 100 CAN messages on the bus through the CAN1 interface, and verified the bitstream received by the technique under test. Since the messages are fixed, it is possible to verify directly on the host if the bitstream read with a given technique corresponds to the expected one.

Also in this case, we waited for a falling edge on the `CANRX` pin before starting the receive operation, as with the reading experiments for the LPC microcontroller.

The receiving peripherals have been set up with the same settings used for sending.

Additionally, the ADC peripheral was used to read bits on the bus. In particular, we used the AURIX `EVDAC` peripheral (Enhanced Versatile ADC), connecting the pin corresponding to Fast Compare Channel 3 (`P00.11`) to the `CAN0RX` signal.

This channel provides a way to perform fast comparisons between the input voltage and a reference voltage, outputting the result of the comparison as a single bit: '0' if the signal is lower to the reference, '1' in the opposite case.

For our experiments, we set the compare value to half of the full-scale range, so at each conversion the fast channel was able to output a '0' if the sampled voltage was low and a '1' if the voltage on the bus was high. As a result, we were able to sample incoming bits at a sampling rate of 1 MHz, with a resolution of 1 bit.

5.6 Results

Table 5.2 reports the maximum bus speed achieved with each experiment carried out on writing techniques, while Table 5.3 concerns the reading techniques. The code used for the experiments is available in the CANPass repository, inside the `tests/` folder.

MCU	GPIO	GPIO+timer	I2C	UART	SPI
LPC11	200 kbit/s	120 kbit/s	200 kbit/s	1 Mbit/s	1 Mbit/s
TC399	1 Mbit/s	1 Mbit/s	-	1 Mbit/s	1 Mbit/s

Table 5.2: Maximum speed achieved for each writing technique tested.

MCU	GPIO	GPIO+timer	UART	SPI	ADC
LPC11	100 kbit/s	120 kbit/s	1 Mbit/s	1 Mbit/s	-
TC399	500 kbit/s	1 Mbit/s	1 Mbit/s	1 Mbit/s	1 Mbit/s

Table 5.3: Maximum speed achieved for each reading technique tested.

Since the AURIX TC399XP MCU is wildly more powerful than the LPC microcontroller, we were able to achieve the highest possible bitrate with many of the techniques analyzed. However, for the cases in which the speed achieved with our techniques was the same of the speed achieved with bitbanging, it is important to make the following remarks:

1. Bitbanging techniques occupy the whole CPU time while reading/writing, while peripheral-based techniques offload the read and write tasks to the peripheral

2. No processing can be done during one bit and the following one with bitbanging, while peripheral techniques can be implemented also in a non-blocking

way, as discussed in Section 4.2.5; this means that, while the peripheral is processing the incoming/outgoing bitstream, the CPU can execute checks and process the bitstream in real-time

3. Peripheral-based techniques are not affected by the current CPU load, while bitbanging techniques are

Moreover, after observing the data recorded from the CAN bus with the logic analyzer, we were able to see small differences between bit timings produced by bitbanging techniques. A precise measurement of this deviation was out of the scope of these experiments, but further experiments can be conducted in the future to measure the difference in bit times at a sub-bit level, which might be used by advanced voltage-based IDSs as a hint of an anomalous packet.

On the other hand, in the LPC microcontroller the difference between bitbanging and our approach is more evident, as the CPU alone cannot cope up with the speed of the bus. This is a very important result, since it shows how peripheral-based techniques enable previously impossible precision in injecting and reading bits on the bus.

5.6.1 AURIX Results

Bitbanging techniques (i.e. *GPIO* and *GPIO+timer*) executed on the TC399 were already able to cope with high bus speeds.

In particular, writing bitbanging techniques tested on the TC399 were able to repeatedly send messages on the CAN bus at a 1 Mbit/s baudrate without desynchronizing or causing errors on the legitimate receiver. This is mainly due to the high clock speed of the CPU cores of this platform. However, it has to be noted that no other task was being executed in the same core by the microcontroller, hence the CPU was completely dedicated to bitbanging. Moreover, as already noted, differences in generated bit timings were observed at a sub-bit level. Other writing techniques also achieved the same speed, showing that arbitrary messages can be sent on the CANBus also using the SPI and UART peripherals. As already noted, the main difference is that these techniques can be implemented in a *non-blocking* way, allowing for the CPU to execute other operations, e.g. monitoring the bus in real-time while the message is being sent.

On the other hand, reading bits with bitbanging and busy-wait loops was proven to be less reliable than writing, since some packets were incorrectly read at a baudrate of 1 Mbit/s. The main reason for this is that, while, during writing, the transmitting device imposes the timing of the communication to other devices,

by means of the *soft resynchronization* mechanism, during reading operations the device does not have this power. In our case, since the interval generated by software between one bit and the other was slightly more than 1 μs, this difference was accumulated during the sampling of the packet until a bit was incorrectly read. Decreasing the interval between bits caused instead the bit timing to be much lower than 1 μs, causing bits to be read twice.

However, adding a hardware timer with the resynchronization mechanism illustrated in Section 4.3 was enough to cope with these small timing deviations.

The peripheral-based techniques tested on the TC399 during reading experiments did not suffer from this problem, and were able to correctly identify all the bits in all the packets received.

5.6.2 LPC Results

The benefits of our approach are much more evident on resource-constrained microcontrollers such as the NXP LPC11C24.

On this platform, writing techniques that employed hardware peripherals performed 5 to 10 times better than the basic bitbanging implementation, reaching the maximum bus bitrate of 1 Mbit/s while the bitbanging techniques could not be faster than 200 kbit/s. This goes to show the increased reliability that our techniques display with respect to traditional bitbanging on such platforms.

In particular, the bitbanging technique implemented with busy-waits was limited by the CPU clock speed and achieved a maximum baudrate of 200 kbit/s, while the timer implementation was limited by the timer frequency and achieved a maximum baudrate of 120 kbit/s. The SPI and UART writing techniques, on the other hand, were able to reach a bus speed of 1 Mbit/s, which is the maximum possible speed of the standard CAN bus. Finally, due to the restrictions that the protocol imposes, the I2C technique was implemented only for a specific bitrate (200 kbit/s), as described in Section 5.2.3.

Similar results were obtained when testing reading techniques. Equivalently to what has been described in Section 5.6.1, the bitrate of the bitbanging technique implemented with CPU busy-wait loops had to be further decreased to 100 kbit/s because of accumulated errors during the time of each bit, which were not compensated by any resynchronization technique. The timer implementation for reading was instead able to perform at the same speed as the writing implementation. On the other hand, SPI and UART techniques were able to achieve the full 1 Mbit/s speed on the bus, receiving each bit correctly on each message transmission.

5.7 Final Remarks

As a summary, with these experiments we have:

1. Validated the implementation of our framework
2. Proven the possibility of reading and writing whole messages bit-by-bit using conflicting peripherals
3. Proven the cross-platform nature of our techniques
4. Demonstrated that the techniques proposed in this work generally perform better than traditional bitbanging, especially on resource-constrained microcontrollers

These tests were conducted as a preliminary study on the reliability of the bit timings produced by peripheral techniques, and to demonstrate the possibility of using such techniques on a real CAN bus. In these tests, we did not provide an implementation of a specific attack, but rather demonstrated the general capabilities of each technique involved.

Further tests might be conducted on more realistic environments, e.g. with multiple CAN nodes, multiple unrelated messages and different timings, to test the performance of specific synchronization or bit injection attacks in the aforementioned cases.

Chapter 6

Conclusions

6.1 Conclusions

In this work, we have demonstrated that it is possible to communicate directly on the CAN bus using other peripherals that share the same pins as the CAN peripheral.

In particular, the SPI and ADC peripherals can be used unboundedly, while the UART and I2C peripherals impose stricter rules over what can be read or written to the bus, which does not constitute a problem when reading or writing small sequences of bits.

We have also implemented a new tool, called *CANPass*, that is able to deploy such techniques on different platforms, namely the LPC11C24 microcontroller and the AURIX TC399XP microcontroller.

The experimental results show that these techniques can be used to communicate with ordinary CAN nodes in a reliable way. Also, they provide better robustness than typical bitbanging techniques. This is especially evident on the low-end LPC microcontroller, where the speed of bit injection was dramatically increased by using hardware peripherals instead of software-level bitbanging.

In addition, we have shown that the proposed techniques can be implemented on unmodified microcontrollers without the need for additional hardware other than that which is strictly necessary to communicate on the CAN bus.

Since the CANPass framework provides a way to manipulate the CAN link-layer without recurring to a CAN controller, it can be used to violate the protocol's rules and perform link-layer attacks. Moreover, since it can be reliably implemented on different microcontrollers without modifying the hardware, it can be used to mount link-layer attacks on remotely compromised devices, which was previously thought to be impractical at high speeds.

These new possibilities for reliable bit injection at the CAN data link layer cast a new light on CAN link-layer attacks, making them more relevant from a security perspective and forcing researchers and vendors to reconsider their threat models regarding CAN attacks.

As a side note, we point out that, apart from enabling remote link-layer attacks, our framework can be also used to:

- reduce hardware requirements for physical attacks, since we have demonstrated that using conflicting peripherals can make a low-end and inexpensive microcontroller perform precise bit injection on a 1 Mbit/s CAN bus
- perform fault injection tests on already existing networks without adding further equipment

6.2 Limitations

The approach presented in this work has some notable limitations, some of which are common to any bit-injection technique on the CAN bus. In particular, because of how the CAN bus is physically implemented, any CAN bit-injection technique, including CANPass, has the following limitations:

- Only dominant bits can be forced on the bus: unless external hardware is used to short-circuit the CANH and CANL signals at a physical level, there is no way of "forcing" a recessive state on the CAN bus instead of a dominant state, due to how the signals are electrically constructed. Nevertheless, if no device is actively driving the bus signals, a recessive state can be transmitted without any obstacle.
- The microcontroller must communicate on the CAN bus through an on-chip CAN controller and an external CAN transceiver: if an external CAN controller is used, no bit injection nor bit sniffing technique can be employed. However, since this is by far the most common configuration used nowadays, we do not consider it a major downside for our approach.

In addition, we highlight some limitations that are specific to our approach, which are:

- There must be a conflicting peripheral: since our approach revolves around the possibility of exploiting pin conflicts between the CAN controller and other peripherals, if the targeted CAN controller has no conflict our method

cannot be applied. However, as we have shown in Section 3.4, this condition is often verified on automotive microcontrollers.

- Some of the techniques presented have hard requirements over the form of packets exchanged, which means that not every message can be read or written with these techniques, as already highlighted in Chapter 3 and Chapter 4.
- Although the CANPass framework has been designed to have as little overhead as possible, the need for extensibility leads to a tradeoff between these two aspects. While the CANPass implementation of the techniques described in this work is surely more extensible than an ad-hoc implementation for a specific microcontroller, the former will probably never beat the latter in terms of performance.

Finally, it is important to consider the fact that, being a newborn project, the CANPass framework currently provides the implementation for only two platforms. This is a limitation that can be solved in future work.

6.3 Future Work

As the CANPass framework is designed to be extensible, future work on this tool will primarily consist of increasing the number of available platforms and the number of techniques implemented. From the platforms side, some popular microcontrollers such as Arduino, ESP32, STM32 and PIC18 devices, which are not currently implemented, might be added to the framework.

For what concerns techniques, many other peripherals can be employed for communicating on the CAN bus, namely DAC peripherals, PWM generators, and any other serial interface either proprietary or standard. In addition, if a CAN peripheral is added to the hardware interface layer, the CANNon [22] technique might be implemented as a Sender technique. The *loading* of the CANNon, as it is called in the paper, can be implemented in the `setMsgToSend()` function, while the *firing* can be done with the `sendMsg()` function.

Finally, some additional work should be done to further research the capabilities of our approach. In particular, testing this approach on a real vehicle and demonstrating its stealthiness against state-of-the-art Intrusion Detection Systems can greatly increase the credibility of our work in the context of real-world attacks.

Bibliography

[1] R. Beek, C.; Samani. Defcon - connected car security, 2017.

[2] Bitbane. Cant. DEFCON 26 - CAR HACKING VILLAGE, 2018.

[3] Gedare Bloom. Weepingcan: A stealthy can bus-off attack. *Workshop on Automotive and Autonomous Vehicle Security (AutoSec) 2021*, 2021, 02 2021.

[4] Mehmet Bozdal, Mohammad Samie, Sohaib Aslam, and I.K. Jennions. Evaluation of can bus security challenges. *Sensors*, 20:16–17, 04 2020.

[5] Mehmet Bozdal, Mohammad Samie, and Ian Jennions. A survey on can bus protocol: Attacks, challenges, and potential solutions. In *2018 International Conference on Computing, Electronics Communications Engineering (iCCECE)*, pages 201–205, 2018.

[6] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 6, USA, 2011. USENIX Association.

[7] Kyong-Tak Cho and Kang G. Shin. Error handling of in-vehicle networks makes them vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1044–1055, New York, NY, USA, 2016. Association for Computing Machinery.

[8] John Donovan. What engineers need to know when selecting an automotive-qualified mcu for vehicle applications, 2014.

[9] Miro Enev, Alex Takakuwa, Karl Koscher, and Tadayoshi Kohno. Automobile driver fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2016, 01 2016.

[10] BARR Group. 2018 embedded systems safety & security survey. [Online]. Available: https://barrgroup.com/embedded-systems/market-surveys/2018-safety-security.

[11] CAN in Automation. Can lower- and higher-layer protocols, 2021.

[12] Infineon. Aurix development studio ide. [Online]. Available: https://www.infineon.com/cms/en/product/promopages/aurix-development-studio/.

[13] Infineon. Sak-tc399xp mcu. [Online]. Available: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/aurix-family-tc39xxx/sak-tc399xp-256f300s-bd/.

[14] Infineon. Tc399 evaluation board. [Online]. Available: https://www.infineon.com/cms/en/product/evaluation-boards/kit_a2g_tc399_5v_trb/.

[15] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[16] ISO Central Secretary. Road vehicles — controller area network (can) — part 2: High-speed medium access unit. Standard ISO 11898-2:2003, International Organization for Standardization, Geneva, CH, 2003.

[17] ISO Central Secretary. Road vehicles — controller area network (can) — part 1: Data link layer and physical signalling. Standard ISO 11898-1:2015, International Organization for Standardization, Geneva, CH, 2015.

[18] K. Johansson, Martin Törngren, and L. Nielsen. Vehicle applications of controller area network. In *Handbook of Networked and Embedded Control Systems*, 2005.

[19] Kaveh Bakhsh Kelarestaghi, Mahsa Foruhandeh, Kevin Heaslip, and Ryan Gerdes. Intelligent transportation system security: Impact-oriented risk assessment of in-vehicle networks. *IEEE Intelligent Transportation Systems Magazine*, 13(2):91–104, 2021.

[20] H. Kimm and H. Ham. Integrated fault tolerant system for automotive bus networks. In *2010 Second International Conference on Computer Engineering and Applications (ICCEA 2010)*, volume 1, Los Alamitos, CA, USA, mar 2010. IEEE Computer Society.

[21] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, 2010.

[22] S. Kulandaivel, S. Jain, J. Guajardo, and V. Sekar. Cannon: Reliable and stealthy remote shutdown attacks via unaltered automotive microcontrollers. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 195–210, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[23] Jean Kumaga. A day in the life of digi-key. *IEEE Spectrum*, 51(2):50–56, 2014.

[24] Microchip. Mcp2551 can transceiver. [Online]. Available: https://www.microchip.com/en-us/product/MCP2551.

[25] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. DEFCON 21 (2013), 2013.

[26] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. IOActive technical whitepaper, 2014.

[27] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. Black Hat USA 2015, 2015.

[28] Pal-Stefan Murvay and Bogdan Groza. Dos attacks on controller area networks by fault injections from the software layer. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: Hacking tesla from wireless to can bus. *Briefing, Black Hat USA*, 25:1–16, 2017.

[30] NXP. Lpc11c00 microcontroller family. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1100-cortex-m0-plus-m0/scalable-entry-level-32-bit-microcontroller-mcu-based-on-arm-cortex-m0-cores:LPC11C00.

[31] NXP. Lpcxpresso board. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-

microcontrollers/general-purpose-mcus/lpc1100-cortex-m0-plus-m0/lpcxpresso-board-for-lpc11c24-with-cmsis-dap-probe:OM13093.

[32] NXP. Mcuxpresso ide. [Online]. Available: https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE.

[33] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A stealth, selective, link-layer denial-of-service attack against automotive networks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, volume 10327 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2017.

[34] PulseView. Pulseview webpage. [Online]. Available: https://sigrok.org/wiki/PulseView.

[35] TASKING. Tasking tricore tools. [Online]. Available: https://www.tasking.com/products/tricore-vx-software-development-tools.

[36] Ken Tindell. Canhack. [Online]. Available: https://github.com/kentindell/canhack, 2018.

[37] A. Waibel. The art of bit-banging: Gaining full control of (nearly) any bus protocol. [Online]. Available: https://www.youtube.com/watch?v=sMmc0hSi5rs, 2016.

[38] Li Yue, Zheming Li, Tingting Yin, and Chao Zhang. Cancloak: Deceiving two ecus with one frame. *Workshop on Automotive and Autonomous Vehicle Security (AutoSec) 2021*, 2021, 02 2021.

[39] Haichun Zhang, Xu Meng, Xiong Zhang, and Zhenglin Liu. Cansec: A practical in-vehicle controller area network security evaluation tool. *Sensors*, 20(17), 2020.